Semantic Caching in Ontology-based Mediator Systems

Marcel Karnstedt
marcel@karnstedt.com
University of Halle-Wittenberg
06099 Halle/Saale, Germany

Kai-Uwe Sattler Ingolf Geist Hagen Höpfner
{*kus*|*geist*|*hoepfner*}*@iti.cs.uni-magdeburg.de*
University of Magdeburg
P.O. Box 4120, 39016 Magdeburg, Germany

Abstract: The integration of heterogenous web sources is still a big challenge. One approach to deal with integration problems is the usage of domain knowledge in form of vocabularies or ontologies during the integration (mapping of source data) as well as during query processing. However, such an ontology-based mediator system still have to overcome performance issues because of high communication costs to the local sources. Therefore, a global cache can reduce the response time significantly. In this work we describe the semantic cache of the ontology-based mediator system YACOB. In this approach the cache entries are organized by *semantic regions* and the cache itself is tightly coupled with the ontology on the concept level. Furthermore, the cache-based query processing is shown as well as the advantages of the global concept schema in the creation of complementary queries.

1 Introduction

Many autonomous, heterogeneous data sources exists in the Web on various topics. Providing an integrated view to the data of such sources is still big challenge. In this scenario several problems arise because of autonomy and heterogeneity of the sources as well as scalability and adaptability with regard to a great number of – possibly changing – data sources. Approaches to overcome these issues are for instance metasearch engines, materialized approaches and mediator systems, which answer queries on a global schema by decomposing them, forwarding the sub-queries to the source systems and combining the results to a global answer. First generation mediators achieve integration mainly on a structural level, i.e. data from different sources are combined based on structural correspondences, e.g. the existence of common classes and attributes. Newer approaches of mediators use semantic information, such as vocabularies, concepts hierarchies or ontologies to integrate different sources. In this paper, we use the YACOB mediator system which uses domain knowledge modeled as concepts as well as their properties and relationships. The system supports the mapping of the data from the local sources to a global concept schema. The semantic information are not only used to overcome the problems resulting from heterogeneity and autonomy of the different sources but also during query processing and optimization.

However, response times and scalability are still problems because of high communication costs to the local sources. One approach to reduce the response times and improve the scalability of the system is the introduction of a *cache* which holds results of previous queries. Thus, queries can be answered (partially) from the cache saving communication costs. As page or tuple based cache organizations are not useful in distributed, heterogeneous environments, the YACOB mediator supports a *semantic cache*, i.e., the cache entries are identified by queries that generated them. This approach promises to be particularly useful because of the typical behavior of the user during the search: starting with a first, relatively inexact query the users want to get an overview of the contained objects. Subsequently, the user iteratively refines the query by adding conjuncts or disjuncts to the original query. Therefore, it is very likely that a cache contains a (partial) data set to answer the refined query.

The contribution of this paper is the description of the caching component of the YACOB mediator. We discuss different possibilities of the organization of cache according to the ontology model as well as the retrieval of matching cache entries based on a modified query containment determination. Furthermore, the paper shows the generation of complementary queries using the global concept model as well as the efficient inclusion of the cache into the query processing.

The remainder of the paper is structured as following: Section 2 gives a brief overview of the YACOB mediator system and its data model and query processing. In Section 3 the structure of the cache as well as replacement strategies and cache management with help of semantic regions is described. The query processing based on the cache is discussed in Section 4. After a comparison with the related work in Section 5 we conclude the paper with some preliminary performance results and give an outlook to our future work in Section 6.

2 The YACOB Mediator System

The YACOB mediator is a system that uses explicitly modeled domain knowledge for integrating heterogeneous data from the Web. Domain knowledge is represented in terms of concepts, properties and relationships. Here, concepts act as terminological anchors for the integration beyond structural aspects. One of the scenarios where YACOB is applicable and for which it was originally developed is the (virtual) integrated access to Web databases on cultural assets that where lost or stolen during World War II. Examples of such databases – which are in fact integrated by our system – are *www.lostart.de*, *www.herkomstgezocht.nl* and *www.restitution-art.cz*.

The overall architecture of the system is shown in Fig. 1. The sources are connected via wrappers which process simple XPath query (e.g., by translating them according the proprietary query interface of the source) and return the result as an XML document of an arbitrary DTD.

The mediator accesses the wrappers using services from the access component which forwards XPath queries via SOAP to the wrappers. The wrappers work as Web services and therefore can be placed at the mediator's site, at the source's site, or at a third place. Another part of the access component is the semantic cache which stores results of queries

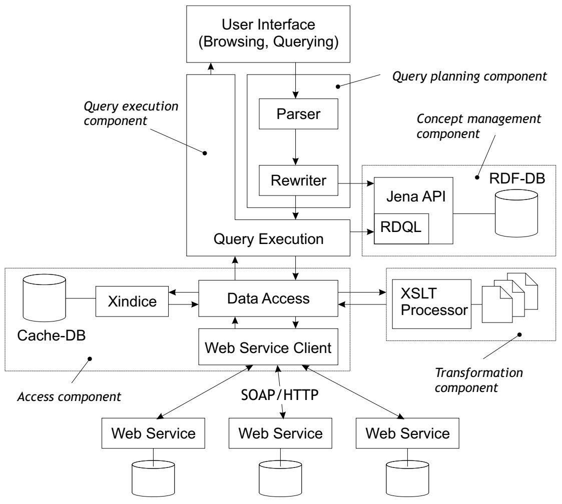


Figure 1: Architecture of the YACOB mediator

in a local database and in this way allow to use it for answering subsequent queries. This part of the system is subject of this paper and described in the following sections. Further components are the concept management component providing services for storing and retrieving metadata (concepts as well as their mapping) in terms of a RDF graph, the query planning and execution component which processes global queries as well as the transformation component responsible for transforming result data retrieved from the sources according the global schema. Architecture and implementation of this system are described in [SGHS03]. Thus, we omit further details.

Exploiting domain knowledge in a data integration system requires ways for modeling this knowledge and for relating concepts to source data. For this purpose, we use a two-level model in our approach: the *instance level* comprises the data managed by the sources and is represented using XML, the *metadata or concept level* describes the semantics of the data and is based on RDF Schema (RDFS). Here, we provide

- concepts which are classes in the sense of RDFS and for which extensions (set of instances) are available in the sources,
- properties (attributes) of concepts,
- relationships which are modeled as properties, too,
- as well as categories which represent abstract property values used for semantic grouping of objects.

These primitives are used for annotating local schemas,i.e., mappings between the global concept level and the local schema are specified in a Local-as-View manner [SGHS03].

In this way, a source supports a certain concept, if it provides a subset of the extension (either with all properties or only with a subset). For each supporting concept, a source mapping specified the local element and an optional filter restricting the instance set. Such a mapping is used both for rewriting queries as well as transforming source data in the transformation component.

In the YACOB mediator queries are formulate in CQuery – an extension of XQuery. CQuery provides additional operators applicable to the concept level such as selecting concepts, traversing relationships, computing the transitive closure etc. as well as for obtaining the extension of a concept. Concept-level operators are processed always at the global mediator level, whereas instance-level operators (filter, join, set operations) can be performed both in the mediator as well as by the source. For a detailed description of CQuery we refer again to [SGHS03, SGS03]. For the remainder of this paper, it is only important to know that a global CQuery is rewritten and decomposed into several source queries in the form of XPath expressions which can be delegated to the sources via the wrappers.

Because we are aware that for the average user of our application domain (historians, lawyers etc.) CQuery is much too complex, we hide the query language behind a graphical Web user interface combining browsing and structured querying. The browsing approach implements a navigation along the relationships (e.g. *subClass*) and properties defined in the concept level. The user can pick concepts and categories in order to refine the search. In each step the defined properties are presented as input fields allowing to specify search values for exact and fuzzy matching.

From the discussion of the architecture as well as the user interface the necessity of a cache should be obviously:

- First, accessing sources over the Web and encapsulating sources using wrappers (i.e. translating queries and extracting data from HTML pages) result in poor performance compared to processing queries in a local DBMS.
- Second, a user interface paradigm involving browsing allows to refine queries. That means, the user can *restrict* a query by removing queried concepts or by conjunctively adding predicates and he/she can *expand* a query by adding concepts or by disjunctively adding predicates. In the first case, the restricted query could be completely answered from the cache (assuming the result of the initial query was already added to the cache). In the latter case, at least portions of the query can be answered from the cache and quickly presented to the user, but additional complementary queries have to executed retrieving the remaining data from the sources.

Based on these observations, we will present in the following our caching approach that uses concepts of the domain model as anchor points for cache items and exploits – to a certain degree – domain knowledge for determining complementary queries.

3 Cache Management

The cache is designed to store result data, which is received as XML documents, and the corresponding queries, which are the semantic description of those results. If a new query arrives it has to be matched against the cached queries and possibly a (partial) result has to be extracted from the cache's physical storage (see Section 4).

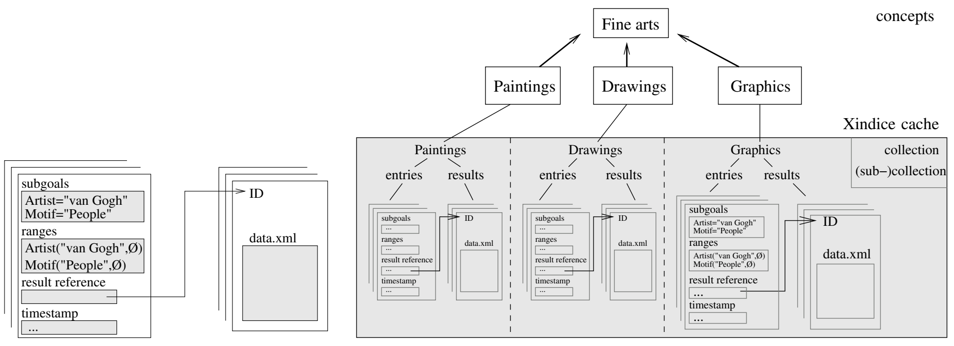
In order to realize the assumed behavior a simple and effective way of storing XML data persistently and fail-safe is needed. One way of storing is the usage of a native XML database solution which is Apache's XINDICE in this work. The open source database XINDICE stores XML documents into containers called collections. These collections are organized hierarchically, comparable to the organization of folders in a file system. The cache is placed below the ontology level, which means the cached entries are collected regarding to the queried concepts. All entries corresponding to a concept are stored in a collection named after that concept's name. The actual data is stored as it is received from the sources in a sub-collection "results", the describing data, namely the calculated "ranges" (see Section 4), the query string decomposed to the single sub-goals and a reference to the corresponding result document, are stored in another sub-collection called "entries" (Fig. 2(a)). During query matching the XML encoded entries are read from the database and the match type for the currently handled query is determined. If an entry matches, a part of or the whole result document is added to the global result data. If only a part of the cached result is needed, i.e. if the two queries overlap in some way, the part corresponding to the processed query has to be extracted. Here another advantage of using a native XML database becomes apparent: XINDICE supports XPath as its query language. In order to retrieve the required data we simply have to execute the current query against the data of the entry.

An important decision is the level of caching: we can store query results either at concept level or at source level. The difference is the form of the queries and corresponding result documents. Caching at concept level means caching queries formulated at the global schema. The queries will be transformed according to the local source schemas after processing them in the cache. The retrieved result documents stored to the cache are already transformed back to the global schema, too.

Caching at source level is placed below of the transformation component. There are separate entries for each source, because the stored queries and results are formulated in the specific schema of a source. An evident advantage of the source level cache is the finer granularity of the cached items, e.g. enabling the detection of temporarily offline sources by the cache manager. The main disadvantages – which are the benefits of using concept level caching – are the increased management overhead because of the rising amount of entries and a loss of performance at cache hit. Caching at concept level does not require any transformations at cache hit: the result is returned immediately. Additionally, a query matching making use of the global ontology, including a smart way of building a complementary query, is supported.

Fig. 2(b) shows a part of the global ontology together with an associated cache entry. This entry is created by executing the following query and storing the result in the cache:

```
//Graphics[Artist='van Gogh' and Motif='People']
```



(a) Cache entry (b) Cache structure

Figure 2: Structure of the semantic cache

Using a storage strategy as described above, the cached data is grouped together into semantically related sets called *semantic regions*. Every cache entry represents one semantic region, where the sub-goals of the predicate are conjunctive expressions. Disjunctive parts of a query get an own entry. The containment of a query is decided between the cached entries and every single conjunction of the disjunctive normal form of the query predicate. The decision algorithm is explained in detail in Section 4. The regions have to be disjoint, so each cached item is associated with exactly one semantic region. This is useful for getting a maximum of cached data when processing a query, in contrast other works let the regions overlap and avoid data redundancy using reference counters ([LC98, LC99, LC01, KB96]). There arise certain problems and open questions if the regions have to be disjoint and are forbidden to overlap. Different strategies are possible if a processed query overlaps with a cached query, more exactly their result sets are overlapping. In this case, the part of the result data already stored in the cache is extracted and a corresponding complementary query is sent to the sources. The data received as result of this query and the data found in cache form a large semantic region. Now, it have to be decided whether keeping this region or splitting it or coalescing the separate parts in some way. Because putting all data in one region will result in bad granularity and lead to storage usage problems, the regions are stored separately. There are still some remaining ways of how to split/coalesce the single parts, all effecting the query answering mechanism and possible replacement strategies.

In our approach the data for the complementary query forms a new semantic region and is inserted into the cache (inclusive the query representing the semantic description). Here, the semantic region holding the cached part of the result data is unchanged. Another possible way is to collect cached data and send the original query instead of the complementary query to the sources. The last approach is useful in the case, that matching all cached entries to a processed query results in a complementary query which causes multiple source queries or which is simply not answerable at all, e.g. due to unsupported operations such as “<>”. Details of building a complementary query and related issues are described in Section 4. In order to keep the semantic regions disjunct, it is important to store only that part in the cache which corresponds to the complementary query created before.

Both ways guarantee that the semantic regions do not overlap, which is one of the formulated constraints to the cache. Collecting the data in such disjoint regions allows a simple replacement strategy: replacement on region level, i.e. if a region has to be replaced, its complete represented data is deleted from the physical storage. This replacement strategy requires the following cache characteristics: At first, the cached regions may not be too large. On the one hand, replacing large regions means deleting a big part of the cached data and results into inefficient storage usage. On the other hand, a large amount of relatively small semantic regions leads to bad performance of the query processing. Small regions may enable a replacement based on a much finer granularity, but the cost of query processing will rise, because many regions have to be considered. Additionally, complementary queries will become very complex and last but not least because small regions mean long query strings that have to be combined when creating the complementary query.

Currently, our implemented replacement strategy is very simple. Timestamps referring to the date of collection and last reference are kept enabling a replacement strategy based on the age and referencing frequency of a cached entry. Entries are removed from the cache together with the corresponding result data, if either its cache holding time expires or if an entry has to be replaced in order to make room for a new entry. Other conceivable strategies could make use of some kind of semantic distance (like in [DFóJ$^+$96]) or other locality aspects. The timestamp strategy is sufficient for the YACOB mediator system because the main concern is to support an efficient interactive query refinement by the cache. (Dis-)advantages of other strategies have to be the subject of future work.

4 Cache-based Query Answering

Cache lookup is an integral part of the query processing approach used in the YACOB mediator. Thus, we will sketch in the following first the overall process before describing the cache lookup procedure.

In general, a query in CQuery consists of two kind of expressions: a concept level expression *CExpr* for computing a set of concepts, e.g. by specifying certain concepts, apply filter, traversal or set operations and an instance level expression *IExpr(c)* consisting of operators such as selection which are applied to the extension of each concept c computed with *CExpr*. The results of the evaluation of *IExpr(c)* for each c are combined by a union operator, which we denote extensional union $\uplus$. Thus, we can formulate a query as follows: $\biguplus_{c \in CExpr} IExpr(c)$. For example, in the following query:

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

```
    <title>$e/title</title>
    <artist_name>$e/artist</artist_name>
  </picture>
```

CExpr corresponds to the **FOR** clause and *IExpr* corresponds to the **WHERE** clause.

If a query contains additional instance level operators involving more than one extension (e.g. a join) these are applied afterwards but are not considered here because they are not affected by the caching approach.

Based on this, a query is processed as follows. The first step is the evaluation of the concept level expression. For each of the selected concept we try to answer the instance level expression by first translating it into an XPath query and applying this to the extension. Basically, this means to send the XPath query to the source system. However, using the cache we can try to answer the query from the cache. For this purpose, the function *cache-lookup* returns a (possibly partial) result set satisfying the query condition, i.e., if necessary additional filter operations are applied to the stored cache entries, as well as a (possibly empty) complementary XPath query. In case of an non-empty complementary query or if no cache entry was found, the XPath query is further processed by translating it according to the concept mapping *CM*(c) and send this translated query to the corresponding source s. Finally, the results of calling *cache-lookup* and/or *process-source-query* are combined.

```
Input:
   query expression in the form of $\biguplus_{c \in CExpr} IExpr(c)$
   result set $R := \{\}$
1  compute concept set $C := CExpr$
2  forall $c \in C$ do
3     /* translate query into XPath */
4     $q :=$ toXPath(IExpr($c$))
5     /* look for the query $q$ in the cache $\rightarrow$ result is denoted by $R_c$,
6     $\overline{q}$ is returned as complementary query for $q$ */
7     $R_c :=$ cache-lookup($q, \overline{q}$)
8     if $R_c \neq \{\}$ then
9        /* found cache entry */
10       $R := R \biguplus R_c$
11       $q := \overline{q}$
12    fi
13    if $q \neq$ empty then
14       $q_s :=$translate-for-source($q$, CM($c$))
15       $R_s :=$process-source-query($q_s$, $s$)
16       $R := R \biguplus R_s$
17    fi
18 od
```

Figure 3: Steps of Query Processing

Note that in this context complementary queries are derived at two points:

match type (Q, C)	situation	cached part of result data	complementary query
exact	data to Q and C identical	C's data	none
containing	C containing Q	Q on C's data	none
contained	C contained in Q	C's data	$Q \wedge \neg C$
overlapping	data to C and Q overlaps	Q on C's data	$Q \wedge \neg C$
disjoint	C's data is no part of result data	none	Q

Table 1: Possible match types between processed query Q and cached entry C

- First, because a global query is decomposed into a set of single concept related queries, complementary queries are derived implicitly. This means, if one wants to retrieve in query q the extension **ext**(c) of a concept c with two sub-concepts c_1 and c_2 where it holds **ext**(c) = **ext**$(c_1) \cup$ **ext**(c_2) and **ext**(c_1) is already stored in the cache, two queries q_1 (for **ext**(c_1)) and q_2 (for **ext**(c_2)) have to be processed. However, because we can answer q_1 from the cache, only the complementary query q_2 needs to be executed (i.e., $q_2 = \overline{q}$), which is achieved by iterating over all concepts of the set C (line 2).
- Secondly, if the cache holds only a subset of **ext**(c_1) restricted by a certain predicate p we have to determine $\overline{q_1}$ with $\neg p$ during the cache lookup.

Because the first issue is handled as part of the query decomposition, we focus in the following only on cache lookup.

We will use the example started in Section 3 to picture this step. Let us assume, we are in a very early (almost starting) state of the cache, where the query

```
//Graphics[Artist='van Gogh' and Motif='People']
```

is the only stored query referencing the result file `data.xml`. Now, the new query has to be processed:

```
//Graphics[(Artist='van Gogh' or Artist='Monet') and Date='ca 1600']
```

During the cache lookup every conjunction found in the disjunctive normal form of the processed query is matched against each cache entry in no special order. As mentioned in Section 3 the semantic regions do not overlap. Thus, independent from the order of processing cache entries, all available parts of the result can be found in cache. In other words, if an entry contains a part of the queried data no other one will contain this data, e.g. if an exact match to the query exists, there will be no other containing match and the exact match will be detected independent of any entries observed before. There are five possible match types that may occur which are summarized in Tbl. 1.

The match types are listed top down in the order of their quality. Obviously, the exact match is the best one, because the query can be answered simply by returning the all data of the entry. If no exact match can be found, a containing match would be the next best.

In this case, a superset of the needed data is cached and can be extracted by applying the processed query Q to the cached documents. The cases of contained and disjoint match type require to process a complementary query. Only a portion of the required data is stored and the complementary query retrieves the remaining part from the source.

Considering our example the disjunctive normal form of the new query is:

```
//Graphics[(Artist='van Gogh' and Date='ca 1600')
  or (Artist='Monet' and Date='ca 1600')]
```

The two parts in brackets are the conjunctions we have to handle separately during the cache lookup.

The algorithm in Fig. 4 displays the procedure of cache lookup in pseudo code.

The match type between the processed and the cached query is determined by calling the procedure *match-type* (line 10). This procedure implements a solution to the query containment problem and is discussed later in this section.

Running over all cached entries we only have to match the query remaining from the step before instead of having to match the original query again each time. This reflects a part of the result data already been found. This data cannot occur in the semantic regions described by other entries and is not needed to retrieved from the sources. In cases of exact and containing matches there will no query remain, because all of the data can be found in the cache and therefore the cache lookup is finished (lines 13 to 18). In all other cases, we still have to match against the remaining entries. If we encounter a disjoint match, the currently checked conjunction has to be matched against the next entry (line 12). In contrast, if a part of the result data is found, only the complementary query is to process further on. In order to avoid checks of conjunctions created during the generation of the complementary query, we first collect them separately (lines 21 and 25) and add them to the set of later checked conjunctions (line 29). The query remaining after checking all (possibly new created) conjunctions against all entries is built in $\overline{q}$ (line 32). Here, the operation '+' denotes a concatenation of the existing query $\overline{q}$ and all conjunctions in CC by a logical OR. If all entries belonging to the queried concept are checked, a possibly remaining query have to be used to fetch data which could not be found in the cache. The procedure returns a set of all collected references to parts of the result stored in cache as well as the complementary query (34). This query is sent to the sources and – in parallel – the cached data are extracted from physical storage.

In the simple example introduced above only one cache entry is created which we have to check for a match. Here, we get an overlapping match between the cached query and the first of the two conjunctions. Thus, the complementary query looks as follows:

```
//Graphics[Artist='van Gogh' and Date='ca 1600' and Motif != 'People']
```

This expression is added to the global complementary query, because no further entry is left that we could check and therefore, we cannot find any further parts of the result data in cache. Comparing the second query conjunction we receive a disjoint match, because the query predicates together are unsatisfiable in the attribute `Artist`. After checking against all existing entries this conjunction becomes part of the complementary query unchanged. The final complementary query is:

```
**Input**:
    Query $q$
**Output**:
    result set $R := \{\}$
    complementary query $\overline{q} :=''$

    $q'$ = *disjunctive-normal-form*($q$);
    $E$ = *get-cache-entries* (*get-concept*($q$));
    **forall** conjunction *Conj* of $q'$ **do**
        $CC = \{Conj\}$;                    /* current conjunctions (still to check) */
        **forall** cache entry $C \in E$ **do**
            $NC := \emptyset$;              /* new conjunctions (to check) */
            **forall** conjunctions $Conj' \in CC$ **do**
                $M$ := *match-type*($Conj', C$);
                **switch** ($M$) **do**
                    **case** 'disjoint':     **break**;
                    **case** 'exact':        $R := R \cup C \rightarrow Data$;
                                             $CC := CC \setminus \{Conj'\}$;
                                             **break**;
                    **case** 'containing':   $R := R \cup Conj'(C \rightarrow Data)$;
                                             $CC := CC \setminus \{Conj'\}$;
                                             **break**;
                    **case** 'contained':    $R := R \cup C \rightarrow Data$;
                                             $CC := CC \setminus \{Conj'\}$;
                                             $NC := NC \cup \{(Conj' \wedge \neg C)\}$;
                                             **break**;
                    **case** 'overlapping':  $R := R \cup Conj'(C \rightarrow Data)$;
                                             $CC := CC \setminus \{Conj'\}$;
                                             $NC := NC \cup \{(Conj' \wedge \neg C)\}$;
                                             **break**;
                **od**
            **od**
            $CC := CC \cup NC$;
            **if** $CC = \{\}$ **then break**;
        **od**
        **if** $CC \neq \{\}$ **then** $\overline{q} := \overline{q} + CC$;
    **od**
    **return** $R, \overline{q}$;
```

Figure 4: Procedure *cache-lookup*

```
//Graphics[(Artist='van Gogh' and Date='ca 1600' and Motif != 'People')
  or (Artist='Monet' and Date='ca 1600')]
```

The cached part of the result data is extracted from the cache database by applying

```
//Graphics[Artist='van Gogh' and Date='ca 1600']
```

to the result document `data.xml`, which is done using the XPath query service provided by XINDICE.

Match Type Determination. In order to determine the match type between processed and cached query the problem of query containment has to be solved, symbolized in the pseudo code by calling method *match-type*. We can restrict the general problem to a containment on query predicates. In the YACOB mediator all predicates are in a special form: they are sets of sub-goals combined by logical OR and/or AND. The sub-goals are only simple attribute/constant expressions, limited to $X\,\theta\,c$, where X is an attribute, c is an constant and $\theta \in \{=, \neq, \sim=\}$. We do not have to forbid numerical constants and the corresponding operations (it is easy to adapt the implemented containment algorithm to numerical domains), but in fact there are currently only attributes defined on string domains in YACOB. The algorithm is based on solutions to the problems of satisfiability and implication. The NP-hardness of the general containment problem is not given here because of the limitation that only constants may appear on the right side of an expression. In all solutions to the problem found in literature the NP-hardness results from allowing the `!=` operator together with comparisons between attributes defined on integer domains. See [GSW96] for a good comparison.

The basic idea is to parse the query and to apply a range for every identified sub-goal. For each attribute these ranges contain the values the attribute may contain. When determining the containment between two queries it is done using the ranges created before.

A special treatment is required for CQuery's text similarity operator '$\sim=$' which is mapped to appropriate similarity operators of the source query interfaces such as `like` or `contains`. In order to decide if a cached query matches the current query, we have to detect such similarities between attribute values without knowing about how actual semantics of the similarity operation in the source system. For solving this problem, we have chosen a pragmatic approach: if a query uses the '$\sim=$' operator the result will include all similar objects, e.g. querying `Artist`$\sim=$`'Gogh'` will return objects with `Artist='v. Gogh', Artist='van Gogh, V.', Artist='v. Gogh, V'` etc. If later a new query filters an attribute value similar to the element stored in cache (in the given example for instance `Artist`$\sim=$`'van Gogh'`) the cached results can be used.

We have implemented the handling of the similarity operator based on substrings. Assuming two queries having only one condition in their predicates, one an attribute A with A$\sim= x$ and the other on the same attribute with A$\sim= y$. Now, if the string x is contained in y as a substring, the cache entry to A$\sim= x$ contains all data data belonging to A$\sim= y$. Otherwise, if the result for A$\sim= y$ is cached and the query is A $\sim= x$ we encounter a contained match.

Complementary Query Construction. The construction of the complementary query in cases of contained and overlapping match types has to be examined in a detailed way. The objective is to create a new query which queries only the data not already found in cache. If the processed query is q and the query of the cached entry is C the new query is $q \wedge \neg C$. This query is obtained by negating each sub-goal c_i of C and combining it with query q by a logical AND. This results into n parts, where n is the number of sub-goals in C, all OR-combined. So, the generated query will extract the data belonging to the result of q not found in C and it is already in disjunctive normal form. In general, some of the n constructed conjunctions in will be unsatisfiable. The following example illustrates this:

- cached query: `//Graphics[Artist='van Gogh' and Motif='People']`
- new query: `///Graphics[Artist='van Gogh' and Date='ca 1600']`
- complementary query:

  ```
  //Graphics[(Artist='van Gogh' and Date='ca 1600'
             and Artist!='van Gogh') or (Artist='van Gogh'
             and Date='ca 1600' and Motif!='People')]
  ```

- pruned to satisfiable parts:

  ```
  //Graphics[Artist='van Gogh' and Date='ca 1600'
     and Motif!='People']
  ```

The implemented satisfiability algorithm is used to detect and delete these parts of the complementary query.

The more entries we find overlapping to the processed query, the more complex the complementary query will become. Each of the cached entries will expand the query by some sub-goals. At the end, the resulting query could be too complex that an efficient processing by the sources is not possible or that they cannot be answered at all. Thus, we need some heuristics to decide whether the constructed complementary query is too complex to process it. If so, the original query should be sent to the sources instead accepting a higher network load and the need for duplicate elimination. However, the cache still supports a fast creation and delivering of an answer set to the user. Such heuristics could be based on query capability descriptions of the sources, but this is currently not supported by our system.

5 Related Work

The entries in a semantic cache are organized by semantic regions. Therefore, the selection of relevant cache entries for answering a query is based on the problems of query containment and equivalence. There are several publications which focus on different aspects of query containment, such as completeness, satisfiability as well as complexity. Surveys of these approaches can be found for instance in [GSW96, Hal01].

Caching data in general and semantic caching in particular are common approaches for reducing response times and transmission costs in (heterogeneous) distributed information systems. These works comprise classical client-server databases [DFóJ+96, KB96], heterogeneous multi-database environments [GG97, GG99], Web databases [LC98, LC01] as well as mobile information systems [LLS99, RD00].

The idea of *semantic regions* was introduced by Dar et al. [DFóJ+96]. Semantic regions are defined dynamically based on the processed queries which are restricted on selection conditions on single relations. Constraint formulas describing the regions are conjunctive representations of the used selection predicates. Thereby, the regions are disjoint.

The semantic query cache (SQC) approach for mediator systems over structured sources is presented in [GG97, GG99]. The authors discuss most aspects of semantic caching: determining when answers are in cache, finding answers in cache, semantic overlapping and semantic independence and semantic remainder in a theoretical manner. Our approach also reflects most addressed aspects, but deals with semistructured data.

Keller and Basu describe in [KB96] an approach for semantic caching that examines and maintains the contents of the cache. Therefore, the predicate description of executed queries are stored on the client as well as on the server. Queries can include selections, projections and joins over one ore more relations but results have to comprise all keys that have been referenced in the query.

Semantic caching in Web mediator systems is proposed in [LC98, LC01]. Most ideas in this approach are based on [DFóJ$^+$96]. However, the authors also introduce a technique that allows the generation of cache hits by using of additional semantic background information. In contrast to our approach, the cache is not located in the mediator access component but in the wrappers. As discussed in the previous sections a tight coupling to the global ontology structure was chosen in the YACOB system.

In mobile information systems semantic caches are typically used for bridging the gap between the portability of mobile devices and the availability of information. The LDD cache [RD00] is optimized in order to cache location depended information, but in fact, uses also techniques which are common for semantic caches. Results are cached on the mobile devices and are indexed with meta information which are generated from the queries. But in this approach the index is only a table which references a logical page which is similar to semantic regions.

6 Discussion and Conclusions

Semantic caching is a viable approach for improving response times and reducing communication costs in a Web integration system. In this paper, we have presented a caching approach which we developed as part of our YACOB mediator system. A special feature of this approach is the tight connection to the ontology level – the cache is organized along the concepts. Furthermore, the modeled domain knowledge is exploited for obtaining complementary queries required for processing queries which can only partially answered from cache.

For evaluation purposes, we ran some preliminary results in our real-world setup[1] showing that response times for queries which can be answered from the cache are reduced by a factor of 4 to 6. However, the results depend strongly on the query mix, i.e. the user behavior, as well as on the source characteristics (e.g. response time and query capabilities), so we omit details here. Currently, we evaluate the caching approach using different strategies for replacement and determining complementary queries in a simulated environment.

In future research, we plan to exploit more information from the concept level in order to

[1]http://arod.cs.uni-magdeburg.de:8080/Yacob/index.html

reduce the effort for complementary queries.

References

[DFóJ$^+$96] S. Dar, M. J. Franklin, B. ór Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *VLDB'96, Proc. of 22th Int. Conf. on Very Large Data Bases*, pages 330–341, Mumbai (Bombay), India, September 3–6 1996. Morgan Kaufmann.

[GG97] P. Godfrey and J. Gryz. Semantic Query Caching for Heterogeneous Databases. In *Intelligent Access to Heterogeneous Information, Proc. of the 4th Workshop KRDB-97, Athens, Greece*, volume 8 of *CEUR Workshop Proceedings*, pages 6.1–6.6, August 30 1997.

[GG99] P. Godfrey and J. Gryz. Answering Queries by Semantic Caches. In *Database and Expert Systems Applications, 10th Int. Conf., DEXA '99, Florence, Italy, Proc.*, volume 1677 of *LNCS*, pages 485 – 498. Springer, August 30 - September 3 1999.

[GSW96] S. Guo, W. Sun, and M. A. Weiss. On Satisfiability, Equivalence, and Implication Problems Involving Conjunctive Queries in Database Systems. *TKDE*, 8(4):604–616, August 1996.

[Hal01] A. Y. Halevy. Answering Queries using Views: A Survey. *VLDB Journal: Very Large Data Bases*, 10(4):270–294, December 2001.

[KB96] A. M. Keller and J. Basu. A Predicate-based Caching Scheme for Client-Server Database Architectures. *VLDB Journal: Very Large Data Bases*, 5(1):35–47, January 1996.

[LC98] D. Lee and W. W. Chu. Conjunctive Point Predicate-based Semantic Caching for Wrappers in Web Databases. In *CIKM'98 Workshop on Web Information and Data Management (WIDM'98), Washington, DC, USA*, November 6 1998.

[LC99] D. Lee and W. W. Chu. Semantic Caching via Query Matching for Web Sources. In *Proc. of the 1999 ACM CIKM Int. Conf. on Information and Knowledge Management, Kansas City, Missouri, USA*, pages 77–85. ACM, November 2–6 1999.

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

[LLS99] K. C. K. Lee, H. V. Leong, and A. Si. Semantic Query Caching in a Mobile Environment. *ACM SIGMOBILE Mobile Computing and Communications Review*, 3(2):28–36, April 1999.

[RD00] Q. Ren and M. Dunham. Using Semantic Caching to Manage Location Dependent Data in Mobile Computing. In *Proc. of the 6th Annual Int. Conf. on Mobile Computing and Networking (MOBICOM-00)*, pages 210–242, New York, August 6–11 2000. ACM.

[SGHS03] K. Sattler, I. Geist, R. Habrecht, and E. Schallehn. Konzeptbasierte Anfrageverarbeitung in Mediatorsystemen. In *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.