Efficient Cache Answerability for XPath Queries

Pedro José Marrón and Georg Lausen

University of Freiburg
Institute of Computer Science
Georges-Köhler-Allee,
79110 Freiburg, Germany
{pjmarron,lausen}@informatik.uni-freiburg.de

Abstract. The problem of cache answerability has traditionally been studied over conjunctive queries performed on top of a relational database system. However, with the proliferation of semistructured data and, in particular, of XML as the de facto standard for information interchange on the Internet, most of the assumptions and methods used for traditional systems – and cache answerability is no exception – need to be revisited from the point of view of the semistructured data and query model. In this paper, we present a formal framework for the efficient processing of XPath queries over XML documents in a cache environment that is based on the classic rewriting approach. Furthermore, we provide details on the implementation of our formal methods on top of HLCaches, an LDAP-based distributed caching system for XML, and argue that our approach is more efficient than traditional query rewriting algorithms while, at the same time, supporting the full expressive power of XPath queries.

Keywords: Semistructured data, cache answerability, query rewritability, XML, XPath, LDAP

1 Introduction

Cache answerability has been traditionally studied in the realm of conjunctive predicates and queries performed on top of relational database systems [Lev00], but the increasing interest in recent years on the characteristics and capabilities of semistructured models and, in particular, XML [BPSMM00], have lead to the restatement of the cache answerability problem in terms of the semistructured data and query model [CGLV00,KNS99,PV99]. Furthermore, the proliferation of techniques to perform data integration (let it be semistructured or not) on the Internet strive the need for efficient cache mechanisms.

The use of XPath [CD99] and XPath-based models for the querying and processing of semistructured data has changed the focus of the rewriting algorithms from conjunctive predicates to regular path queries [CGLV00], or other query languages specifically designed for a particular semistructured data model [PV99].

Other query caching systems, like [LRO96], [DFJ+96] or [QCR00], do not take into consideration semistructured data, and although interesting in their approach, cannot be used in the context our model can be brought up.

The approach we take in our work, and therefore, the focus of this paper, is on the definition of a very simple, but highly efficient general-purpose formal model that allows us to tackle the problem of cache answerability for XML from a more pragmatic perspective than the one usually taken by traditional papers on the topic. The generality of our model enables its implementation on any XPath-aware caching system, and in order to show its feasibility, we have implemented it as part of HLCaches [ML01,Mar01], a hierarchical LDAP-based caching system for XML.

In our system, the methods and algorithms described throughout this paper serve as the basis for the efficient processing of XPath queries in the distributed caching environment offered by HLCaches, since it allows the definition of partial XPath query evaluation techniques, query preprocessing mechanisms, and parallel processing routines that are crucial for the maintenance of the level of availability and processing capabilities expected from an distributed caching system.

Axis Name	Considered Nodes
ancestor	Any node along the path to the root
ancestor-or-self	Same, but including the current node
attribute	Consider only attribute nodes in the tree
child	Any node directly connected to the current node
descendant	Any node from the subtree rooted at the current node
descendant-or-self	Same, but including the current node
following	Any node with id greater than the current node, excluding its descendents
following-sibling	Any same-level node with id greater than the current node
parent	The direct predecessor of the current node
preceding	Any node with id lower than the current node, excluding its ancestors
preceding-sibling	Any same-level node with id lower than the current node
self	The current node

Table 1. Allowed Axis Expressions in XPath

This paper is structured as follows: Section 2 presents a formal description of the XPath query model needed to understand the reformulation of the cache answerability problem detailed in section 3. Section 4 provides an insight in some of the more important implementation issues related to our model, and section 5 concludes this paper.

2 XPath Query Model

As specified in the XPath standard [CD99], the primary purpose of the XPath query language is to address parts of an XML document, usually represented in the form of a tree that contains element, attribute and text nodes.

An XPath Query Q_X is formed by the concatenation of path expressions that perform walk-like operations on the document tree retrieving a set of nodes that conform to the requirements of the query. Each expression is joined with the next by means of the classical Unix path character '/'.

Definition (XPath Query) An XPath Query Q_X is defined as: $Q_X = /q_0/q_1/\ldots/q_n$, where q_i is an XPath subquery defined below, and '/' the XPath subquery separator. □

Definition (XPath Subquery) An XPath Subquery q_i is a 3-tuple $q_i = (C_i, w_i, C_{i+1})$, where:

- C_i is a set of XML nodes that determine the input context.
- w_i is the Path Expression to be applied to each node of the input context (defined below).
- C_{i+1} is a set of XML nodes resulting from the application of the path expression w_i onto the input context C_i. C_{i+1} is also called the output context. □

Definition (XPath Path Expression) A Path Expression w_i is a 3-tuple $w_i = a_i :: e_i[c_i]$, such that:

- a_i is an axis along which the navigation of the path expression takes place (see table 1 for a complete list).
- e_i is a node expression that tests either the name of the node or its content type.
- c_i is a boolean expression of conditional predicates that must be fulfilled by all nodes in the output context. □

Example The query $Q_X = /child :: mondial/child :: country[attribute :: car\_code = \text{“}D\text{”}]$ is composed of two subqueries whose combination selects all `country` nodes directly connected to the `mondial` child of the document root, that have an attribute `car_code` with value “`D`”. □

More formally, and using the classic predicate-based approach found in most rewriting papers, the evaluation of a query Q_X, can be defined in terms of the evaluation of its respective subqueries by means of the following predicate:

Definition (XPath Subquery Evaluation) Given an XPath subquery $q_i = (C_i, w_i, C_{i+1})$, where C_i is the input context, w_i is a path expression, and C_{i+1} the evaluation of w_i on C_i (also called the output context), we define its evaluation by means of the *eval* predicate, as follows:

$$C_{i+1} = eval(C_i, w_i)$$

where the *eval* predicate is simply an abbreviation of the following expression:

$$eval(C_i, w_i) = \bigcup_{n \in C_i} (evalNode(n, w_i))$$

where $evalNode$ performs the evaluation of w_i over a single input node, returning all other nodes in the document that satisfy w_i. □

Definition (XPath Query Evaluation) Given the XPath query $Q_X = /q_0/\ldots/q_n/$, its evaluation is defined in terms of the *eval* predicate as follows:

$$\begin{aligned} Q_X = C_{n+1} &= eval(C_n, w_n), \quad \text{where} \\ C_{i+1} &= eval(C_i, w_i), 0 \leq i \leq n \end{aligned}$$

The result of the query is simply the last output context from subquery q_n, that in turn, depends on the output context of q_{n-1}, and so on.

As defined in the XPath standard [CD99], C_0 is said to contain only the root of the document tree. □

Given the highly serial characteristics of the XPath query model, the evaluation process for a given XPath query can be easily visualized using the graphical representation of figure 1, where, as an example, the evaluation process of a query consisting of seven subqueries is depicted. The ovals inside each context between two subqueries indicate the individual XML nodes that satisfy the subquery at each point.

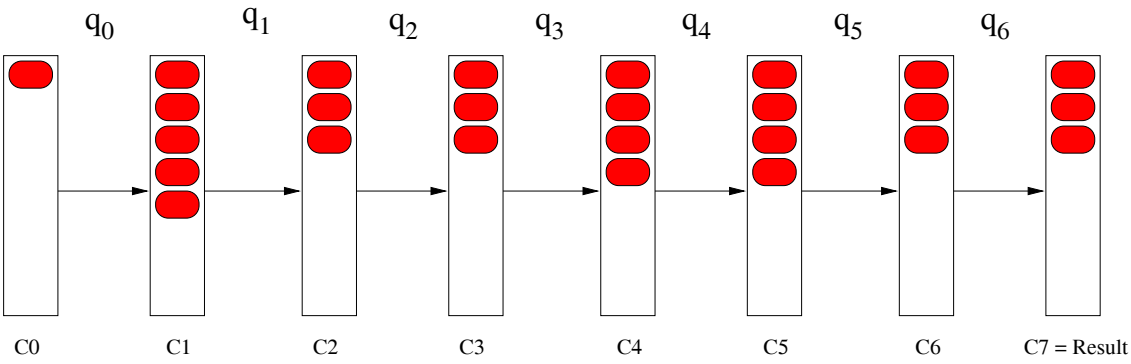

Fig. 1. XPath Query Evaluation Example

It is worth mentioning at this point that the evaluation of a given subquery q_i involves the application of q_i on each one of the individual nodes contained in the previous context, so that it is possible to keep track of which node in context C_i generates what set of nodes from C_{i+1}.

3 Cache Answerability

Cache answerability is the basis for more complex problems whose solution usually implies the more efficient processing of queries on a given system. As we have already mentioned in the introduction, data integration, and in particular semistructured data integration is becoming more and more common, and requires efficient cache solutions that must rely on efficient cache answerability algorithms.

The problem of cache answerability is usually reduced to determining, given a particular query, whether or not there exists a rewriting for the query in terms of elements or predicates already known to the cache, assuming that the retrieval of results from the cache can be performed more efficiently than their repeated evaluation. Of course, this is only the case if the collection and maintenance of information in the cache can be implemented in such a way that the path taken by the query for its evaluation is not slowed down by the data gathering phase used as the basis for the query rewriting algorithms.

In our case, in order to fully support the rewriting of XPath queries, we only need to store the output context of each subquery in our cache as it is evaluated in the first place. The predicate $cache(I, w^c)$, defined in exactly the same way the *eval* predicate was introduced in the previous section, is stored in the cache, and contains the set of path expressions with their respective input and output contexts that have been evaluated by the cache thus far. The difference between the *eval* and *cache* predicates lies in the efficiency of their implementation. Whereas the former needs to invoke a parser on the subquery expression and evaluate it on top of the document tree, the latter simply performs a look up on the current contents of the cache to immediately retrieve the answer to the subquery.

Using these definitions, and taking into account that the nature of XPath expressions allows us to perform rewritings at the subquery level, we obtain the following definitions:

Definition (Equivalent Rewriting) Given an XPath subquery $q_i = (C_i, w_i, C_{i+1})$ that needs to be evaluated by the application of an $eval(C_i, w_i)$ predicate, an equivalent rewriting is another predicate $cache(I^c, w^c)$, such that the following properties hold:

- $w = w^c$; and
- $C_i = I^c$.

The evaluation of the subquery q_i is then, $C_{i+1} = cache(I^c, w^c)$. □

However, looking at this definition, it is clear that we can relax the second constraint to allow for a greater number of equivalent rewritings to be detected.

Definition (Weak Equivalent Rewriting) Given an XPath subquery $q_i = (C_i, w_i, C_{i+1})$ that needs to be evaluated by the application of an $eval(C_i, w_i)$ predicate, a weak equivalent rewriting is another predicate $cache(I^c, w^c)$, such that the following properties hold:

- $w = w^c$; and
- $C_i \subseteq I^c$.

The evaluation of the subquery q_i, and therefore, the contents of C_{i+1} is then the set of nodes in the output context generated as a consequence of the evaluation of w_i on the input context C_i, that is, $C_{i+1} = cache(C_i, w^c)$.

In other words, if our cache contains a superset of the answers needed to provide a rewriting for the subquery q_i, we are still able to evaluate the subquery q_i only with the contents of our cache. □

Finally, we can define what it means for a rewriting to be partial:

Definition (Partial Equivalent Rewriting) Given an XPath subquery $q_i = (C_i, w_i, C_{i+1})$ that needs to be evaluated by the application of a $eval(C_i, w_i)$ predicate, a partial equivalent rewriting is another predicate $cache(I^c, w^c)$, such that the following properties hold:

– $w = w^c$; and
– $C_i \supseteq I^c$.

Then, the evaluation of the subquery q_i is the set of nodes in the output context found in the partial equivalent rewriting, plus the set of nodes that needs to be evaluated by means of the *eval* predicate, that is, $C_{i+1} = cache(I^c, w^c) \cup eval(C_i \setminus I^c, w)$. □

These definitions allow us to create a framework where, independently of the storage model used for XML documents, and taking only the characteristics of the XPath query language into account, the problem of finding equivalent rewritings for a given query and, by extension, the problem of cache answerability can be very easily solved.

Let us illustrate the functionality of our framework with an example:

Example Let us assume that our cache contains an instance of the mondial database [May], where various pieces of information about geopolitical entities is stored. Let us also assume that the query $Q_1 = /mondial/country//city$ has already been evaluated, and its intermediate results stored in our cache by means of several *cache* predicates, namely:

$$\begin{aligned} cache\ contents = & cache(C_0, \text{“}/mondial\text{”}), \\ & cache(O_1^c, \text{“}/country\text{”}), \\ & cache(O_2^c, \text{“}//city\text{”}) \end{aligned}$$

where O_1^c and O_2^c are the stored results of the evaluation of “/mondial” on C_0 and “/country” on O_1^c, respectively.

In order to evaluate the query $Q_X = /mondial/country[car\_code = \text{“D”}]//city$ using the *eval* predicate, we need to solve the following expression:

$$\begin{aligned} Q_X = C_3 = & eval(C_2, \text{“}//city\text{”}), \quad \text{where} \\ & C_2 = eval(C_1, \text{“}/country[car\_code = \text{“D”}]\text{”}) \\ & C_1 = eval(C_0, \text{“}/mondial\text{”}) \end{aligned}$$

However, the evaluation of Q_1 provided us with a series of *cache* predicates that we can use in order to rewrite Q_X as follows:

$$\begin{aligned} Q_X = C_3 = & cache(C_2, \text{“}//city\text{”}), \quad \text{where} \\ & C_2 = eval(C_1, \text{“}/country[car\_code = \text{“D”}]\text{”}) \\ & C_1 = cache(C_0, \text{“}/mondial\text{”}) \end{aligned}$$

As we can see from the cache contents detailed above, we can find two equivalent rewritings, one for the first subquery, and another one for the last subquery. The subquery in the middle does not exist in our cache, and therefore, needs to be evaluated by means of the *eval* predicate.

In this example, we can see two different kinds of rewritings: an equivalent rewriting for the $/mondial$ subquery, since the contents of C_0 are fixed and defined to be the root of the XML data, and a weak equivalent rewriting for the $//city$ subquery, since the contents of C_2 are a subset of the contents of O_2^c defined in the cache. This is obvious since the query $/country$ retrieves all `country` nodes in the document, whereas $/country[car\_code = \text{“D”}]$ selects only one `country` node from all existing countries. □

4 Implementation Issues

As we have already mentioned, our model has been implemented as part of the query evaluation engine of the HLCaches system, whose basic structure and evaluation algorithms have been

published in [ML01]. However, the generality of the model described in the previous section allows for our mechanisms to be implemented and deployed not only in HLCaches, whose XML storage model is based on LDAP [WHK97,HSG99], but on any XPath processing system that follows the XPath standard [CD99].

In order to provide an implementation of our model, the following functionality needs to be provided:

XML Data Model: An efficient storage and retrieval mechanism for XML.
XPath Evaluation Model: The implementation of the *eval* predicate in such a way, that the evaluation of a subquery is completed before the evaluation of the next subquery starts. This requirement is needed due to the highly serial nature of the XPath evaluation model.
Cache Data Model: Storage of cache contents (the *cache* predicate) in structures that allow for their efficient checking and retrieval.
Cache Evaluation Model: Algorithms or query types used to determine the result of a particular *cache* predicate.

In HLCaches, we have implemented this model using the following approaches:

4.1 XML Data Model

LDAP is used in HLCaches as the underlying representation model for the encoding of arbitrary XML documents. The exact representation, as well as the internal details of the storage mechanisms fall out of the scope of this paper, but the interested reader is referred to the aforementioned publications.

For the purposes of our discussion regarding the implementation of our model, it suffices to know that an LDAP server maintains the directory *schema* model (equivalent to a DTD [BPSMM00] or XMLSchema [Fal01,TBMM01] representation), and the directory *data* model.

The directory schema model manages the meta-data about the contents of the LDAP tree, which implies the storage of mainly three types of information:

LDAP Schema Class Hierarchy: Contains information about the **required** and **allowed** attributes a particular class of nodes are able to store, as well as the hierarchical relationships among the different classes of nodes in the tree.
Valid Attributes: Represent the set of recognized attributes as well as their type, which determines the kind of search and modify operations allowed on a specific attribute.
Type Definition: Stores the set of allowable types that can be given to a specific attribute.

The directory instance, on the other hand, manages a set of nodes and their respective attributes whose representation, similarly to XML, is a tree-based structure that can be stored and retrieved very efficiently. The hierarchical structure of an LDAP directory is kept by means of two special purpose attributes: **object class** (or **oc** for short) that stores the set of classes a node belongs to, and a **distinguished name** (or **dn** for short), defined below:

Definition (LDAP Distinguished Name) An LDAP distinguished name is a comma separated sequence of attribute-value pairs that uniquely identifies a particular node in the LDAP tree.

A distinguished name for a particular entry is formed by taking the distinguished name of the parent node in the hierarchy, and prepending an attribute-value pair unique to all the siblings of the corresponding node. This attribute-value pair is referred to as the **relative distinguished name**. □

Since the distinguished name contains each relative distinguished name from a particular node up to the root, and there is only one parent for each node, the distinguished name is enough to uniquely identify a particular entry in the instance hierarchy.

As it can be seen in the following example, attributes in LDAP are multivalued, that is, there is no restriction on the number of values a particular attribute in a specific node is allowed to

take. This allows the **oc** attribute to store all the (potentially many) classes a particular node in the LDAP tree is an instance of.

Example (LDAP Directory Instance) Figure 2 contains a graphical representation of an instance in an LDAP directory, where both, the use of distinguished names to represent the hierarchical relationships, and the purpose of attribute names to store information about a particular node is explicitly stated. □

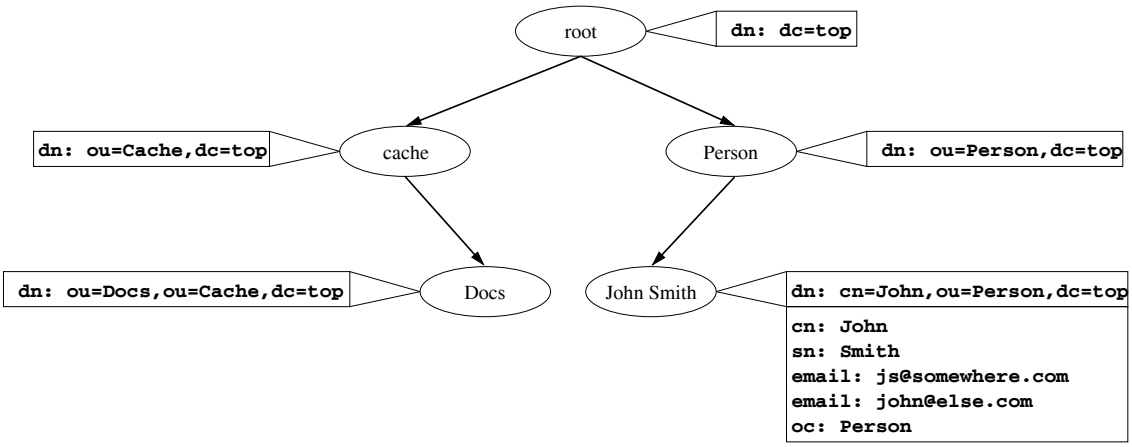

Fig. 2. LDAP Directory Instance Example

The similarities between the LDAP and XML model allow us to store XML documents without the need to provide cumbersome transformations like the ones needed to represent XML in relational databases, making LDAP the ideal underlying data model for implementation on a cache.

4.2 XPath Evaluation Model

Similarly, the Lightweight Directory Access Protocol offers a querying model based on filter specification that happens to be very close in nature to that of native XPath, so that, as detailed in [ML01], every XPath query can be translated into LDAP queries of the form:

Definition (LDAPQL Query) An LDAPQL Query Q_{HL} is a 4-tuple

$$Q_{HL} = (b_{Q_{HL}}, s_{Q_{HL}}, f_{Q_{HL}}, p_{Q_{HL}})$$

such that:

- $b_{Q_{HL}}$ is the distinguished name of the base entry in the directory instance where the search starts from.
- $s_{Q_{HL}}$ is the scope of the search, which can be:
  `base` if the search is to be restricted to just the first node,
  `onelevel` if only the first level of nodes is to be searched,
  `subtree` if all nodes under the base should be considered by the filter expression,
  `ancestors` if all the ancestors of the node up to the root are to be searched.
- $f_{Q_{HL}}$ is the filter expression defined as the boolean combination of atomic filters of the form $(a\ op\ t)$, where:
  - a is an attribute name;
  - op is a comparison operator from the set $\{=, \neq, <, \leq, >, \geq\}$;

- t is an attribute value.

– $p_{Q_{HL}}$ is an (optional) projection of LDAP attributes that define the set of attributes to be returned by the query. If $p_{Q_{HL}}$ is empty, all attributes are returned. □

Example (LDAPQL Query) The LDAPQL query

$$Q_L = (\text{“cn=Queries,cn=Cache,dc=top”}, subtree, (oc = XMLQuery), \{hash\})$$

retrieves the `hash` attribute from all `XMLQuery` nodes stored under the “cn=Queries,cn=Cache, dc=top” node. □

For the purposes of this paper, it suffices to know that the *eval* predicate explained in the previous sections is implemented by means of generic algorithms that translate an arbitrary XPath expression into an LDAPQL construct and evaluates it, following the serial approach depicted in figure 1.

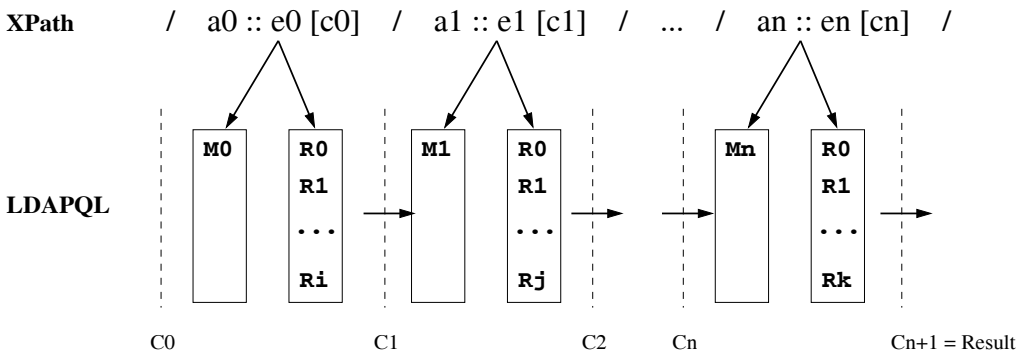

Fig. 3. XPath Evaluation

More specifically, given the nature and structure of the XPath model and of our evaluation algorithm, we can summarize the process involved in the translation and evaluation of XPath queries with the picture represented in figure 3. In it, we can see that each subquery evaluation involves the generation of two types of queries:

1. *Main queries*, depicted in the figure by queries $M_0 \ldots M_n$ at each step; and
2. *refinement queries*, represented, for example, by the $R_0 \ldots R_i$ set in the first subquery.

The output context of a specific step is uniquely determined by evaluating the set of main and refinement queries on the input context at each step of the computation. For example, C_2 is computed by evaluating M_1 and $R_0 \ldots R_j$ on C_1.

At any given path step w_i, there is one unique main query M that captures the semantics of the axis a_i and node expression e_i, and a set of refinement queries $\{R_i\}$ that correspond to each boolean predicate in c_i.

Example (LDAPQL Evaluation Example) Given the following query extracted from the Mondial database [May]: $Q_X = /child :: mondial/child :: country[attribute :: car\_code = \text{“}D\text{”}]$, the application of the evaluation algorithm, produces the following results:

– $q_0 = /child :: mondial/$
  - $C_0 = \{dn(root)\}$, since we start at the beginning of the document.
  - $w_0 = child :: mondial[\,]$
  - $C_1 = \{dn(mondial)\}$

- $q_1 = child :: country[attribute :: car\_code = \text{“}D\text{”}]$
  - $C_1 = \{dn(mondial)\}$
  - $w_1 = child :: country[attribute :: car\_code = \text{“}D\text{”}]$
  - $C_2 = \{dn(Germany)\}$, since Germany is the only country in Mondial whose `car_code` attribute has the value “`D`”.

The evaluation of the first subquery w_0, produces the following LDAP queries:

- $w_0 = child :: mondial[\,]$
  - $M_0 = (dn(root), \texttt{onelevel}, (\&(oc = \texttt{XMLElement})(name = \text{“}mondial\text{”})), \{\})$
  - $R_0 = \{\}$

Since $w_0 = child :: mondial[\,]$ has no conditional predicates, the set of refinement queries R_0 is empty.

The evaluation of the second subquery w_1, on the other hand, produces the following queries:

- $w_1 = child :: country[attribute :: car\_code = \text{“}D\text{”}]$
  - $M_1 = (dn(mondial), \texttt{onelevel}, (\&(oc = \texttt{XMLElement})(name = \text{“}country\text{”})), \{\})$
  - $R_1 = \{(dn(country), \texttt{onelevel}, (\&(oc = \texttt{XMLAttribute})(\&(name = \text{“}car\_code\text{”})(value = \text{“}D\text{”}))), \{\})\}$

In this case, the set of refinement queries is not empty because w_1 contains the predicate $attribute :: car\_code = \text{“}D\text{”}$.

The only query in R_1 is generated by our algorithms because the first term of the equality testing predicate is a path expression, and the second term a simple value (the constant “D”). □

Therefore, the evaluation of XPath queries, independently of the underlying model used for their computation (LDAPQL, DOM [ea00,HHW+00], etc.) consists of two phases that occur at every step:

1. A (possibly) *expansive* phase, represented by the set of *main queries*.
2. A (definitely) *implosive* phase, represented by the set of *refinement queries*.

4.3 Cache Data Model

The core of the cache data representation lies in the specification of the custom-defined `XMLQuery` class in the LDAP directory schema to allow for the storage of the *cache* relation.

```
XMLQuery OBJECT-CLASS ::= {
  SUBCLASS OF {top}
  MUST CONTAIN {oc,hash,context,scope,xpathquery,result}
  TYPE oc OBJECT-CLASS
  TYPE hash STRING
  TYPE context DN
  TYPE scope STRING
  TYPE xpathquery STRING
  TYPE result (DN, DN)
}
```

Fig. 4. LDAP Class for Query Representation

Figure 4 contains the complete representation of the `XMLQuery` node, where the `oc`, `hash`, `context`, `scope`, `xpathquery` and `result` attributes are stored.

The meaning of the `oc` attribute has already been defined in the previous section. It simply contains the name of the LDAP class a particular node belongs to. In our case, all nodes used to represent either a query or part of it, have a value of `XMLQuery` in their `oc` attribute.

The `hash` attribute contains the MD5 encoded string [MvOV97] that uniquely identifies a query, and is used to very efficiently determine whether or not there are any `XMLQuery` nodes in the system that contain the previously cached result for a particular set of nodes.

The next four attributes, `context`, `scope`, `xpathquery` and `result` define a query or subquery in terms of the characteristics described in the XPath specification [CD99].

The `context` attribute stores the set of nodes in the input set of the *cache* predicate, and the `result` attribute, a tuple that stores each input node with its corresponding output node. In other words, the contents of the `result` attribute is a set of distinguished name tuples (dn_i, dn_j), such that dn_j is the result of applying the query stored in the `xpathquery` attribute under the scope defined in the `scope` attribute on the node dn_i from the `context` of the query.

Finally, the `scope` and `xpathquery` attributes simply contain the human-readable form of the stored query to be used for control and debugging purposes.

4.4 Cache Evaluation Model

Combining the models we have seen so far and the fact that, in our particular implementation of the HLCaches system, we are dealing with an LDAP system, we can redefine what it means for a rewriting to be equivalent, weak equivalent or partial equivalent as follows:

Definition (HLCaches Equivalent Rewriting) Given an XPath subquery $q_i = (C_i, w_i, C_{i+1})$, an equivalent rewriting is an LDAP node n in an directory instance, such that all of the following properties hold:

1. $oc(n) =$ `XMLQuery`,
2. $C_i = context(n)$, and
3. $hash(w_i) = hash(n)$.

The functions $oc(n)$, $context(n)$ and $hash(n)$ simply return the value or values the corresponding attribute has stored at node n.

We further assume that the hash function used to create the entry in the LDAP node and the hash of the path expression w_i normalize the expression before applying the encoding so that equivalent expressions like $a \wedge b$ and $b \wedge a$ are encoded to the same string. □

The evaluation of an equivalent rewriting in HLCaches is thus performed in a very efficient way, since the result is already stored in the `XMLQuery` node.

Definition (HLCaches Equivalent Rewriting Evaluation) Let $q_i = (C_i, w_i, C_{i+1})$ be an XPath subquery, and n its equivalent rewriting for a specific directory instance. Then, the evaluation of the subquery, and therefore, the contents of C_{i+1} are $C_{i+1} = result(n)$. □

Following the same approach, we have the following definition for weak equivalent rewritings:

Definition (HLCaches Weak Equivalent Rewriting) Given an XPath subquery q_i, defined as usual $q_i = (C_i, w_i, C_{i+1})$, a weak equivalent rewriting is an LDAP node n in a directory instance, such that all of the following properties hold:

1. $oc(n) =$ `XMLQuery`,
2. $C_i \subseteq context(n)$, and
3. $hash(w_i) = hash(n)$.

The same restrictions and remarks given for the definition of equivalent rewritings hold. □

Given the new definition of weak equivalency, we need to redefine what it means to evaluate and compute the result set of a weak equivalent rewriting, since the contents of the `result` attribute in the rewriting are (possibly) a superset of the required solution.

Definition (HLCaches Weak Equivalent Rewriting Evaluation) Given an XPath subquery $q_i = (C_i, w_i, C_{i+1})$ and a weak equivalent rewriting n for a specific directory instance, the evaluation of the subquery, and therefore, the contents of C_{i+1} are computed by means of an LDAPQL query, as follows:

$$C_{i+1} = \text{LDAP}(n, \texttt{base}, \{(\&(oc = \texttt{XMLQuery})(result = (C_i, *)))\}, \{result\}) \quad \square$$

This definition assumes, as we have already mentioned, that each of the `result` entries in the rewriting are stored in the form of a tuple relation where the first tuple is the individual LDAP node evaluated, and the second one of the possibly many result nodes.

Finally, we can redefine what it means for a rewriting in HLCaches to be partial:

Definition (HLCaches Partial Equivalent Rewriting) Given an XPath subquery q_i, defined as usual $q_i = (C_i, w_i, C_{i+1})$, a partial equivalent rewriting is an LDAP node n in a directory instance, such that all of the following properties hold:

1. $oc(n) = \texttt{XMLQuery}$,
2. $C_i \supset context(n)$, and
3. $hash(w_i) = hash(n)$.

The same restrictions and remarks given for the previous definitions hold. $\square$

Therefore, and following the same mechanism used for the evaluation of weak equivalent rewritings, we have:

Definition (HLCaches Partial Equivalent Rewriting Evaluation) Given a subquery q_i, defined as usual $q_i = (C_i, w_i, C_{i+1})$, and a partial equivalent rewriting n for a specific directory instance, the partial evaluation of the subquery, and therefore, partial contents of C_{i+1} are computed by means of the following LDAP query:

$$C_{i+1} = \text{LDAP}(n, \texttt{base}, \{(\&(oc = \texttt{XMLQuery})(result = (context(n), *)))\}, \{result\}) \quad \square$$

Given the set of definitions detailed above, it remains to determine how to find equivalent rewritings in an efficient way. Following the view materialization approach, other researchers have developed rather complicated algorithms that try to achieve this goal [LMSS95,LRO96], although the efficiency of their approaches is not that impressive. The bucket algorithm [LRO96], for example, uses what could be considered a purely brute force approach after performing a quite rudimentary pruning of candidate views. Even after this pruning, the complexity of the bucket algorithm is still $O(|V| \cdot |Q|)$, where $|V|$ is the number of views in the system, and $|Q|$ the size of the query in terms of individual predicates. Furthermore, their approach is tailored exclusively for conjunctive queries, whose expressiveness is definitely a subset of that of XPath queries.

HLCaches, on the other hand, is not limited to conjunctive queries, allows the full expressive power of XPath, and is able to find equivalent rewritings very efficiently at the subquery level. Figure 5 contains pseudocode for the `FIND_EQUIVALENT` algorithm whose purpose is to look for equivalent subqueries applying the previous definitions in an efficient way.

The advantages of such an algorithm in comparison to the classical algorithms that try to find equivalent rewritings is twofold:

1. The search can be performed using just one LDAP query, and a simple subset test to remove partial equivalent rewritings. Given the nature of the MD5 encoding [MvOV97] used to implement the hash function, the number of nodes having the same hash is very limited in practice.
2. The search and evaluation of the rewriting are performed in one step, thus eliminating the need for an extra (costly) evaluation procedure.
3. The nature of LDAP instances, where information is stored in hierarchical trees, allows us to define the "cn=Queries, cn=Cache, dc=top" node to be the root of all stored queries, thus speeding up the process of finding equivalent rewritings.

Algorithm FIND_EQUIVALENT (q_i /* XPath subquery */)

Let q_i be an XPath subquery of the form $q_i = (C_i, w_i, C_{i+1})$
Let *Result* be a set of nodes that holds the result
Let t = "cn=Queries, cn=Cache, dc=top" be the cache top node

/* Perform an LDAP query to find all candidate rewritings */
$Result = \text{LDAP}(t, \texttt{subtree}, (\&(oc = \texttt{XMLQuery})(hash = hash(w_i))), \{\})$

/* Test each candidate for equivalency */
for each candidate $c \in Result$
  if (not $C_i \subseteq context(c)$)
    /* Remove non-equivalent rewritings */
    $Result = Result \setminus c$

return *Result*

Fig. 5. FIND_EQUIVALENT algorithm

For illustration purposes, the `FIND_EQUIVALENT` algorithm shown here only accepts equivalent rewritings, but it can be very easily modified to also return partial equivalent rewritings, simply by removing the extra subset checking at the end of the procedure. Such an algorithm, besides providing us with partial equivalent rewritings, also allows us to implement partial evaluation of XPath queries.

5 Conclusion

In this paper, we have provided a formal model to efficiently solve the problem of cache answerability for XPath queries when performed over XML data. The generality of our model is backed-up by the fact that it can be represented and studied independently of the storage model used for XML, but we have also provided examples and details about the implementation of such a model in the context of HLCaches, an LDAP-based distributed caching system developed by the authors for the efficient processing of XPath queries.

The efficiency of our implementation lies on the fact that the underlying representation model (LDAP) is very similar to the storage model defined by XML. Furthermore, the results and implementation details given on our LDAP-based implementation show that the checking, and evaluation of rewritings at the subquery level for XPath expressions can be performed in a very efficient way, as opposed to more classic approaches, where the efficiency of their query containment and query rewriting algorithms depend exponentially on the number of views stored in the system.

References

[BPSMM00] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0 (second edition). `http://www.w3.org/TR/2000/REC-xml-20001006`, October 2000.

[CD99] James Clark and Steve DeRose. XML path language (XPath) version 1.0. `http://www.w3c.org/TR/xpath`, November 1999.

[CGLV00] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. View-based query processing for regular path queries with inverse. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 58–66, Dallas, Texas, USA, May 2000. ACM Press.

[DFJ+96] Shaul Dar, Michael J. Franklin, Björn Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of 22th International Conference on Very Large*

Data Bases (VLDB) 1996, pages 330–341, Mumbai, Bombai, India, Septemter 1996. Morgan Kaufmann.

[ea00] L. Wood et al. Document object model (DOM) level 1 specification (2nd ed.). `http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/`, September 2000.

[Fal01] David C. Fallside. XML Schema part 0: Primer. `http://www.w3.org/TR/xmlschema-0/`, May 2001.

[HHW+00] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document object model (DOM) level 2 core specification. `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/`, November 2000.

[HSG99] T. A. Howes, M. C. Smith, and G. S. Good. *Understanding and Deploying LDAP Directory Services*. Macmillan Network Architecture and Development. Macmillan Technical Publishing U.S.A., 1999.

[KNS99] Yaron Kanza, Werner Nutt, and Yehoshua Sagiv. Queries with incomplete answers over semistructured data. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 227–236, Philadelphia, Pennsylvania, USA, May 1999. ACM Press.

[Lev00] A. Levy. Logic-based techniques in data integration. In J. Minker, editor, *Logic-Based Artificial Intelligence*. Kluwer Publishers, 2000.

[LMSS95] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 95–104, San Jose, California, USA, May 1995. ACM Press.

[LRO96] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of 22th International Conference on Very Large Data Bases (VLDB) 1996*, pages 251–262, Mumbai, Bombai, India, Septemter 1996. Morgan Kaufmann.

[Mar01] Pedro José Marrón. *Processing XML in LDAP and its Application to Caching*. PhD thesis, Universität Freiburg, October 2001.

[May] Wolfgang May. Mondial database. `http://www.informatik.uni-freiburg.de/~may/Mondial`.

[ML01] Pedro José Marrón and Georg Lausen. On processing XML in LDAP. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, pages 601–610, Rome, Italy, September 2001. Morgan Kaufmann.

[MvOV97] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[PV99] Yannis Papakonstantinou and Vasilis Vassalos. Query rewriting for semistructured data. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *Proceedings of the ACM SIGMOD International Conference 1999*, pages 455–466, Philadephia, Pennsylvania, USA, June 1999. ACM Press.

[QCR00] Luping Quan, Li Chen, and Elke A. Rundensteiner. Argos: Efficient refresh in an XQL-based web caching system. In *Proceedings of the Third International Workshop on the Web and Databases*, pages 23–28, Dallas, Texas, May 2000.

[TBMM01] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. Xml schema part 1: Structures. `http://www.w3.org/TR/xmlschema-1/`, May 2001.

[WHK97] M. Wahl, T. Howes, and S. Kille. Lightweight directory access protocol (v3). RFC 2251, December 1997.