# Query Reformulation for Keyword Searching in Mediator Systems

Ingolf Geist Torsten Declercq Kai-Uwe Sattler Eike Schallehn

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

## Abstract

Integration of heterogeneous data sources is still an important task. Mediator systems are one approach to support a structured search over heterogeneous sources. These systems provide comprehensive query languages, which are very powerful but hard to use for inexperienced users. Therefore, easier query interfaces have to be developed. One well-known and effective interface is the keyword search. However, one has to consider the capabilities of sources, which mostly support only structured queries.

The aim of this paper is the development of a keyword query component based on a concept-based mediator to overcome this problem. The efficient keyword query execution is supported by an index on global level as well as a concept model, consisting of concepts and their properties and relationships. The search is performed in a two-step process that comprises (i) index lookup to select relevant concepts and properties and (ii) query reformulation for an efficient query processing by the mediator system. The system is evaluated by means of an application scenario that includes different data sources from the area of cultural assets.

# 1 Introduction

Providing an integrated access to heterogeneous data sources from the Web is still a big challenge. In this context, 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 (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.

Search engines such as Google or Inktomi have the advantage of a very easily usable query interface. The user has to provide only a few keywords. However, a set of documents is returned as the result containing the keywords at any position. In contrast, mediator systems allow to formulate structured queries addressing individual classes of objects and their properties and in this way to access structured sources such as relational, object-oriented or XML databases. But using this approach it is more difficult for inexperienced users to formulate a query because knowledge about the global schema as well as the query language is required.

In this paper, we present the YACOB mediator – a semantic integration system that uses domain knowledge in the form of concepts and their relationships for formulating and processing queries. From this semantic layer a mapping is defined to the source data, i.e., it is specified 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. 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 mediator implements the concept-based query language CQuery supporting queries both on concept as well as instance level. Both the RDF-based integration model as well as the query language are described in detail in [SGS03, SGHS03]. In this paper, we focus on a special component of the mediator system: the keyword search component. The purpose of this component is to support keyword-based queries on virtually integrated data. This requires to transform a query consisting of a set of keywords into a structured CQuery query which is decomposed by the mediator into a set of source queries. In order to restrict the number of expensive source queries our query transformation approach exploits an index representing the "occurrence" of keywords in the semantic layer, i.e., if it appears as the name of a concept or a property, or in the data layer, i.e., if the keyword corresponds to a property value of an object.

The remainder of this paper is organized as follows. In Section 2 we give a brief overview on the integration model as well as the query language. Based on the integration and the query language we develop the keyword query processing in Section 3 which comprises a keyword index as well as a CQuery generator component. In Section 4 we describe the implementation and evaluate the keyword search on an example domain. After a discussion of related work in Section 5 we conclude the paper and point out to future work in Section 6.

## 2 Integration Model and Query Language

In order to represent the data from heterogeneous sources as well as their semantics we use a two-level integration model. The data or instance level consists of the data managed by the sources and is represented in XML. Thus, data objects "exported" by a source via a wrapper and processed inside the mediator are XML elements of any (source-specific) DTD. For the sources we assume that they are able to answer simple XPath queries. The necessary transformations from XPath to the local query language or call interface are performed by wrappers.

At the second level – the meta or concept level – the semantics of the data and their relationships are described. For this purpose we use RDF Schema. RDF is a simple graph-based model, where nodes model resources or literals and edges represent properties. RDF Schema (RDFS) extends this by defining primitives for specifying vocabularies such as classes or class relationships. 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.

A third part of the integration model is the specification of the mapping to the local schemas of the sources, i.e. 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 follow the Local-as-View approach where a source schema is defined as a view on the global (concept) schema. For each source, mappings of the concepts and properties have to be specified. A concept mapping describes, 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. A concept mapping consists of the source name, the local element name, and optionally a filtering predicate. 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 form of an XPath expression allows to further restrict the instance set.

A property mapping 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. Using these several mappings the elements of the concept schema are

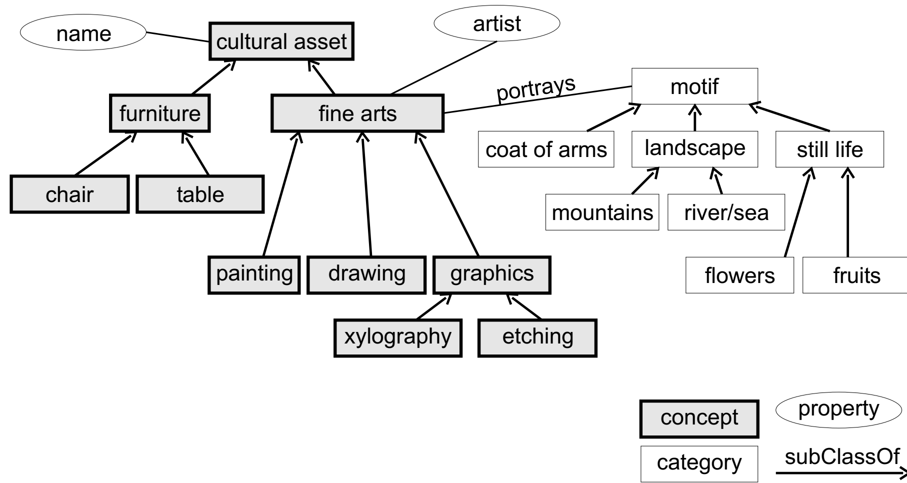



Figure 1: Concept Hierarchy

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.

For formulating queries using this model the query language CQuery is provided, which is derived form XQuery. 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. 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 el-

ements, too. A special meaning has the "*" operator that returns the transitive closure of the sub-concept relationship. 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 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. Beside standard operators such as =, <>, <, > etc. the **WHERE** clause may contain predicates using the text similarity operator $\sim$=.

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 and returns all instances of the concept "painting" or its sub-concepts for which any property contains the keyword "flower".

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.

During query processing a global CQuery is translated into an internal algebra representation. Next, the concept-level part of the query (the **FOR** clause) is evaluated. Based on the result – a set of concepts – and the associated mappings the relevant sources are identified. Using the concept and property mappings the query is decomposed into sub-queries and translated according the local schema into an XPath query, which is sent to the source. The results of the source queries are transformed into the global schema (again using the mappings) and remaining global operators such as join, union etc. are applied. The whole process including optimization steps not mentioned here is described in [SGS03, SGHS03].

# 3 Keyword Search and Concepts

The previous section described the features of the YACOB system and its query language CQuery, which is a powerful but very complex query language. Therefore, it is necessary to develop an easily usable but efficient user interface. A keyword search is easy to use and does not require knowledge about the structure of the meta level and its data. Because the integrated sources provide mostly only a structured query interface instead of a keyword search, it is not possible to send the keywords to each source and execute the keyword queries in the sources. Furthermore, the sources are only accessible through a structured query interface of the mediator.

We can express our problem informally as following: For a given keyword set a keyword query returns all relevant integrated objects through the mediator. An object is an instance of a concept that is integrated by the mediator. It is represented by an XML fragment. An object is relevant to a keyword set, if it contains all keywords in its attribute values or the object belongs to a concept that is determined by the keywords, respectively. The keyword search comprises the meta level as well as the instance level. For example, the keyword query "Paintings Gogh" selects all objects containing "Paintings" and "Gogh" in their properties and all objects belonging to the concept "Paintings" and containing "Gogh" in one of their properties.

A generic implementation is the search over all concepts and properties for a given keyword, which is supported by the meta level query capabilities of CQuery. This kind of queries is possible but very inefficient because it scans all concepts for every source for the keywords. The following query supports the keyword search for "Gogh" in the data level:

$Q_4$ :
```
FOR $c IN concept[name='Cultural As-
set']/*
LET $e := extension($c)
WHERE $e/properties ~= 'van Gogh'
```

Because of the bad runtime characteristics of this kind of queries, we need a technique to create more efficient queries. Efficient query means, that only necessary concepts and properties should be accessed. In our approach we solve this problem by using a *keyword index* that relates a keyword to meta-data part, e.g. concept, category and/or property. The integrated objects cannot indexed directly, because they do not have a globally unique identifier.

The search space of the mediator can be described conceptually by a *virtual document* that has to be indexed. This document comprises the conceptual level – concepts, categories and properties – as well as the data level, i.e. the objects. Please note, that the document is not materialized on the global level, but serves as the motivation of the chosen index structure. Fig. 2 illustrates the tree structure of an document fragment, that comprises information about the concepts `Paintings` and `Drawings` and their objects.

The meta level objects are modeled by the tags `concept`, `category` and `property`, respectively. Each meta level object has a unique identifier and includes a name tag, which is indexed in the system. Furthermore, the properties are included in their corresponding concept tags, and the categories contain links to the corresponding concepts and properties. The concept hierarchies are maintained in this structure.

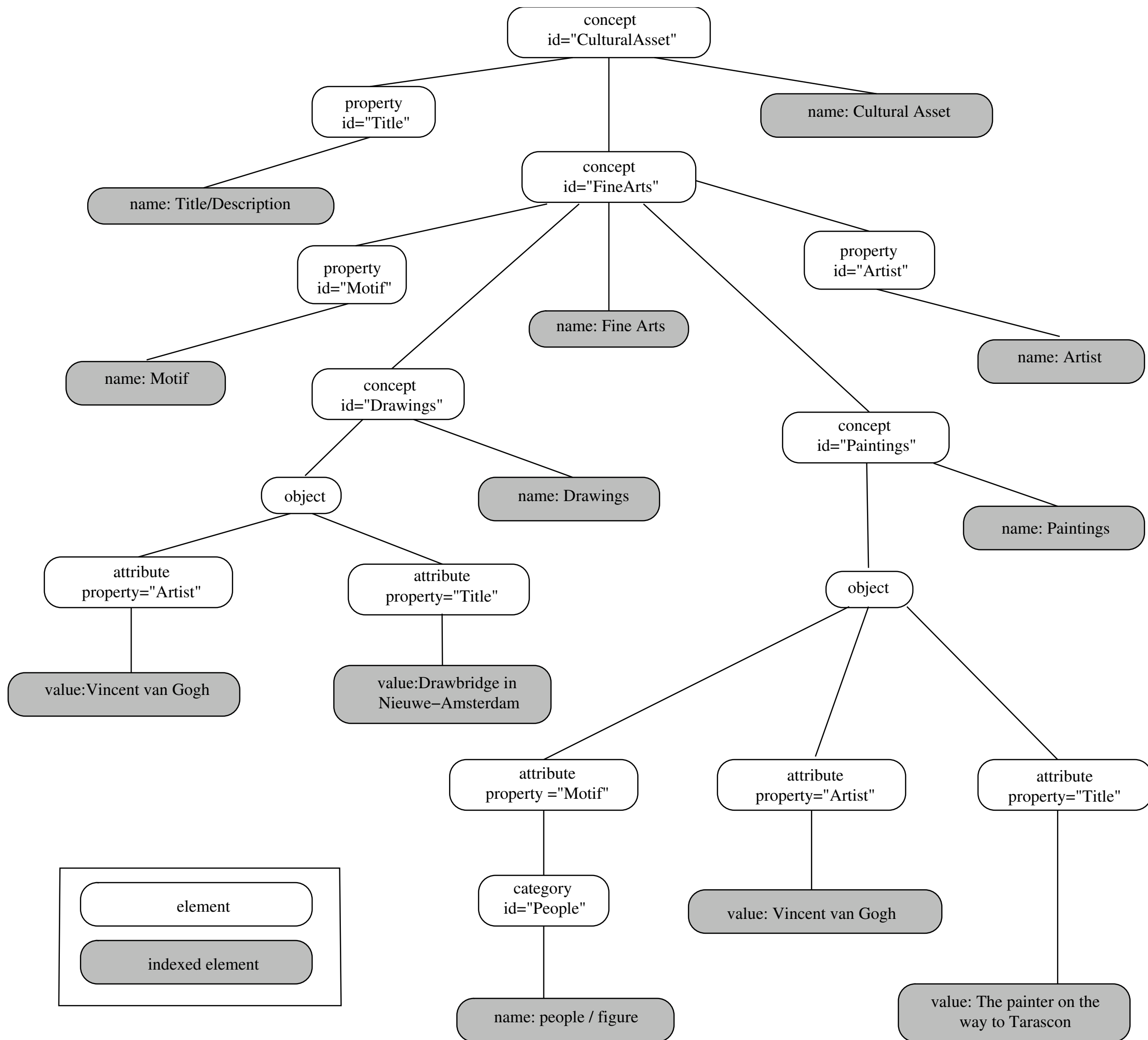



Figure 2: Example document fragment

The data level is modeled by object tags, which are included in the corresponding concept tags. The objects are structured by sub tags, named attribute, that correspond to the properties. The values of properties are included in the attribute tags and are indexed. Special attribute values correspond to categories and contain a reference to the category information. Except of that, they are handled and indexed like ordinary attribute values.

The objective of the keyword search is the retrieval of the relevant objects from the virtual document. As the objects are virtual, we need to create CQuery statements to fetch them from the sources. The position of a keyword determines its corresponding concepts and properties. Each keyword can be classified into four roles depending on their position (determined by a path expression) in the virtual document:

**Concept Keyword ($r_c$):** A concept keyword is contained in a concept name tag and corre-

sponds to a concept. It represents all objects associated to this concept and its subconcepts.

**Category Keyword ($r_k$):** A category keyword is contained in category name tag and corresponds to a category. Its position is specified by a concept, a property and the category. A category keyword represents all objects, that contain this category value in the specified property.

**Property Keyword ($r_p$):** A property keyword corresponds to a property which is assigned to a concept. A concept and the property specify the position. The keyword represents all objects that possess this property and contain a value for the property.

**Value Keyword ($r_v$):** A value keyword is found in a value of an attribute of an object. The position is specified by a concept and property associated to the object and attribute. It represents all objects that belong to the corresponding concept and contain this keyword in the specified property.

One keyword can be contained in different elements and therefore it can have different roles assigned. For example, consider the keyword `Paintings` which represents the concept *Paintings*, but is also contained in the book title "Paintings by Vincent van Gogh". Therefore, it has the roles $r_c$ and $r_v$.

Based on this discussion, the keyword query processing comprises three steps:

1. Retrieval of relevant index entries,
2. Query reformulation and
3. Execution of the CQuery statement by the mediator and presentation of the result to the user.

In the first step all relevant index entries are selected from the keyword index based on the given keywords and the concept hierarchy. In the second step, a CQuery statement is generated with help of these information. The query comprises only the relevant concepts and the relevant parts of their extension. In the third step the query expression is transferred to the query execution components of the mediator and is executed. Afterwards the result is presented to the user. The third step has already been presented in [SGS03, SGHS03], therefore only the index (Sec. 3.1) and the query reformulation (Sec. 3.2) are presented here.

## 3.1 Keyword Index

### 3.1.1 Structure

The structure of the keyword index follows directly from the structure of the virtual document. We cannot index the objects directly, because they are virtual and it does not exist a unique and persistent global identifier for them. But we can index the meta level, i.e. concepts, categories and the properties. Furthermore, we can index the value keywords and store the path in form

of concept and property as well as possibly the category, in order to create CQuery statements to fetch the associated objects. The index is built from data of the different sources. A crawler uses the mediator components to select objects based concepts and sources. In the following, we assume a complete index, which is up to date.

A keyword index entry is defined as a 5-tuple: $\langle kw, role, c, p, k\rangle$ with $kw$ is a keyword, $c$ a concept, $p$ a property, $k$ a category and $role$ the role of keyword. The structure of the index follows the idea of the inverted list [Sal89, BYRN99], where a list of documents is assigned to a keyword, which is contained in these documents. In our case, a list of paths, is assigned to the keyword. So, we can determine the position of the keyword in the concept hierarchy and its role.

Tab. 1 illustrates the positions of the different keyword roles by giving XPath expressions in order to find the indexed elements.

| **Keyword Role** | **Position in document (XPath)** |
|---|---|
| $r_c$ | //concept[@id="$c$"]/name |
| $r_p$ | //concept[@id="$c$" or .//concept/@id="$c$"]/property[@id="$p$"]/name |
| $r_k$ | //concept[@id="$c$"]/object/property[@id="$p$"] //category[@id="$k$"]/name |
| $r_v$ | //concept[@id="$c$"]/object/attribute[@property="$p$"]/value |

Table 1: Index entries and virtual document

The index entries represent XPath expressions to retrieve the referenced set of objects from the virtual set. In the actual implementation the paths are transformed to CQuery expressions to get the objects from the sources. For example, consider the entry for a value keyword "tarascon": $\langle tarascon, r_v, paintings, title, \perp\rangle$. The corresponding queries in CQuery are the following:

$Q_5$:
```
FOR $c IN concept[name="paintings"]
LET $e := extension($c)
WHERE $e/title ~= "tarascon"
```

Tab. 2 gives an overview about different query representations of index entries. Note, that a property keyword represents all objects that contain a attribute corresponding to referenced property.

### 3.1.2 Implementation and Entry Retrieval

The keyword index is implemented using a relational database system. A database table named `concept_index` represents the index. The table is structured as described in Tab. 3. There are different ways to partition a keyword index table for XML documents [FKM00]. We use the simple solution because in first tests the retrieval time of the source objects was more important than the index access.

Additionally to the already described attributes the columns `source`, `path` and `weight` were added to the index. The attribute `source` stores the source where objects with this keyword were found. `path` holds the path to the concept of the keyword, which extracted

| Index entry | CQuery expression |
|---|---|
| $\langle kw, r_c, c, \perp, \perp\rangle$ | **FOR** $c **IN** concept[name="$c$"]<br>**LET** $e := extension($c) |
| $\langle kw, r_p, c, p, \perp\rangle$ | **FOR** $c **IN** concept[name="$c$"]<br>**LET** $e := extension($c)<br>**WHERE exists($e/$p$)** |
| $\langle kw, r_k, c, p, k\rangle$ | **FOR** $c **IN** concept[name="$c$"]<br>**LET** $e := extension($c), $k := $c/$kp$[name="$k$"]<br>**WHERE** $e/$kp$ = $k |
| $\langle kw, r_v, c, p, \perp\rangle$ | **FOR** $c **IN** concept[name="$c$"]<br>**LET** $e := extension($c)<br>**WHERE** $e/$p$ = "$kw$" |

Table 2: Corresponding queries for an index entry

| field name | description |
|---|---|
| keyword | the keyword as string |
| source | the source, from which the objects are originated |
| role | role of the keyword: $r_c$, $r_k$, $r_p$, $r_v$ |
| path | path to the corresponding concept on meta level |
| concept | represented concept |
| property | represented property |
| category | the represented category |
| weight | the number of represented objects |

Table 3: Index structure

from the concept model. The `weight` column maintains the number of objects that contain the keyword. This value can be used to establish a ranking of results. An index is created on the keyword column to support a faster access to the entries.

Relevant entries to a given keyword are retrieved by a SQL query of the following form:

```
SELECT * FROM concept_index
WHERE keyword = 'tarascon'
AND source in {'lostart','herkomst'}
AND path like '%/paintings/%'
```

The requested keyword is "tarascon". The number of returned index entries can be restricted to certain sources by using the `source` attribute or to certain concepts by using the `path` attribute. The `path` attribute represents the path from the root concept to the indexed concept. The condition `path like '%/paintings/%'` selects all entries for the concept "painting" and its sub-concepts.

## 3.2 Query reformulation

The input of a keyword query is a set of keywords $\mathcal{KW} = \{kw_1, \ldots, kw_2\}$. The goal is to find a set of objects that are relevant to the keyword set. The objects are fetched by a CQuery, which is created during the query processing. The query reformulation described here, finds this CQuery $Q$ from the set of keywords. The processing utilizes the keyword index $Idx$ and the concept hierarchy $\mathcal{C}$. The complete algorithm is shown in Fig. 3.

The first step of the query reformulation is the selection of relevant index entries (see Fig. 3 line 1-11). The selection comprises three steps:

(i) select all index entries that support a keyword from $\mathcal{KW}$,

(ii) add for each concept entry concept entries of the sub-concepts and

(iii) select only entries that correspond to a concept that is relevant to all keywords of $\mathcal{KW}$.

By performing step (ii) we select also objects of sub-concepts, e.g. a keyword "Fine Arts" correspond to concept "Fine Arts" as well as to the sub-concepts "Drawings" and "Paintings". The idea behind step (iii) is that only concepts that support all keywords can contain objects that are relevant.

Based on the selected index entries the relevant concepts are stored in the set $Concepts$ (line 12-14) and the **WHERE** condition is computed (line 15-22). The **WHERE** is computed for each concept separately. We assume that the conditions are in disjunctive normal form ($\mathcal{DNF}$). A simple conjunction (union) of the conditions for all selected index entries would be too restrictive. Consider, the following example: the conditions {artist $\sim=$ "Gogh"} and {title $\sim=$ "Gogh"}.[1] The conjunction of the conditions is too restrictive, because an object is relevant if it contains a keyword in at least one property. Thus, the right condition is the disjunction of them. To avoid this problem, we group the conditions by keyword and compute the Cartesian product of these sets. The disjunction (union) of all conditions for each concept models the **WHERE** clause of the CQuery query.

The last step in keyword query reformulation (line 23-24) is the generation of the query string from the set of selected concepts and the conditions set. The relevant concepts are selected in the **FOR** clause and the extensions of the concepts are restricted by the **WHERE** clause. Fig. 4 illustrates all steps using an example. Given the keyword set `gogh`, `tarascon`, `painting`, the relevant index entries are retrieved and subsequently transformed into concepts and conditions. Finally, the corresponding CQuery is generated, that fetches the relevant objects. We added a type constraint to distinguish between instances of different concepts.

## 3.3 Discussion

In this section we discuss additional issues that have to be solved in the keyword search system and how we support the solution with our query system.

[1] In the case we handle category keywords similar to this approach with the difference, that an entry $e$ represents the condition $e.p = e.cat$. Entries of the roles concept and property are not relevant for construction of the **WHERE** condition.

**Input**:
keyword set $\mathcal{KW}$
set keyword index entries $Idx$
concept hierarchy $\mathcal{C}$
with function subconcepts($C$) returns all sub-concepts $\mathcal{C}_C \subset \mathcal{C}$ of concept $C \in \mathcal{C}$
set of concepts $Concepts = \emptyset$
set of conditions in $\mathcal{DNF}$ $Cond = \emptyset$
**Output**:
Query String $Q$

1 */* select relevant index entries */*
2 $\mathcal{E}_{\mathcal{KW}} := \{e : e \in Idx, e.kw \in \mathcal{KW}\}$
3 */* add subconcept for concept keywords */*
4 **forall** $e \in \mathcal{E}_{\mathcal{KW}} \wedge e.role = r_c$ **do**
5 $C_e :=$ subconcepts$(e.concept)$
6 **forall** $c \in C_e$ **do**
7 $\mathcal{E}_{\mathcal{KW}} \cup \{\langle e.kw, r_c, c, \perp\rangle\}$
8 **od**
9 **od**
10 */* select all index entries that belong to concepts that support all keywords*/*
11 $\mathcal{E}'_{\mathcal{KW}} := \{e : e \in \mathcal{E}_{\mathcal{KW}}, \forall k \in \mathcal{KW} \setminus \{e.kw\} \exists e' \in \mathcal{E}_{\mathcal{KW}} : e'.kw = k \wedge e.concept = e'.concept\}$
12 **forall** $e \in \mathcal{E}'_{\mathcal{KW}}$
13 $Concepts = Concepts \cup e.c$
14 **od**
15 */* group by concept */*
16 $\mathcal{E} = \{E : E \subseteq \mathcal{E}'_{\mathcal{KW}}, \forall e, e' \in E : e.concept = e'.concept\}$
17 **forall** $E \in \mathcal{E}$ **do**
18 */* group by keyword */*
19 $E_{\mathcal{KW}} = \{E_{kw} : E_{kw} \subseteq E, \forall e, e' \in E_{kw} : e.kw = e'.kw\}$
20 */* Cartesian product */*
21 $Cond = Cond \cup (E_{kw_1} \times E_{kw_2} \times \ldots \times E_{kw_n}), \forall E_{kw_i} \in E_{\mathcal{KW}}$
22 **od**
23 */* generate query */*
24 $Q =$ generateQuery$(Concepts, Conditions)$

Figure 3: Query Reformulation

### 3.3.1 User Guiding and Ranking

An effective search interface guides the user through the complete search process, that comprises several iterations. In our system we can utilize the index information as well as the concept hierarchy to make proposals to the user if the search was not successful. We distin-

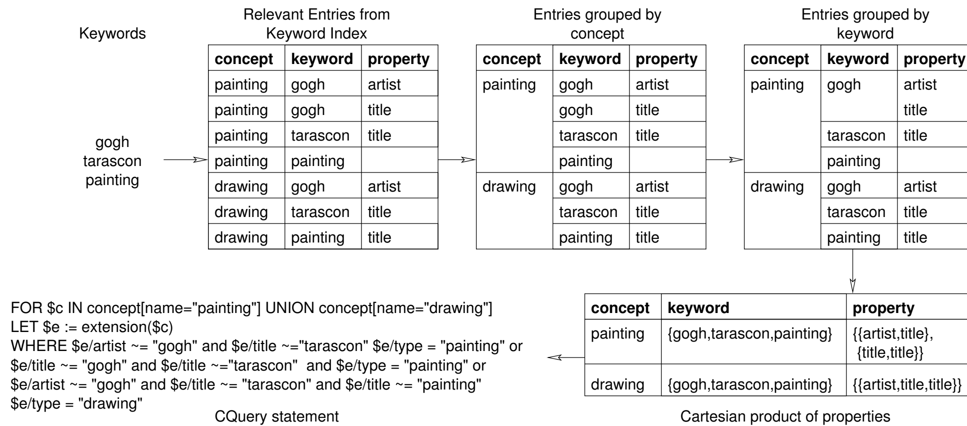



Figure 4: Example Keyword query processing

guish two cases:

**Too many results.** In the case of too many results, the objects are grouped by their concept association and all concepts plus corresponding example objects are presented. The concepts are ranked by the number of concepts. If category values are assigned to a concept, the user can select a category value. By clicking on a concept or category the user can add keywords from the meta level to restrict her query result. This approach combines a browsing and a keyword search interface.

**Wrong or no results.** Empty or wrong result sets can have different reasons. First, a keyword can be misspelled, thus, the user interface should include a spell checker. Second, the query is over-specified and for the keyword set exists no results. This over-specification occurs by giving too many keywords or by selecting a too specialized concept. The system can help by providing information about less specified or similar concepts. Assume the query "paintings dürer", but no paintings of A. Dürer are included in the databases. So, the system should propose other concepts instead of paintings based on the concept hierarchy, e.g. the super concept and/or the siblings, i.e Fine Arts, Drawings etc.

### 3.3.2 Index Building and Maintenance

Another issue is the building and maintenance of the keyword index. In our current implementation, we assume a complete index which is built by crawling all participating sources and querying the extensions of all supported concepts. These query results are processed and used for creating the index. This means, the XML fragments representing source objects are searched for keywords by extracting textual data, eliminating stop words and apply stemming. Finally, the resulting keywords are stored in the index together with the concepts and properties where they appear.

In this context, two further issues arise. First, we have to consider changes of the sources, i.e., from time to time the keyword index has to be updated in order to reflect the current content of the sources. Second, to have complete information about the content of the sources is often not a realistic assumption. Instead we have to deal with cases where we know only portions of the whole extensions and in this way can keep only a partial index. The impact of a partial keyword index is discussed in Section 4.2. Hence, we consider in the following only the index maintenance strategy.

One approach for dealing with partial indexes and in this way to reduce the effort for building and maintaining the keyword index is currently under evaluation. The main idea is to combine caching and indexing. After a bootstrap crawling phase where a crawler collects data only for a set of user-given queries and builds an initial index, we can use two sources for feeding and focusing the crawler:

- By monitoring CQuery queries processed by the query engine we can identify frequently used concepts and properties as well as the responding sources. This is similar to the idea of a focused crawler [CvD99, STS$^+$03].
- We can use query results as input for the keyword extraction step.

Both strategies ensure that the keyword index contains keywords which appear most frequently in queries and/or results and in this way help to derive queries for performing keyword-based searches more efficient and return relevant source objects. However, because we cannot guarantee a complete result the user interface should advise the user to this and offer an opportunity to retrieve the missing results by performing a generic query or using the browsing interface.

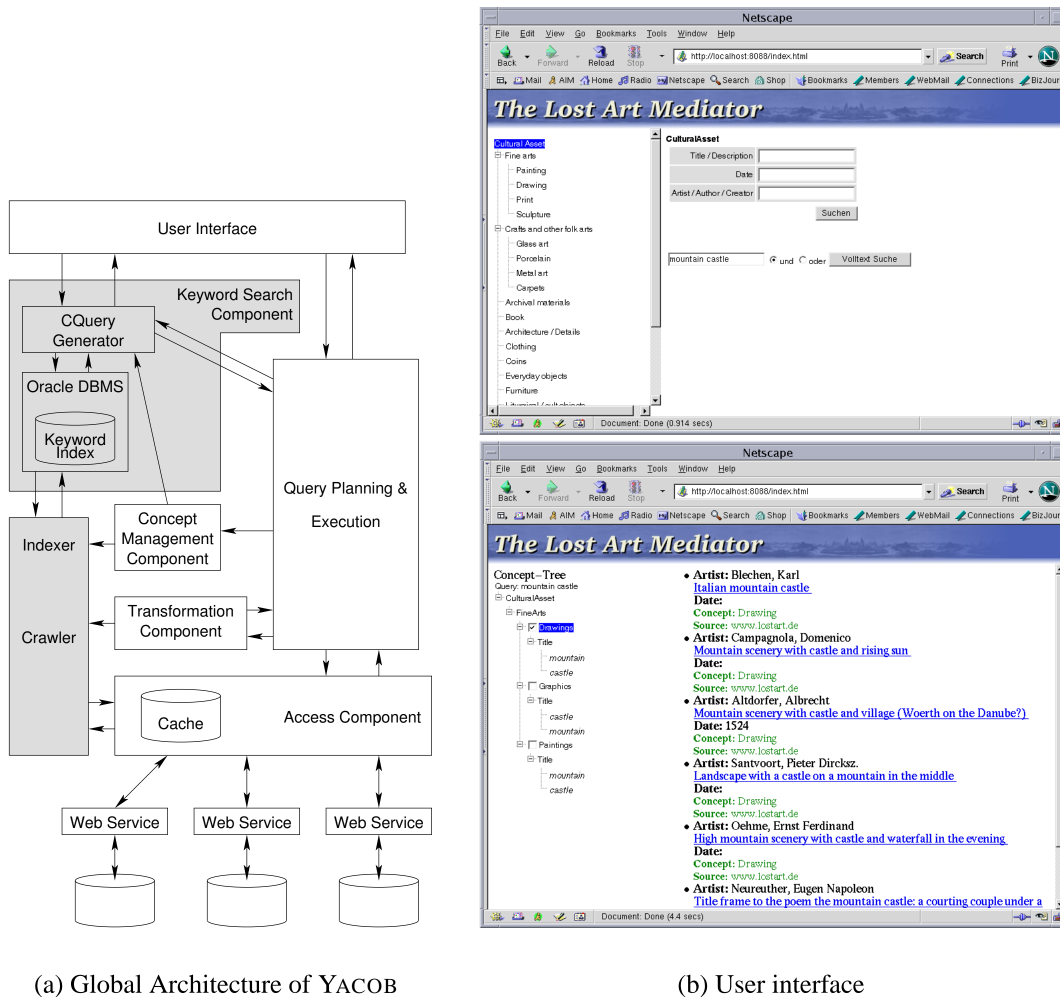
# 4 Implementation and Evaluation

## 4.1 Implementation

The YACOB mediator system is implemented in Java and the Oracle DBMS. The overall architecture is depicted in Fig. 5(a). The focus of the implementation explanation lies on the keyword search component. The other components are necessary to execute CQuery statements and are described in detail in [SGS03, SGHS03].

The *keyword index* is implemented using Oracle DBMS. The Oracle Text Cartridge is used for keyword preparation (stemming, stop word exclusion etc.). The index is stored in one relation that contains all entry information. The *CQuery generator* was implemented in Java and uses the keyword index as well as concept management component to generate CQuery statements, which are executed by the mediator. For index building and maintenance a *crawler* was implemented as background process. The crawler is controlled by the statistics from the index. For data access it generates XPath expressions for the sources by utilizing the concept management component, the transformation component and the access component. The *user interface* contains several HTML pages that are generated by Java Server Pages. Screenshots of the interface are given in Fig. 5(b). The upper picture describes the combined browsing and

keyword input interface. The lower shows the result objects as well as the selected concepts and properties.




(a) Global Architecture of YACOB

(b) User interface

Figure 5: Architecture and User Interface of the Mediator System

## 4.2 Experiments

Given the approach and implementation described in previous sections, the key questions regarding the usefulness of the approach are as follows.

**Query complexity:** mapping keyword queries to structured queries without any further knowledge about the domain would require large disjunctive queries for all sources, concepts and properties. Using the proposed index the query is transformed to a representation where only relevant concepts and properties and sources supporting these concepts are

queried. This way transfer costs and the load of the source systems can be reduced in terms of the number of source queries.

**Accuracy of the query results:** if the index covers all data in all sources the query transformation produces correct queries as shown before. On the other hand, building a complete index is difficult, as discussed in previous sections. We will show that for partially complete indexes the approach yields reasonable results.

All the experiments described below were done on a real-life database containing detailed information about 40.000 cultural assets that were lost during World War II. The assets belong to 22 concepts including paintings, furniture, books, etc. For evaluation purposes different indexes covering a sampled ratio of objects ranging from 10% to 100% were created.

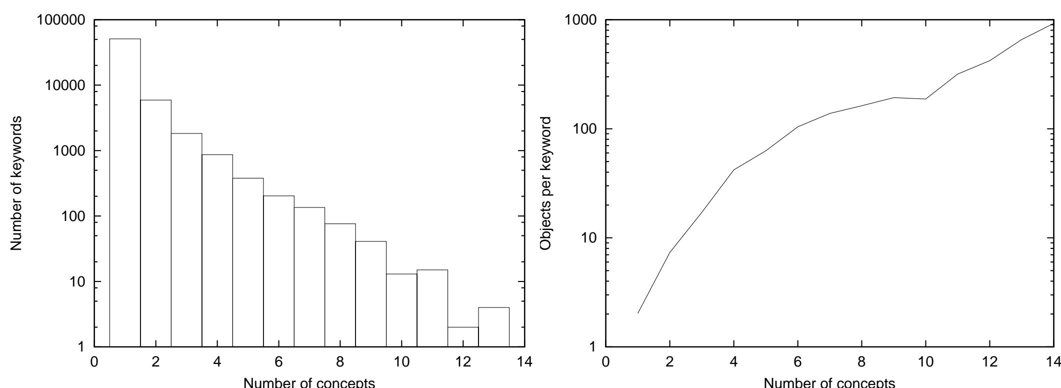



Figure 6: Keyword distribution characteristics of the data set

As shown in the first Fig. 6, the relationship between keywords and related concepts in the used data set is very close. That means, the majority of keywords relate to only one concept, and only a small percentage of the indexed keywords relate to more than three concepts. Hence, for keywords in the index very small and precise queries can be derived. On the other hand, the second Fig. 6 shows that the selectivity of the generated queries is very high, as for these majority of queries involving only few concepts the returned result sets are very small. In summary, the approach guarantees small queries and small result sets for indexed keywords.

Above we described the resulting query complexity for keywords found in the index, but did not yet discuss the rate of reduction of the number of source queries necessary to process a keyword search in the mediator. Due to the kind of concept mapping – a source extension is associated with a concept – we can simply determine the number $N_Q$ of source queries for a generic query like $Q_6$ by $N_Q = \#sources * \#concepts$. The reason is that our query decomposition strategy generates a source query for each concept, if we do not consider cases where redundant queries can be eliminated for sub-concepts relationships as described in [SGS03].

Based on the real-world dataset described above we generated a set of sample queries containing keywords from the index and computed the number of source queries (for a single source and 19 concepts) using the reformulation approach. As shown in Fig. 7 the number

of queries is dramatically reduced compared to the generic approach. The figure describes a conjunctive keyword query.

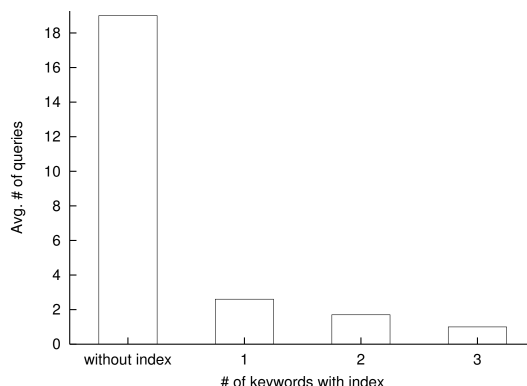



Figure 7: Query reduction

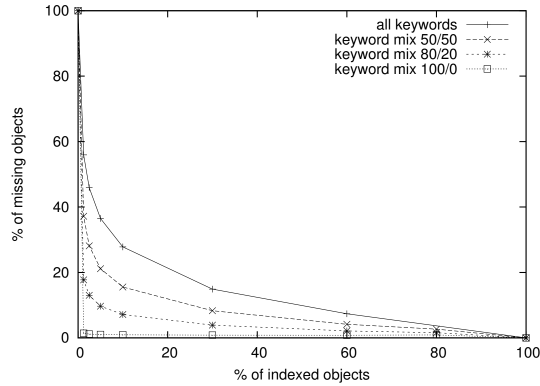
In the second experiment we investigated the impact of partial keyword indexes. For this purpose we built indexes for different portions of the whole dataset ($10\%$, $20\%$, etc.) and ran several sets of sample queries. These query mixes contained queries using keywords from the index and keywords not contained in the index in different ratios (Fig. 8(a)). The y axis of the diagram in Fig. 8(a) shows the fraction of missing result objects, i.e. objects not contained in the expected result set of the queries. This portion of the result has to be obtained by a "difference" query but is not considered here. Interpreting the results from Fig. 8(a) we can state that keyword-based search can benefit even from partial indexes. As shown in our experiments already an index covering $20 - 30\%$ of all objects can cope $80\%$ of the potential result objects. Combining this with a caching approach as described in Section 3.3.2 we can expect to efficiently process most of keyword-based queries by reformulating them into structured mediator queries.

Another point of interest was the size of the generated index. Fig. 8(b) shows the index growing slightly better than linear to its full size of 4 MByte. Considering the full database size of 37 MByte these results are certainly reasonable, especially in virtual integration scenarios.

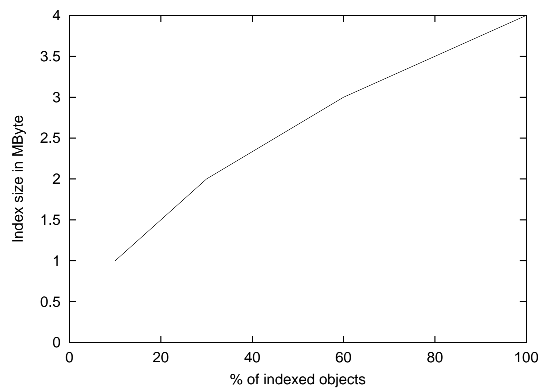
# 5 Related Work

The work related to our approach has been done mainly in the areas of information integration / mediator systems as well as of keyword-based search systems.

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 [GMPQ+97] and Information Manifold [LRO96]. Newer approaches try to overcome heterogeneities of the source data by representing domain knowledge explicitly using semantic




(a) Accuracy of the query results




(b) Size of the index for growing indexing rate

Figure 8: Accuracy and Index Size

relationships or application-specific constraints and exploiting this information for query formulation and processing.

Beside our YACOB mediator described in this paper there are several examples of such systems. KIND [LGM01] uses so-called domain maps for representing domain knowledge, which are specified in the generic conceptual model GCM, that is based on a subset of F-logics. 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. Another 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. All these systems are more or less “traditional” mediator systems implementing a structured query language. Thus, beside simple “like”-predicates no keyword search is supported.

The keyword search is well studied in document bases [vR79, Sal89, BYRN99] and Internet search engines [ACGM$^+$01]. The inverted list index structure is a common way to implement the keyword search over documents as well as structured data sources [vR79, Sal89, BYRN99, FKM00, MRYGM01, ACD02].

Meta search engines [SE97, LG98] are similar to our approach. But, whereas meta search engines use keyword search interface from different search engines and combines the results, our approach implements a keyword search over several structured sources, that offer only a structured query interface. Therefore, our approach is more related to keyword search systems over structured data, as relational databases [DEGP98, MV00, WWS$^+$00, ACD02, BHN$^+$02, HP02] or XML [FKM00].

The DBXplorer [ACD02] approach uses also an inverted list to relate keywords and columns. But in this system only keywords on the data level, like the system DISCOVER [HP02], are considered. In contrast we consider also the meta level. The system BANKS [BHN$^+$02] also includes the meta information, i.e. the database schema, in the keyword search. The system uses a data graph in which the relevant sub trees are searched for. Because in our approach the objects are not materialized, we cannot use an explicit data graph. The master index of DISCOVER targets to actual tuples which is not applicable in our system.

In all database search systems tuple joins via using meta information are included. This feature is not developed yet in our approach. But it can be added using the meta model, that uses other properties than `subclassOf` and compute the joins based on paths over these properties.

In [FKM00] the authors describe the integration of keyword queries in an XML query language. Their index also describes different types of keywords and their position in an XML document. In contrast our approach indexes a *virtual* XML document, that can be loaded by execution of CQuery statements. Furthermore, our keyword query was implemented on top of the query engine. However, all these approaches deal with the keyword search in central databases and XML documents, respectively. In contrast, our approach queries several heterogenous databases using an index scheme and a mediator.

The XXL language [TW02] is another approach to include keyword search in XML. XXL extends a subset of XMLQL by a similarity operator and allows a similarity search on XML documents. Ontologies are used to focus the search. Special indexes for element paths, element values and ontologies allow the efficient retrieval of results. The output of XXL is ranked, in contrast we support only a boolean retrieval, which is improved to a ranked output in the future. XXL implements the keyword query within the query language, in contrast we implement the keyword search on top of CQuery.

A crawler is used for index creation and maintenance. As it uses an ontology as well as information about user preferences for selection the relevant source data, it can be compared to focused crawlers (e.g. [CvD99, STS$^+$03]).

# 6 Conclusion

In this paper we presented an approach to provide keyword-based search for a mediator that integrates sources of structured data. Building a useful user interface for complex systems is a difficult task. It is even more difficult for data integration systems, because, on the one hand, the user is not able to deal with the often high complexity of the integrated structures, and, on the other hand, required integration query languages provide specialized concepts unfamiliar to common users or even experts. The proposed approach uses domain knowledge in the form of an index to map an easily specifiable keyword query to a structured query conforming to a specialized query language. Given keywords are mapped to concepts and attributes, and the query is reformulated accordingly to fit further query processing and minimize the load and transfer rates of the underlying sources.

The proposed approach can generally be used to access structured information with keyword search queries, but for the reasons mentioned above it is most useful in virtual integration scenarios. For materialized databases other solutions applying text indexing features available in most of today's commercial DBMS are conceivable.

Key aspects of the solution are the creation and the maintenance of the index used for mapping values to concepts, attributes, and categories. Apart from the techniques mentioned in this paper we currently investigate additional ways of adding mappings to the index. Furthermore, we are trying to incorporate the results of the index lookup for purposes of guidance and hinting towards further query refinement within the user interface.

# 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.

[ACGM+01] A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan. Searching the Web. *ACM Transactions on Internet Technology (TOIT)*, 1(1):2–43, 2001.

[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.

[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.

[BYRN99] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[CvD99] S. Chakrabarti, M. van den Berg, and B. Dom. Focused Crawling: a new approach to topic specfic Web resource discovery. *WWW8 / Computer Networks*, 31(11-16):1623–1640, 1999.

[DEGP98] S. Dar, G. Entin, S. Geva, and E. Palmon. DTL's DataSpot: Database Exploration Using Plain Language. In A. Gupta, O. Shmueli, and J. Widom, editors, *VLDB'98, Proceedings of 24rd Int. Conf. on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 645–649. Morgan Kaufmann, 1998.

[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.

[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.

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

[LG98] S. Lawrence and C. Lee Giles. Inquirus, the NECI Meta Search Engine. *WWW7 / Computer Networks*, 30(1-7):95–105, 1998.

[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.

[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.

[MV00] U. Masermann and G. Vossen. Design and Implementation of a Novel Approach to Keyword Searching in Relational Databases. In *ADBIS-DASFAA'2000*, pages 171–184, 2000.

[Sal89] G. Salton. *Automatic Text Processing*. Addison-Wesley, 1989.

[SE97] E. Selberg and O. Etzioni. The MetaCrawler architecture for resource aggregation on the Web. *IEEE Expert*, 12(1):8–14, Jan/Feb 1997.

[SGHS03] K. Sattler, I. Geist, R. Habrecht, and E. Schallehn. Konzeptbasierte Anfrageverarbeitung in Mediatorsystemen. In G. Weikum, H. Schöning, and E. Rahm, editors, *Proc. BTW'03 - Datenbanksysteme für Business, Technologie und Web, Leipzig, 2003, GI-Edition, Lecture Notes in Informatics*, pages 78–97, 2003.

[SGS03] K. Sattler, I. Geist, and E. Schallehn. Concept-based Querying in Mediator Systems. Technical Report 2, Dept. of Computer Science, University of Magdeburg, 2003.

[STS+03] S. Sizov, M. Theobald, S. Siersdorfer, G. Weikum, J. Graupmann, M. Biwer, and P. Zimmer. The BINGO! System for Information Portal Generation and Expert Web Search. In *CIDR'2003*, 2003.

[TW02] A. Theobald and G. Weikum. The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. In *EDBT'2002*, pages 477–495, 2002.

[vR79] C.J. van Rijsbergen. *Information Retrieval*. Butterworths, 1979.

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

[WWS+00] J. Tsong-Li Wang, X. Wang, D. Shasha, B. A. Shapiro, K. Zhang, X. Zheng, Q. Ma, and Z. Weinberg. An Approximate Search Engine for Structural Databases. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD Int. Conf. on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, page 584, 2000.