Concept-based Querying in Mediator Systems

Kai-Uwe Sattler Ingolf Geist Eike Schallehn

Department of Computer Science
University of Magdeburg, P.O. Box 4120, 39016 Magdeburg, Germany
{kus|geist|eike}@iti.cs.uni-magdeburg.de

Abstract

One approach to overcome heterogeneity as part of data integration in mediator systems is the usage of meta data in form of a vocabulary or ontology in order to represent domain knowledge explicitly. This requires to include this meta level during query formulation and processing. In this paper, we address this problem in the context of a mediator which uses a concept-based integration model and an extension of the XQuery language called CQuery. This mediator has been developed as part of a project for integrating data about cultural assets. We describe the language extensions, their semantics as well as the rewriting and evaluation steps. Furthermore, we discuss aspects of caching and keyword-based search in support of an efficient query formulation and processing.

1 Introduction

Providing an integrated access to heterogeneous data sources from the Web is still a big challenge. Here, several issues arise such as autonomy, heterogeneity as well as scalability and adaptability with regard to a great number of – possibly changing – data sources. Suitable solutions range from simple meta search engines over materialized approaches to mediator systems which answer queries on a global schema by decomposing them, forwarding the sub-queries to the source systems and combining the results into a global answer.

One can classify mediator systems roughly based on the kind of correspondence specification between the local schemas of the sources and the global mediator schema. Using a Global-as-View approach (GAV) the mediator schema is defined as a view on the local schema. In contrast, with the Local-as-View (LAV) approach we start with a global schema – that could be derived from a domain or reference model – and define the local schemas as views on it, i.e., a local schema contains only a subset in terms of the schema elements as well as the extensions. Query processing (or more exactly query rewriting) is simpler for the GAV approach, but the LAV simplifies adding or removing sources, because only correspondences between the global and the particular local schema have to be considered.

In mediator systems of the first generation integration is achieved mainly on a structural level. Data from the diverse sources are combined based on structural correspondences such as membership in classes of the same structure or the existence of common attributes. This works well in more or less homogeneous domains. In scenarios characterized by rather disjunct domains this approach leads to a great number of global classes that again requires detailed domain knowledge in order to be able to formulate the resulting more complex queries.

An alternative is the explicit modeling and usage of domain knowledge in form of semantic meta data, i.e. a vocabulary, a taxonomy, a concept hierarchy, or even an ontology. Similar efforts are known from the Semantic Web community, where a knowledge-based processing of Web documents is to be achieved by adding a semantic layer containing meta data. First results of this work mainly include models and languages for ontologies, e.g. RDF Schema (RDFS) and DAML+OIL, as well as the corresponding technologies. Because of the strong relationships a combination of Semantic Web and mediator approaches seems very promising. However, a special requirement from data integration is to define a mapping from the ontology layer to the source data, i.e., to specify how a data source supports a certain concept from the ontology both in a structural as well as a semantic way. This information is necessary for query rewriting and decomposition and has to be provided as part of the registration of a source.

Based on this observation, in this paper we present the YACOB mediator that uses domain knowledge in the form of concepts and their relationships for formulating and processing queries. This system has been developed for providing integration and query facilities in databases on cultural assets that were lost or stolen during World War II. The main contribution of our work is *(i)* the definition of a RDF-based meta model combining the representation of concepts as terminological anchors for the integration with information describing the mapping between the global concepts and the local schemas using a LAV approach, *(ii)* the query language CQuery supporting queries both on concept as well as instance level, and *(iii)* the presentation of operators and strategies for rewriting, decomposing and executing global queries including caching and keyword-based search.

The remainder of this paper is organized as follows. In Section 2 we introduce the integration model based on RDFS and XML. In Section 3 we present the query language by means of examples, describe the operators and their semantics as well as the process of query evaluation. An important aspect of a distributed query system for achieving good performance is the caching of query results. Here, the concepts of the semantic layer provide a good anchor for managing semantic caches. Thus, we discuss the realization of such a semantic cache in the context of the concept-based mediator in Section 4. In order to provide interactive query facilities in a search engine like manner our approach supports a keyword search, where a query is derived from a user-given set of keywords based on an index that maps keywords to schema objects. This technique is described in 5. Section 6 gives an overview of the overall architecture of the system and its implementation. After a discussion of related work in Section 7 we conclude the paper and point out to future work.

2 A Concept-based Model for Data Integration

For representing data that have to be integrated as well as the associated semantic meta data two levels have to be taken into account: *(i)* the actual data or instance level comprising the data managed by the sources and *(ii)* the meta or concept level describing the semantics of the data and their relationships.

2.1 Data and Concept Model

As the data model we use in our work a simple semistructured model based on XML. That means, data are represented in XML in the mediator as well as during exchange between mediator and the wrappers/sources. Furthermore, we assume data sources exporting data in XML and are able to answer simple XPath expressions. The necessary transformations from XPath to SQL or other query languages are simplified by using only a subset of XPath, e.g. avoiding function calls. The exported query results can be structured according to any DTD – the transformation into a global schema is performed by the mediator.

The model for representing concepts is based on RDF Schema. The Resource Description Framework (RDF) – developed by the W3C as a mechanism for creating and exchanging meta data for Web documents – is a simple graph-based model, where nodes model resources or literals and edges represent properties. Based on this, RDF Schema (RDFS) defines primitives for specifying vocabularies such as classes or class relationships. Though, there is a principal similarity with traditional data models, RDFS is characterized by some important features, e.g., properties are defined independently from classes and are restricted in their association with certain classes only by specifying domain constraints.

Using modeling primitives like *Class*, *Property* or *subClassOf* RDFS is suitable for developing basic vocabularies or ontologies. An example from the considered application domain is shown in Fig. 1. A concept in our mediator model corresponds to a class in RDFS, concept properties correspond to RDFS properties. Relationships between concepts beyond the standard RDFS properties such as *subClassOf* are also modeled as properties, where the domain and range are restricted to concepts.

In this way, the concept layer plays the role of the global mediator schema, but makes a more advanced semantic modeling possible. For instance, beside schema information in form of classes or concepts we can even represent instances in form of values. For this purpose, we introduce categories as a special kind of a RDFS class which has – in contrast to a concept – no associated extension. A category can be understood as a term represented in different sources by different values. Like concepts, categories can be organized in hierarchies using the *subClassOf* relationship. An example of the usage of categories is the property “portrays” of concept “fine arts” shown in Fig. 1. The domain of this property is described by the category “motif”, for which additional sub-categories are defined. In this way, information is explicitly represented that would be otherwise hidden in the data. For example, in one source the categories from Fig. 1 (or a subset of them) could be represented by strings such as "landscape" or "still life" (or in any other language) whereas in another source other coding schemes could be used. But at the global level the user is faced only with the category terms.

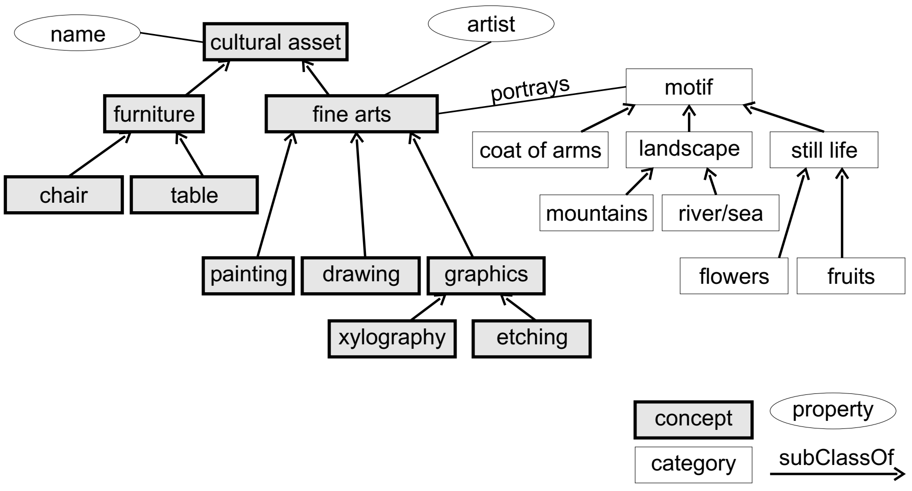


Figure 1: Concept Hierarchy

In summary, our model relies primary on the RDF Schema primitives (classes and properties) and extends these by the notion of concept and category as specialization of classes.

We can formally define the concept and data model as follows. Let *URI* be the set of Uniform Resource Identifiers, *Name* the set of valid identifiers (names) and $\mathcal{L}$ the set of literals. We denote the set of classes by $\mathcal{T} = \mathit{URI} \times \mathit{Name}$, comprising a unique URI and a class name. Furthermore, we can distinguish classes into

- concepts (denoted by $\mathcal{C} \subset \mathcal{T}$) as classes for which extensions are available in the data sources,
- categories (denoted as $\mathcal{V} \subset \mathcal{T}, \mathcal{V} \cap \mathcal{C} = \emptyset$), which represent abstract property values and are used for semantic grouping of objects, but have no extensions.

With concepts we can associate properties $\mathcal{P}$ with $\mathcal{P} = \mathit{Name} \times \mathcal{C} \times \{\mathcal{T} \cup \mathcal{L}\}$, consisting of an identifier, the concept to which the property is associated, as well as a class as instance domain or a literal. Moreover, we introduce a specialization relationship **is_a** $\subseteq \mathcal{T} \times \mathcal{T}$ representing the *subClassOf* relationship where it holds for two concepts $c_1, c_2 \in \mathcal{C} \subset \mathcal{T}$: If c_2 **is_a** c_1 (c_2 is derived from c_1) then $\forall (p, c_1, x) \in \mathcal{P} : \exists (p, c_2, x) \in \mathcal{P}$. In addition, we assume disjunct hierarchies for concepts and categories.

The data model of our approach is defined following the semistructured model OEM [GMPQ+97]. Let be $\mathcal{O} = \mathit{ID} \times \mathit{Name} \times \{\mathcal{L} \cup \mathit{ID} \cup \mathbb{P}\mathit{ID}\}$ the set of all objects, where $(\mathit{id}, \mathit{name}, v) \in \mathcal{O}$ consists of a unique object identifier *id*, an element name *name* and a value *v*. Here, *v* can represent an atomic value, an object identifier (representing an object reference), or a set of object identifiers. The extension **ext** : $\mathcal{C} \rightarrow \mathcal{O}$ of a certain concept $c = (\mathit{uri}, \mathit{name})$ comprises a set of instances with an element name equal to the concept name and a set of identifiers *v* corresponding to the properties defined for the concept:

$$\mathbf{ext}(c) = \{o = (\mathit{id}, \mathit{elem}, v) \mid \mathit{elem} = c.\mathit{name} \wedge \forall (p, c, c') \in \mathcal{P} : \exists i \in v : i.\mathit{elem} = p\}$$

Note, that this data model is used mainly internally and for exchanging data and queries between the sources and the mediator. The global query formulation is based completely on the concept model.

2.2 Concept Mapping

An important part of the mediator model is the specification of the mapping to the local schemas of the sources, i.e. the description how a source provides data for a given concept and how the structure of the concept (the set of properties) is mapped to local properties. Here, we *(i)* describe concepts and properties independently according to the RDFS paradigm, and *(ii)* follow the LAV approach, i.e., a source schema is defined as a view on the global (concept) schema.

Based on this, both a concept and a property mapping description are needed, which are represented by RDFS classes, too. An instance of the concept mapping class specifies, how the associated concept is supported by the data source. Here, "supporting a concept" means a source provides a subset of the extension of the given concept. In this context, we can distinguish two cases:

(a) the data source provides objects containing all properties defined by the global concept,

(b) the data source provides only partial information, i.e., the instances should be completed by trying to retrieve the missing property values from other sources supporting the same concept but possibly with other complementary properties.

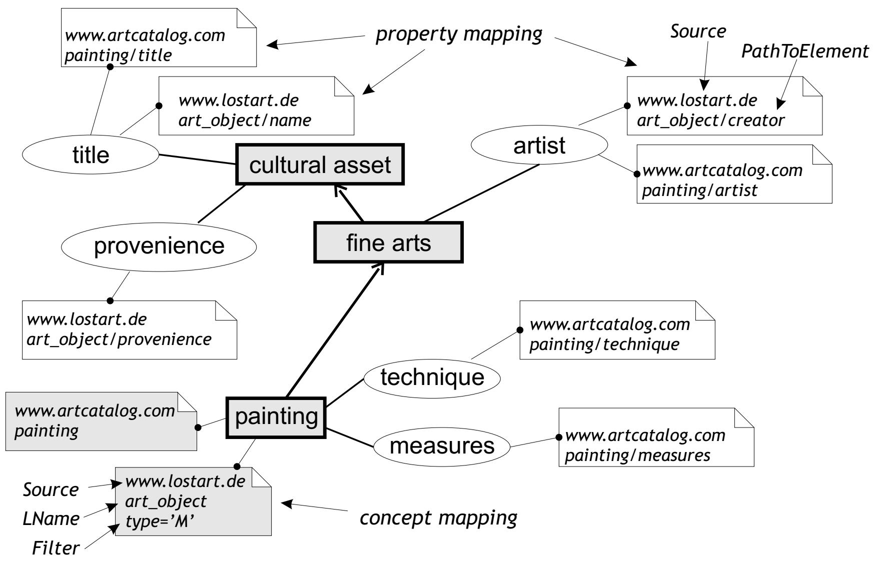
For both cases the concept mapping class *CM* formally consists of the components source name, local element name, and optionally a filtering predicate: $CM = (Source, LName, FilterPredicate)$. Using the source name, the source can be identified when instances of the associated concept are to be retrieved. The local element name denotes the XML element used in the source for representing the instances of this concept. In addition, the filtering predicate in the form of an XPath expression allows to further restrict the instance set. This could be necessary if in a source several concepts are mapped to the same extension (i.e., to the same XML element), but with a distinguishable property value (e.g., `type = 'Painting'`).

The property mapping *PM* defines the mapping between the property of a concept and an XML element or XML attribute of source data. It consists of the source name and an XPath addressing the representing element: $PM = (Source, PathToElement)$.

For a category we assume that it is represented in a data source only by a simple literal. Therefore, a property defined with a category as domain may have in a given data source only the corresponding literal value or the values corresponding to the sub-categories. Such a mapping is specified by a value mapping *VM* consisting of the source name and the source-specific literal: $VM = (Source, Literal)$.

Using these several mappings the elements of the concept schema are annotated. To each concept supported by a given source, the appropriate concept mappings as well as the property and value mappings are associated. Due to the usage of specialization relationships between concepts, it is not necessary to annotate every concept. Instead mappings are given only for concepts which represent leafs of the hierarchy with regard to the source (Fig. 2).

This example also shows, that a source has not to provide complete data in terms of the concept definition. If one source does not support all elements referenced in a query, the mediator tries to supplement this data using data from another source by applying an outer union operation. The potential sources can be identified with the help of the property mapping information. If no appropriate source is found, the missing data cannot be retrieved and the global object remains incomplete, i.e., the returned XML element contains not all sub-elements as required by the concept definition.



property mapping
Source
PathToElement
www.artcatalog.com painting/title
www.lostart.de art_object/name
www.lostart.de art_object/creator
www.artcatalog.com painting/artist
title
cultural asset
artist
fine arts
provenience
www.lostart.de art_object/provenience
www.artcatalog.com painting/technique
technique
www.artcatalog.com painting
painting
measures
www.artcatalog.com painting/measures
Source
LName
Filter
www.lostart.de art_object type='M'
concept mapping

Figure 2: Mapping Information in the Concept Hierarchy

Using the mapping specifications for each source we are able to resolve description conflicts (naming conflicts of elements), structural conflicts as well as data conflicts. However, the latter is supported only for categories because in this case a mapping between values is performed. The necessary transformations are performed as part of the query rewriting using the mapping information and during the processing of results by applying source-specific XSLT rules. These XSLT rules can be automatically derived using the following rules.

(1) For a concept mapping to concept c defining a local element `lelem`, the following XSLT template is generated:

```
<xsl:template match="lelem">
  <c> <xsl:apply-templates /> </c>
</xsl:template>
```

(2) For a property mapping to property p specifying a path `elem1/elem2/elem3`, a corresponding XSLT template of the following form is derived:

```
<xsl:template match="elem1">
  <p> <xsl:value-of select="elem2/elem3" /> </p>
</xsl:template>
```

(3) For a property p with a domain consisting of the hierarchy of categories $k_1, k_2, \ldots, k_m$ with associated value mappings $VM_1, \ldots, VM_m$ specifying the literals $v_1, \ldots v_m$, the following XSLT template for the property mapping is created:

```
<xsl:template match="elem1">
  <p> <xsl:choose>
    <xsl:when test="elem2/elem3 = 'v1'">
      <xsl:text>k1</xsl:text>
    </xsl:when>
    <xsl:when test="elem2/elem3 = 'v2'">
      <xsl:text>k2</xsl:text>
    </xsl:when>
    ...
  </xsl:choose> </p>
</xsl:template>
```

During the registration of a source the mappings for the supported concepts, their properties and categories are specified by the administrator. Based on this, the source-specific XSLT rules can be derived automatically.

3 The CQuery Language

The data integration model introduced in the previous section leads to the following two issues with respect to planning and executing queries:

- Operations are required which are applicable both to concept level as well as to data level and allow a transition between these levels.
- A global query has to be rewritten and decomposed using the mapping descriptions in a way that allows to identify the sources providing data to the queried concepts and to process the sub-queries autonomously by the appropriate source system.

In the following we first present the language by explaining some example queries and describe the operators and the steps of query processing afterwards.

3.1 CQuery by Example

For formulating queries using this model the query language CQuery – a derivative of XQuery – is provided. The main features of this language are essentially at semantic level, for the syntax the FLWR notation from XQuery is used. A query in CQuery consists of the following components: *(i)* selection of concepts satisfying a certain condition or by applying operations such as traversing relationships or set operations, *(ii)* obtaining and filtering data as instances of the previously chosen concepts, and *(iii)* combining and projecting the results. Thus, a typical query looks as follows:

Q_1 :
```
FOR $c IN concept[name='painting']
LET $e := extension($c)
WHERE $e/artist = 'van Gogh'
RETURN
  <picture>
    <title>$e/title</title>
    <artist_name>$e/artist</artist_name>
  </picture>
```

This query returns an XML document consisting of `picture` elements built from the title and artist name of paintings created by Vincent van Gogh. The meanings of the clauses are as follows. In the **FOR** clause the concepts are chosen. The pseudo-element `concept` is interpreted as document tree containing all defined concepts, where the concept properties are sub-elements of their concept. In this way, the language mechanisms from XQuery can be used to query the concept level as well. Thus, dedicated language extensions as proposed for example in the context of RDF query languages are not necessary. Beside selections on concept properties the supported operations include among others set operations like **UNION**, **EXCEPT**, and **INTERSECT** between sets of concepts as well as traversing concept relationships. Relationships are handled as elements, too. Therefore, the following expression

```
concept[name='fine arts']/!subClassOf
```

returns for the concept schema from Fig.1 the set of all concepts derived directly from the concept “fine arts”. The prefix “!” denotes the inverse relationship to `subClassOf` (not explicitly named here). In contrast, the suffix “+” specifies the computation of the transitive closure regarding this relationship. For example, the expression

```
concept[name='fine arts']/!subClassOf+
```

would return the set of all concepts directly or indirectly derived from “fine arts”. Because `!subClassOf+` is often used in queries for finding all concepts derived from a given concept, there is the shorthand “`*` for this operation. This means that for the query containing `concept[name='fine arts']/*` not only the sources are chosen supporting exactly the concept “fine arts” but all sources representing concepts derived from this.

With the **LET** clause the transition from the concept level to the instance level is specified. For this purpose, the predefined function `extension()` can be used, which returns the extension of a given concept, i.e. the set of all instances of the supporting sources. Because the **FOR** clause represents an iteration over the concept set, the function is applied to each single concept using the concept variable. The result set – a set of instances which is again bound to a variable – can be further filtered by the condition specified as part of the optional **WHERE** clause. Here, components of elements (i.e., sub-elements in terms of XML) are addressed by path expressions, too. Thus, `$e/artist` in query Q_1 denotes the value of property `artist` of the object currently bound to variable `$e`. If in the **FOR** clause more than one concept is assigned to the variable, the union of the several extension sets determined by applying `extension()` is computed. We call this operation in the following *extensional union*.

Whereas operations in the **FOR** clause are applied only to global meta-data at the mediator, the evaluation of the `extension()` function as well as of the filtering conditions in the **WHERE** clause require access to the source systems. Thus, the `extension()` function initiates the query processing in the source system – eventually combined with the filter condition as part of the source query. A join operation can be formulated by declaring more than one variable for extension sets in the **LET** clause and by specifying an appropriate condition in the **WHERE** part:

Q_2 :
```
FOR $c1 IN concept[name='sculpture']/*,
    $c2 IN concept[name='collection']
LET $e1 := extension($c1), $e2 := extension($c2)
WHERE $e1/exhibition = $e2/museum
RETURN
  <sculpture>
    <title>$c1/name</title>
    <museum>$e2/name</museum>
    <location>$e2/city</location>
  </sculpture>
```

The different extensions can be obtained from the same or different concept variables. The first case corresponds to a self join, the latter one represents an ordinary join. For example, the above query returns a set of elements describing sculptures together with information about the location where they are exhibited. Depending on the concept mapping, this information could be obtained from different sources.

The **LET** part of a query allows not only to obtain the extension of a concept. In addition, queries on properties can be formulated by using the pseudo-element `property` as a child node of a concept referring to the set of all properties of the given concept. If the obtained properties are bound to a variable, this variable can be used for instance selection.

Q_3 :
```
FOR $c IN concept[name='painting']/*
LET $e := extension($c), $p := $c/properties
WHERE $e/$p = 'flowers'
  ...
```

This corresponds to a disjunctive query including all properties. In this way, we support schema operations, i.e. operations on meta-data or more exactly on properties. Such operations are particularly useful in query languages for heterogeneous databases in order to resolve structural conflicts.

In a similar manner one can refer to a category associated with a property. This is achieved in the **LET** clause by assigning a path expression containing this property to a variable. Then, this variable holds a set of categories and can be used in the **WHERE** clause instead of a concrete value:

Q_4 :
```
FOR $c IN concept[name='painting']/*
LET $e := extension($c),
    $k := $c/portrays[name='still life']
WHERE $e/portrays = $k
  ...
```

In this query the set of categories for "still life" and the specializations of this is bound to the variable `$k`. During query rewriting the occurrence of `$k` in the **WHERE** clause is substituted by the source-specific value for "still life" or – in the case of more than one category – by a disjunctive condition consisting of all translated category values.

Finally, the **RETURN** clause has the same meaning as in XQuery: it allows to restructure the query result according a given XML document structure, where the result elements (i.e., the concepts and their instances) are referenced by the variables.

3.2 Query Processing

The semantics of the operations of the CQuery language can be easily defined based on an extended query algebra. Using the definitions of the concept and data model from Section 2 two kinds of operators have to be considered following operators from the relational algebra. The operators for the concept level include the selection Σ_{Cond}, the set operators $\cup$, $\cap$, $-$ and the path traversal Φ_P, which returns for the concept of the set C all concepts referenced using the relationship p:

$$\Phi_p(C) = \{c' \mid \exists c \in C : (p, c, c') \in \mathcal{P}\}$$

The operation for traversing the inverse relationship can be formulated as follows:

$$\Phi_{\overline{p}}(C) = \{c' \mid \exists c \in C : (p, c', c) \in \mathcal{P}\}$$

Furthermore, we introduce the operation Φ_p^+ returning the transitive closure of a concept (or concept set respectively) regarding a given relationship p:

$$\Phi_p^+(C) = \{c' \mid \exists c_s \in C : (p, c_s, c') \in \mathcal{P} \vee \exists c_i \in \Phi_p^+(\{c_s\}) : (p, c_i, c') \in \mathcal{P}\}$$

For the data level, the standard algebra operations selection (σ), projection (π) and Cartesian product ($\times$) are provided.

Now, a query formulated in CQuery can be translated into an algebra expression consisting of the above introduced operators using the following rules:

1. The clause **FOR** `$c` **IN** `concept[`*Cond*`]` is translated into an expression $\Phi^+_{\overline{\textbf{is\_a}}}(\Sigma_{Cond}(\mathcal{C}))$, where $\overline{\textbf{is\_a}}$ means the inverse relationship for c_2 **is_a** c_1.

2. Traversal operations like `concept[`*Cond*`]/`*prop*$_1$`/.../`*prop*$_n$ are translated into $\Phi^+_{\overline{\textbf{is\_a}}}(\Phi_{prop_n}(\ldots \Phi_{prop_1}(\Sigma_{Cond}(\mathcal{C}))).$

3. Set expressions such as `concept[`*Cond*$_1$`]` **UNION** `concept[`*Cond*$_2$`]` are translated into:

$$\Phi^+_{\overline{\textbf{is\_a}}}(\Sigma_{Cond_1}(\mathcal{C})) \cup \Phi^+_{\overline{\textbf{is\_a}}}(\Sigma_{Cond_2}(\mathcal{C}))$$

4. The expression **LET** `$e := extension($c)` **WHERE** *Cond* is translated by employing the expression *CExpr* as the translation of the expression bound to the variable `$c` in the following expression:

$$\biguplus_{c \in CExpr} \sigma_{Cond}(\textbf{ext}(c))$$

   Here, $\biguplus$ means an outer union. This operation permits to combine partially overlapping data sources even with only a partial union compatibility. For this purpose, we assume that the attributes common to all participating sources act as identifying keys. As the consequence, a tuple, that occurs in every input relation with its key value, appears only one time in the result set (as a combination of the individual source tuples). Otherwise the missing elements are omitted.

5. If more than one variable is declared in a **LET** clause for extensions and/or category and property sets, these sets are combined by applying the Cartesian product.

6. The **RETURN** clause is translated into a projection expression π that may contain XML tags, literals and path expression referring to attributes.

By applying these rules the following example query

```
Q5 : FOR $c IN concept[name='painting']/*
     LET $e := extension($c)
     WHERE $e/artist = 'van Gogh'
     RETURN
       <painting>
         <title>$e/title</title>
         <artist>$e/artist</artist>
       </painting>
```

is translated into the following expression where *Proj* stands for the projection expression from Q_5:

$$\biguplus_{c \in \Phi^{+}_{\textbf{is\_a}}(\Sigma_{\text{name='painting'}}(\mathcal{C}))} \pi_{\mathit{Proj}}(\sigma_{\text{artist='van Gogh'}}(\textbf{ext}(c)))$$

Translating a query formulated in CQuery into the internal algebra representation is also the preparation step for the query processing illustrated in Fig. 3. This algorithm processes basic query expressions on concepts sets and their associated extensions. Hence, queries involving Cartesian products or joins have to be decomposed in advance in order to process them using this algorithm. Furthermore, this step includes simple algebraic optimizations such as push-down of selections as well as substituting accesses to schema operations (as in query Q_3) by disjunctive conditions and substituting category variables by the corresponding value set (see query Q_4).

The first step in the processing algorithm is the evaluation of the concept level operators. For each of the selected concepts we try to answer the query expression *IExpr* referring to the extensions of the concepts by determining the relevant concept mappings and in this way the sources which have to be queried. Next, we try to answer the sub-query for each source from the cache (see Section 4). If this fails, the query is translated and sent to the source. However, due to possible overlapping between different concepts of one source we have to eliminate redundant concepts. This is performed using the function $\mathrm{cmin}(C, s)$, which returns for a given source s and a set of concepts C the minimal subset of concepts based on the following heuristics.

Elimination. Considering concept hierarchies as class hierarchies with extensional relationships we could omit the query for obtaining the extension of c_2 if c_2 **is_a** c_1 because of $\textbf{ext}(c_2) \subseteq \textbf{ext}(c_1)$. However, in our scenario concept hierarchies are not the result of schema integration and extensional analysis. Hence, the above assumption does not hold in every case but only if both concepts are supported by the same source and are mapped to the same class. This can be checked using the concept mapping $\mathit{CM}(c)$: c_2 **is_a** $c_1 \wedge \mathit{CM}(c_1).\mathit{LName} = \mathit{CM}(c_2).\mathit{LName} \wedge \mathit{CM}(c_1).\mathit{Source} = \mathit{CM}(c_2).\mathit{Source} \Rightarrow \textbf{ext}(c_2) \subseteq \textbf{ext}(c_1)$. In this case, we can eliminate the sub-query for c_2.

Merging. Sub-queries on concepts from parallel branches of a concept hierarchy that refer to the same source can be merged if they correspond to the same class, i.e., if it holds $\mathit{CM}(c_1).\mathit{LName} = \mathit{CM}(c_2).\mathit{LName}$. In this case the possibly specified filtering conditions from the concept mappings have to be connected disjunctively.

```
Input:
    query expression in the form of $\biguplus_{c \in CExpr} IExpr(c)$
    result set $R := \{\}$
compute concept set $C := CExpr$
forall $c \in C$ do
    /* derive source query */
    determine all properties $p_1(i = 1 \ldots n)$ and categories $k_j(j = 1 \ldots m)$ referenced in IExpr
    forall $CM_s$ associated with $c$ do
        $s := CM_s(c).Source$
        /* consider only non-redundant concepts */
        if $c \in \mathrm{cmin}(C, s)$ then
            /* look for the query in the cache $\rightarrow$ result is denoted by $R_c$ */
            $R_c := \text{cache-lookup}(IExpr(c), CM_s(c))$
            if $R_c \neq \{\}$ then
                /* found cache entry */
                $R := R \uplus R_c$
            else
                /* construct a query for each supporting source */
                Translate IExpr(c) according to $CM_s(c)$, $PM_s(p_i)$, and $VM_s(k_j)$
                    into a source query $Q$
                process $Q$ at source $CM_s(c).Source$; result is denoted by $R_Q$
                $R := R \uplus R_Q$
            fi
        fi
    od
od
```

Figure 3: Steps of Query Processing

In the next step the remaining sub-queries are translated into source queries on the basis of the mappings. Here, the concept mapping provides the local element name, the property mappings provide the associated attribute or child element names and the value mappings are used to map the categories to source-specific values. So for example, a query expression of the form $\sigma_{P\theta v}(\textbf{ext}(c))$ is translated using the mappings *CM*(c) and *PM*(p) into the following XPath expression:

$$/\langle CM(\mathrm{c}).LName\rangle[\langle PM(\mathrm{p}).PathToElement\rangle\ \theta\ \mathrm{v}]$$

As shown in this example, source queries are only simple selections on local extension that can be answered even by non-DBMS based sources. Thus, more advanced operations like union or join are computed only at the global mediator level. Delegating these operations to the source systems would require to take into account the source capabilities as described e.g. in [RS97] but this is not considered here.

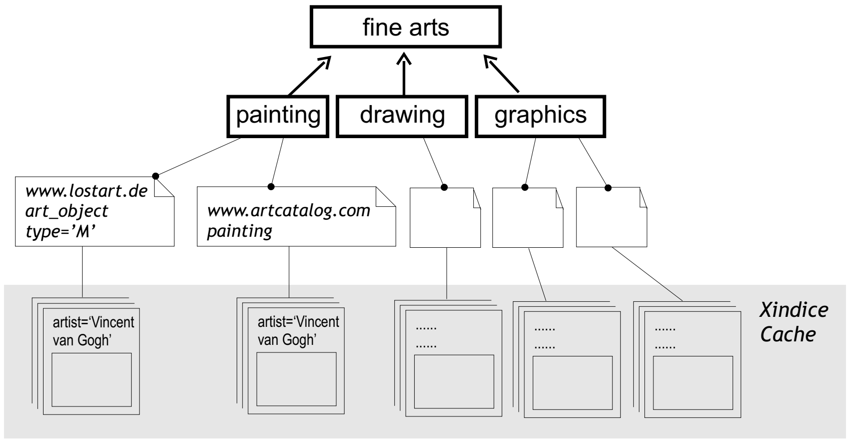


Figure 4: Overview of the YACOB caching approach

4 Caching of Query Results

The virtual integration of data sources described here implies extensive requirements regarding the efficiency of query processing. This results mainly from the fact, that sources are assumed to be connected via low bandwidth connections. Furthermore, the YACOB system was designed to be used in scenarios with interactive query formulation and refinement supported by according user interfaces as described later on in Section 5. Hence, two major concerns during the design of the system were to present query results to the user as quickly as possible, and to minimize the amount of redundant and irrelevant queries sent to the source systems. To address these issues a semantic caching strategy which is tightly coupled with the overall concept-based approach was developed. Roughly sketched, in our approach partial results provided by sources are stored along with a partial source query in a local XML database, capable of answering XPath queries for looking up cache entries or answering refined queries from the cache.

While caching in general is used to improve performance related to a number of communication and I/O-problems, the main intention of this strategy is to support interactive query refinement. If applicable, previous query results stored locally should be used either directly or to answer the query if additional restrictions were specified. Otherwise, the source systems would have to be queried during each new refinement iteration, resulting in a high number of redundant queries. Therefore, we consider the following typical aspects of query formulation.

Query restriction by removing queried concepts: this is for instance the case, if during query refinement the focus moves down a specialization hierarchy or queried concepts are removed explicitly using the **EXCEPT** operator on the set of queried concepts.

Query restriction by conjunctively adding predicates: to further reduce the size of a result set the selection condition can be narrowed by adding additional criteria.

Query expansion by adding queried concepts: by moving up a specialization hierarchy or explicitly adding concepts for a join or union, the set of concepts to be queried grows.

Query expansion by disjunctively adding predicates: this is the case, if, in addition to a specified selection criterion, an alternative selection is added to the previous result.

The two scenarios of adding or removing concepts conform to the result of a modified **FOR** clause. These are addressed simply by providing a caching mechanism below the concept level as shown in Fig. 4. Actually, there is a cache space for each concept mapping, i.e. for each concept provided by a source, allowing additional result accuracy in case of connection problems. This way, during query decomposition relevant cache spaces are determined automatically and the cache processing can be moved to the level of answering the selection part according to the `WHERE` clause.

As outlined in Section 3.2 the selection consists only of logically combined predicates. Furthermore, the expression is transformed to disjunctive normal form, so a cache entry consists of *(i)* a header describing the query in form of a set of predicates and some additional information for cache maintenance, and *(ii)* the cached result already transformed according to the global DTD. An example cache entry is:

```
<entry date="11/22/2002" time="6:30:05">
  <predicate>artist=Vincent van Gogh</predicate>
  <predicate>technique=Watercolor</predicate>
  <result>
        <painting technique='Watercolor'</painting>
            <artist>Vincent van Gogh</artist>
            <title>Drawbridge in Nieuw-Amsterdam</title>
            <technique>Watercolor</technique>
            ...
        </painting>
        ...
  </result>
</entry>
```

```
Input:
   concept mapping $CM_s(c)$
   query expression $IExpr(c)$
   result set $R := \{\}$

cache-lookup($IExpr(c), CM_s(c)$)
   $DNF :=$ disjunctive-normal-form($IExpr(c)$)
   connect to cache $CC$ for concept mapping $CM_s(c)$
   forall $conj \in DNF$ do
      create cache entry query $Q_{CC}$ from $2^{conj} - \{\}$
      perform $Q_{CC}$ on $CC$; candidate set is $R_{CC}$
      if $R_{CC} \neq \{\}$ then
         /* partial query can be answered from the cache */
         evaluate $conj$ over $R_{CC}$; result is $R_{conj}$
         remove $conj$ from $IExpr(c)$
         $R := R \cup R_{conj}$
      fi
   od
```

Figure 5: Cache evaluation during query processing

According to this cache entry format the cache is evaluated as shown in the algorithm in Fig. 5. The selection expression is transformed to disjunctive normal form, so we can consider the partial query a set of conjunctively combined predicates. The predicates are transformed to a trimmed string representation and used for looking up according cache entry candidates via XPath queries. Currently only predicate equality for attribute selections of the form *attribute='value'* is considered, i.e. there is no check for semantic containment on the predicate level. An example for the latter would be *year* $>$ *1970* is contained in *year* $>$ *1956*. The approach could be extended to support this, but for the current application scenario and the focus on interactive query refinement the implemented algorithm is sufficient.

Using the set of predicates we can decide if the result of the partial query is in the cache or can be answered from the cache. If the set is equal to a set of predicates described in a cache entry header, the corresponding previous result is returned. If the new predicate set is a superset of the predicates described in a cache entry, the cached result is a superset of the sought result and the query can be answered from this cache entry. The lookup of cache entries works as follows: all possible non-empty subset combinations are created and combined in a disjunctive XPath query, e.g. when trying to find a van Gogh painting of the drawbridge from the cache entry example, the corresponding cache lookup query is as follows:

```
/entry[count(predicate)=1 and
       predicate='artist=Vincent van Gogh']/result/* |
/entry[count(predicate)=1 and
       predicate='title=Drawbridge in Nieuw-Amsterdam']/result/* |
/entry[count(predicate)=2 and
       predicate='artist=Vincent van Gogh' and
       predicate='title=Drawbridge in Nieuw-Amsterdam']/result/*
```

This way it is guaranteed, that only cache entries suitable for answering the query are returned. In a second step, the full input query is evaluated over the returned result from step one. This is necessary, because additional predicates may be specified in the current query. As the partial query is answered from the cache, the conjunct can be removed from the query expression for further processing. If the result from step one was empty, the query has to be answered by the source. Otherwise, the result of step two is combined with results of all partial queries and returned as the partial result for the current source.

The cache entries are physically stored using Xindice, a native XML database capable of performing XPath queries over document collections. According to the previous descriptions a collection is maintained for each concept mapping registered with the YACOB system. Every result processed by the system is moved to the cache after applying the transformation and adding the cache entry header information including the current timestamp. Entries are removed automatically when the difference between the timestamp and the current time exceeds a fixed multiple of the average session time. This cleaning task is implemented using XMLObjects, a server-side extensibility feature of the Xindice server.

5 User interface and Search Strategies

One application domain of our approach is the integration of different databases on cultural assets ("lost art"). The users in this domain are mainly historians, lawyers or people looking for lost art. It is not acceptable for them to write a CQuery statement to get the required information. Thus, a graphical user interface has to be provided. Here, we can distinguish two main approaches of search interfaces: *(i)* concept navigating and browsing and *(ii)* keyword-based search.

The first approach, browsing the concept model, allows the user to refine her search by navigating in the concept and the category hierarchies. The system provides for each step the first results to give the user a first impression of the possible results. As most operations are a refinement of the previous query, the cache structures as described in Section 4 can be used extensively. The keyword-based search is the second way of searching the integrated data. In this case the user provides a set of keywords which is mapped to a CQuery by the system. In this paper we only describe the exact match case in the keyword-based search. The inexact keyword-based search requires several extensions which are out of the focus of this paper. Furthermore, it is possible to combine the keyword-based search and browsing approach. Here, the user starts with one of the both approaches and then refines the results by using a second approach. So, the advantages of both search strategies can be utilized.

5.1 Steps of the Keyword-Based Search

The problem of the keyword-based search is the formulation of an efficient CQuery statement. The user-given keywords correspond to the meta level consisting of concepts and categories as well as to the data level. This circumstance has to be reflected in the resulting CQuery query. A naive way to implement the keyword-based search is the following CQuery statement:

Q_6 :
```
FOR $c IN concept[name='Cultural Asset']
LET $e := extension($c), $p := $c/properties
WHERE $e/$p = 'keyword1' OR $e/$p = 'keyword2' OR ...
RETURN
  ...
```

This approach has two major disadvantages. First, query Q_6 evaluates over all concepts, all properties and all sources denying the system optimization opportunities. Second, the query does not use the meta level, but rather applies all keywords to the data level. Possible interesting facts cannot be found by the query. Following, we propose a more efficient way of creating a CQuery statement from a set of keywords by using a preprocessing step and a keyword index structure.

A keyword corresponds either to a concept name, a category name or a data value. Input of the keyword-based search is a set of keywords $KW \subseteq \mathit{Name}_C \cup \mathit{Name}_V \cup \mathit{DValues}$ with $\mathit{Name}_C \subset \mathit{Name}$ the set of all names of concepts, $\mathit{Name}_V \subset \mathit{Name}$ the set of all names of categories and *DValues* the set of data values in the sources. Furthermore, we assume for the next discussion following rules: $\mathit{Name}_C \cap \mathit{Name}_V = \emptyset$, $\mathit{Name}_C \cap \mathit{DValues} = \emptyset$ and $\mathit{Name}_V \cap \mathit{DValues} = \emptyset$.

As keywords belong to different groups the first task of the preprocessing is the classification of the keywords in the groups concept, category and data value. The membership to the first two groups can be decided by using the global concept model and the meta level query capabilities of CQuery. These classification steps correspond to the `ConceptFilter` and `CategoryFilter` as depicted in Fig. 6. The filter implementations are very efficient, because the model is stored on the global level in the mediator. The result of the `ConceptFilter` is a set of concepts. A set of ⟨property, category⟩ pairs is the result of the `CategoryFilter`. The `CategoryFilter` further adds concepts to the concept set, which are corresponding to the found properties.

After passing the both filters the keyword set contains henceforth only data values. These keywords can be applied to all properties of the selected concepts, which is still too inefficient. We further improve the query execution by using a *keyword index*, which is the third step illustrated in Fig. 6. The keyword index is similar to an inverted list index structure, but a list of ⟨property, concept⟩ pairs are assigned to a keyword instead of documents. Now, the system can find for a data value the place where it is stored logically in the concept model. The result of the `DataValueFilter` is a set of ⟨property, data value⟩ pairs. The filter further adds the found concepts to the concept set.

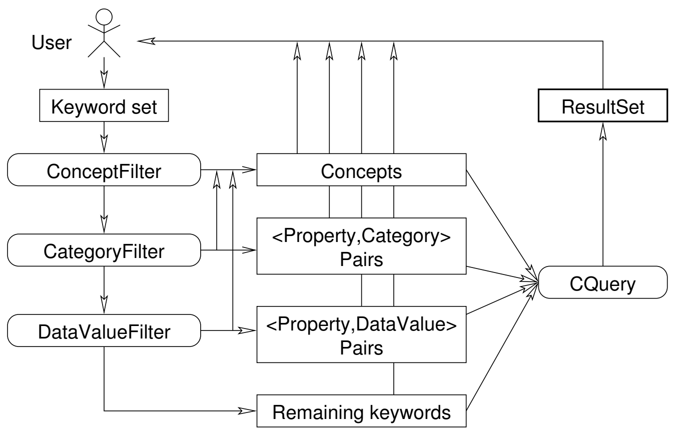


Figure 6: Keyword-based search: components and process

The index is not built complete in advance, because of the data origins from different sources. During the query processing the systems builds the index iteratively using the current query results. As the index is complete, it is possible that keywords are not found. These keywords are assigned to all properties of the already selected concepts. Unfortunately, the query statement can degenerate to the naive statement described in the beginning.

Finally, the system creates a CQuery statement and executes it. The query comprises the union of the most specialized concepts, i.e. all concepts that do not have a subclass specified in the concept set, the computation of the extension as well as of the categories, and a selection in the **`WHERE`** clause. The system presents the result as well as the selected concepts and categories to the user. Now, the user has a starting point for a possible further research. The complete process is depicted in Fig. 6.

5.2 Structure and Update of the Keyword Index

To support the keyword-based search in the YACOB system, an entry of the keyword index has the structure $\langle dv, p, c\rangle$ with dv is a data value, p is a property of a concept c in which dv occurs. Thus, the position of a keyword is determined exactly on the global level. The index is implemented with help of a relational database system. There are different ways of storing such an index in a relational database. As the approach is similar to a inverted list we can use approaches of this area e.g. [MRYGM01]. Another approach is using index structures for keyword-based search in relational [ACD02, BHN^{+}02, HP02] or XML databases [FKM00]. For now, we assume that the index is implemented as one table with the attributes `data_value`, `property` and `concept`. An index is created over the column `data_value` to allow efficient match and partial match query evaluation. Fig. 7 shows several example entries.

As the data come from different heterogeneous, autonomous sources, the index is not built up in advance. Rather, the index is created iteratively. Hereby, the index is updated periodically by using information from the query cache. Thus, the query performance (speed and accuracy) is increasing by using the system.

data_value	property	concept
van Gogh	artist	drawing
van Gogh	artist	painting
van Gogh	title	books
...	...	...

Figure 7: Keyword index: Example entries

5.3 Keyword-Based Search Example

Summarizing the section, an example shall illustrate the different steps of the keyword-based search as well as their results. Assume a user provides the keyword set $kset = \{$painting, flowers, van Gogh$\}$, the keywords will be processed as follows: First, the `ConceptFilter` recognizes "painting" as a concept name and adds the concept "painting" to the concept set $C = \{$painting$\}$. By applying the `ConceptFilter` the keyword "flowers" is identified as a category, which is attached to the property "portrays". Thus, a pair $\langle$portrays, flowers$\rangle$ is appended to the property-category set $PV = \{\langle$portrays, flowers$\rangle\}$. Furthermore, the system adds the concept "fine arts" to the concept set: $C = \{$painting, fine arts$\}$, because the property "portrays" is connected to the concept "fine arts". Subsequently, the `DataValueFilter` finds for keyword "van Gogh" the index entries as shown in Fig. 7 and adds to the property-keyword set following pairs: $PK = \{\langle$title, van Gogh$\rangle, \langle$artist, van Gogh$\rangle\}$. As property "title" is attached to concept "book" the concept set comprises the entries $C = \{$painting, drawing, fine arts, book$\}$. Finally, the system creates from the sets C, PV and PK the CQuery statement Q_7. Only the most specialized concepts are used for query composition. As "painting" is more specialized than "fine arts", concept "fine arts" is removed from the concept set. The result set of the query comprises all paintings by "van Gogh" as well as all books with "van Gogh" as title.

```
Q7 : FOR $c IN concept[name='painting'] UNION
     concept[name='drawing'] UNION concept[name='books']
     LET $e := extension($c), $k := $c/portrays[name='flowers']
     WHERE $e/artist = 'van Gogh' OR $e/portrays = $k OR
       $e/title = 'van Gogh'
     RETURN
       <artist>$e/artist</artist>
       <title>$e/title</title>
```

The keyword-based search is currently under development and is still to be evaluated for query speed and accuracy. Furthermore, we plan to include the keyword-based search into the query processor to allow a domain independent usage.

6 Architecture and Implementation

The YACOB mediator system has been implemented in Java using several available standard technologies. The XML data are processed with the help of JAXP (Java API for XML Processing) including XML parsing and XSLT transformation. Accessing the source systems or the wrapper respectively is implemented using Web Services. For managing and manipulating the RDFS based concept model we use Jena – the Java API for RDF. This package provides an RDF parser, an access interface for RDF data, a mechanism for storing RDF data in a relational database as well as the RDF query language

RDQL. The cache is implemented using the XML database system Xindice, the user interface consists of several JSP pages.

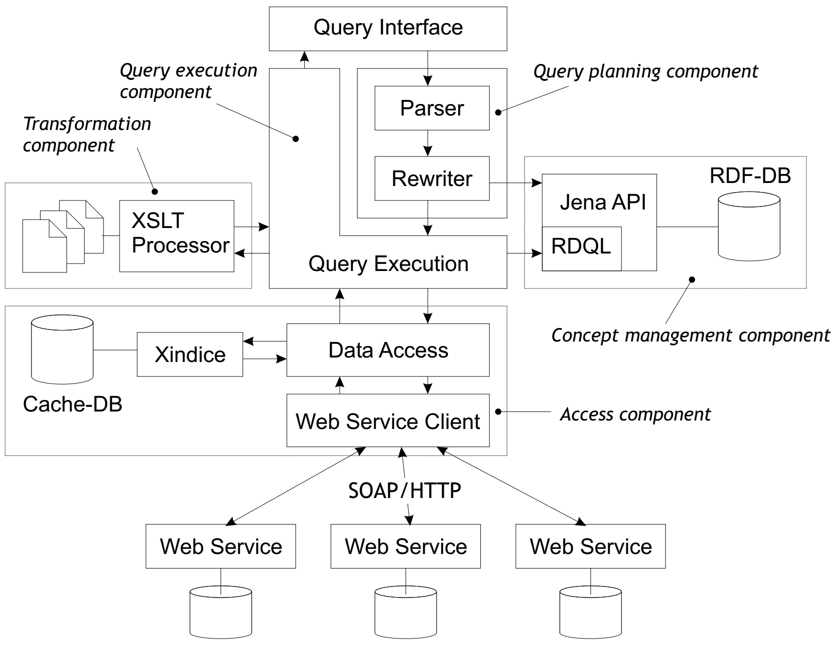


Figure 8: Architecture of the Mediator System

In Fig. 8 the overall architecture is shown. The main components of the system are as follows. The *concept management component*, which is based on the Jena API, manages the concepts, their properties and relationships as well as the information about the mapping to the source systems. Actually, this component provides access to the RDF graph representing the concepts and mappings which is built from a set of RDF files or database tables. The *query planning component* consists of the CQuery parser and the modules for query rewriting, decomposition and translation. It implements the rewriting steps as well the optimization heuristics described in Section 3.2. The *query execution component* implements the query operators and controls the query execution. One of the main tasks of this module is to merge the partial results from the sources by implementing the extensional union operator. In order to benefit from the parallelism of multiple source queries each sub-query below the extensional union operator is executed in a separate thread. The *transformation component* consisting primarily of the XSLT processor provided by JAXP is responsible for transforming the result data retrieved from the sources into a structure according to the global schema. The *data access component* processes XPath queries from the execution component by forwarding them to the source systems via a Web Service or by answering them using the mediator cache.

A special feature of the system is the tight coupling between query planning and execution. Because we are able to identify the relevant sources and to translate the (sub-)queries only after the concept set and the associated mappings are determined, first the **FOR** part of a query has to be evaluated. The appropriate operators are implemented as part of the query execution component and access the RDF data using the Jena API and the RDQL query language. Here, RDQL is used to implement concept-level related plan operators, e.g. selecting concepts, traversing relationships and obtaining mappings for a given concept. However, due to limitations of the current RDQL implementation, some plan operators are implemented by directly accessing and navigating the RDF graph, such as the transitive closure operator. The result of this step – a set of concepts - is returned to the query planner component for determining the mapping information and based on this for decomposing and translating the sub-queries. The final plan in the form of a directed graph of algebra operators is forwarded to the execution engine that invokes the services of the data access component.

XML data that are returned as result of a source query are first transformed using the source-specific XSLT rules into the global schema defined by the concept schema. The transformed data are then stored in the cache and the corresponding mappings are updated. Simultaneously, the result data are further processed by the global query operators according to the plan.

The access to the source systems or eventually required wrappers is implemented using Web Services. This means, each participating source system – or at least the associated wrapper – has to support a simple query service processing an XPath expression and returning an XML document. At the mediator this is implemented using the Java API for XML Messaging (JAXM) that allows to create and process SOAP messages.

7 Related Work

The work related to our approach has been done mainly in the areas of information integration / mediator systems and as part of the efforts towards the Semantic Web.

Over the last years, systems based on the mediator-wrapper-architecture proposed by Wiederhold [Wie92] have been accepted as a viable approach for integrating heterogeneous semistructured data sources in the Web. The most prominent approaches are TSIMMIS and Information Manifold. TSIMMIS [GMPQ$^+$97] is based on the semistructured data model OEM and uses the logic-based language MSL for defining global views and formulating queries. In contrast to this Global-as-View approach, Information Manifold [LRO96] follows a Local-as-View approach by defining the local schemas based on a given global schema structure and content (i.e. the partition of the global relation). MIX [BGL$^+$99] – one of the successors of TSIMMIS – is an XML-based mediator, i.e. it uses XML for representing instance and schema information and supports the query and view definition language XMAS which is based on ideas from XML-QL and MSL.

In all these systems data integration is mainly achieved on a structural level. In contrast, approaches of the semantic or ontology-based integration – such as our YACOB mediator – represent the domain knowledge explicitly using semantic relationships or application-specific constraints and exploit this information for query formulation and processing.

Another example of such a system is KIND [LGM01], where domain knowledge is represented by so-called domain maps. A domain map is specified in the generic conceptual model GCM, that is based on a subset of F-logics. It defines a set of classes (called concepts) as well as their relationships modeled as binary relations. A new source is registered by associating the objects to the concepts of the domain maps. Based on the registered source data one can define integrating views using a rule-based mechanism and following the Global-as-View principle. In contrast to this, the YACOB mediator supports a LAV approach and therefore allows to specify correspondences between the concept level and a sources independently from other sources.

The semantic integration system SIMS [ACHK93] is based on the knowledge representation language Loom. In this language a hierarchical terminological knowledge base representing the domain model is specified. This model is used to describe the contents of the sources independently from each other primary by specifying *is-a* relationships between local and global concepts. In this way, a Local-as-View approach is realized. Compared with SIMS the YACOB mediator does not require an explicit modeling of source data. Instead they are associated with the concepts of the domain model using mapping information. Furthermore, due to the usage of XML technologies we can natively integrate XML/Web sources without wrapping them into knowledge bases.

The Context mediator presented in [GBMS99] also uses a domain model. However, in this case it comprises a set of primitive and semantic types. Instances of semantic types may have different values in different contexts, i.e. there can exist different assumptions in the sources about the interpretation of the values. To each source context axioms can be associated, which specify the conversion of the local values into the global domain model and can be used for query transformation purposes. The category classes of the YACOB concept model are closely related to the semantic types from the Context mediator. However, the axioms proposed there are replaced in YACOB by simple value mappings. Furthermore, the usage of semantic types in the Context mediator addresses mainly the problem of attribute value conflicts and not the problem of overcoming semantic heterogeneity as in our approach.

A system closely related to our approach is the XML mediator STYX [ABFS02], which also follows the LAV principle and uses an ontology as integration model. Here, sources are described by XPath expressions. The query language is similar to OQL with concepts corresponding to classes. The only concept-level operation is the traversal of relationships. In contrast to this, our language supports more powerful operations such as transitive closure and schema-level operations. These operations permit not only to obtain properties of a given concept at runtime, but also to “compute” the set of concepts for which data have to be retrieved from sources.

In recent years there have been several works on semantic caching, particularly in the context of mediator systems. Here, caching is performed at data level by storing the query results together with the queries instead of pages at the physical level [DFJ+96]. This technique is employed for example in data integration scenarios [ASPS96] and in the World Wide Web [LC01]. In our approach we focus on a seamless integration of semantic integration model and caching strategy.

The main concern of the efforts towards the Semantic Web is to improve the automated processing of Web information by adding meta information describing the content, i.e. supporting interoperability on a semantic level. In this context, ontologies representing a source for an exact specification of shared concepts play a key role. Thus, most of the work is dedicated to ontology languages, e.g. DAML+OIL [Hor02]. RDFS as used in our approach can be seen as a basic form of such an ontology language. Particularly for RDF and RDFS several query languages were proposed, which consider special characteristics of RDF descriptions in contrast to simple XML documents. This is addressed for example by operations for accessing schema information, considering type hierarchies and constraints on classes and properties. The most prominent examples of such languages are RQL [KCPA01], Squish, or RDQL from the Jena toolkit that is used in our approach. A detailed survey on these languages is given in [MKA+02]. However, RDF query languages focus only on the meta level and therefore can be only one part of a concept-based query system as described in Section 6. Example usages of ontologies are among others portals [MSS+02] and semantic search engines [DEFS99]. These kinds of systems are based on annotations of the source data, e.g. by embedding meta data in the Web documents. With our approach presented here, we try to employ these mechanisms for the virtual data integration in mediator systems.

8 Conclusion

Representing domain knowledge in terms of concepts and their relationships as semantic meta data for mediators in conjunction with the Local-as-View approach can simplify the integration of new sources as well as the query formulation. Based on this idea we have presented in this paper a query system implementing a concept-based XML query language. Choosing RDF Schema as integration model, we are able to use existing powerful tools for modeling and data exchange, e.g. Protegé as an ontology editor and the Jena API for manipulating and querying RDF data. We have described in detail the process of query rewriting and execution and discussed issues related to caching of data driven by the concept model.

The application domain addressed by our work – integrating Internet databases containing information about cultural assets such as *www.lostart.de* – has turned out as a suitable scenario for semantic integration. We were able not only to model application concepts and relationships in a way simplifying the specification of mappings as well as the query formulation, but the concept model can be exploited to guide the user through the hierarchy of concepts in order to formulate initial queries and refine them. For this purpose a graphical Web interface has been developed addressing such a concept-driven combination of browsing and retrieval. Finally, the system is not restricted to this domain but can be applied to all scenarios where a common set of concepts and relationships representing a semantic "anchor" can be identified.

References

[ABFS02] B. Amann, C. Beeri, I. Fundulaki, and M. Scholl. Ontology-Based Integration of XML Web Resources. In *ISWC 2002*, LNCS 2342, pages 117–131. Springer-Verlag, 2002.

[ACD02] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *ICDE'02*, pages 5–16, 2002.

[ACHK93] Y. Arens, C.Y. Chee, C.-N. Hsu, and C.A. Knoblock. Retrieving and Integrating Data from Multiple Information Sources. *International Journal on Intelligent and Cooperative Information Systems*, 2(2):127–158, 1993.

[ASPS96] S. Adah, K. Selcuk Candan, Y. Papakonstantinou, and V.S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. In *SIGMOD'96*, pages 137–148, 1996.

[BGL$^+$99] C.K. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu. XML-Based Information Mediation with MIX. In *SIGMOD'99*, pages 597–599, 1999.

[BHN$^+$02] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using Banks. In *ICDE'02*, pages 431 – 440, 2002.

[DEFS99] S. Decker, M. Erdmann, D. Fensel, and R. Studer. OntoBroker: Ontology-based Access to Distributed and Semi-Structured Information. In *DS-8: Semantic Issues in Multimedia Systems*. Kluwer, 1999.

[DFJ$^+$96] S. Dar, M.J. Franklin, B.T. Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *VLDB'96*, pages 330–341, 1996.

[FKM00] D. Florescu, D. Kossmann, and I. Manolescu. Integrating Keyword Search into XML Query Processing. In *9th Int. World Wide Web Conference*, Computer Networks, June 2000.

[GBMS99] C.H. Goh, S. Bressan, S.E. Madnick, and M.D. Siegel. Context Interchange: New Features and Formalisms for the Intelligent Integration of Information. *ACM Transactions on Information Systems*, 17(3):270–293, 1999.

[GMPQ$^+$97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J.D. Ullman, V. Vassalos, and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[Hor02] I. Horrocks. DAML+OIL: a Description Logic for the Semantic Web. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 25(1):4–9, 2002.

[HP02] V. Hristidis and Y. Papakonstantinou. Discover: Keyword Search in Relational Databases. In *VLDB'02*, pages 670–681, 2002.

[KCPA01] G. Karvounarakis, V. Christophides, D. Plexousakis, and S. Alexaki. Querying RDF Descriptions for Community Web Portals. In *The French National Database Conference BDA'2001*, 2001.

[LC01] D. Lee and W.W. Chu. Towards Intelligent Semantic Caching for Web Sources. *Journal of Intelligent Information Systems*, 17(1):23–45, 2001.

[LGM01] B. Ludäscher, A. Gupta, and M.E. Martone. Model-based Mediation with Domain Maps. In *ICDE'01*, pages 82–90, 2001.

[LRO96] A.Y. Levy, A. Rajaraman, and J.J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *VLDB'96*, pages 251–262, 1996.

[MKA$^+$02] A. Magkanaraki, G. Karvounarakis, Ta Tuan Anh, V. Christophides, and D. Plexousakis. Ontology Storage and Querying. Technical Report 308, Foundation for Research and Technology Hellas, Institute of Computer Science, April 2002.

[MRYGM01] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a Distributed Full-text Index for the Web. *TOIS*, 19(3):217–241, July 2001.

[MSS$^+$02] A. Maedche, S. Staab, R. Studer, Y. Sure, and R. Volz. SEAL – Tying Up Information Integration and Web Site Management by Ontologies. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 25(1):10–17, 2002.

[RS97] M.T. Roth and P.M. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *VLDB'97*, pages 266–275, 1997.

[Wie92] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38–49, 1992.