# 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 show 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] or [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.

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. □

| **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

**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 six subqueries is depicted. The ovals inside each context between two subqueries indicate the individual XML nodes that satisfy the subquery at each point.

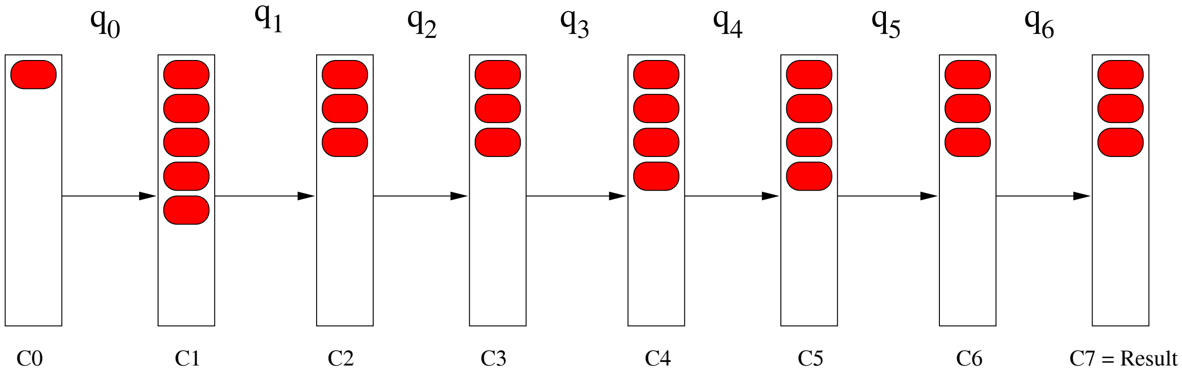

Figure 1: XPath Query Evaluation Example

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:

$$\begin{aligned} \textit{cache contents} \;&=\; \textit{cache}(C_0, \text{``}/\textit{mondial}\text{''}), \\ &\phantom{=}\;\; \textit{cache}(O_1^c, \text{``}/\textit{country}\text{''}), \\ &\phantom{=}\;\; \textit{cache}(O_2^c, \text{``}//\textit{city}\text{''}) \end{aligned}$$

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:

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

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:

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

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 and Evaluation Model:** Storage of cache contents (the *cache* predicate) in structures that allow for their efficient checking and retrieval.

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 publication.

For the purposes of our discussion regarding the implementation of our model, it suffices to know that the LDAP data model, similarly to XML, offers a tree-based representation of data that can be stored and retrieved very efficiently. The hierarchical structure of an LDAP directory is kept by means of a **distinguished name**, that we define 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 hierarchy.

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 every XPath query can be translated into LDAP queries of the form [ML01]:

**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

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

retrieves the `hash` attribute from all `XMLQuery` nodes stored under the “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.

```
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)
}
```

Figure 2: LDAP Class for Query Representation

## 4.3 Cache Data and Evaluation Model

The core of the cache data representation lies in the specification of a custom-defined LDAP class – a `XMLQuery` node – that stores the contents of the *cache* relation.

Figure 2 contains the complete representation of the `XMLQuery` node, where the `hash`, `context` and `result` attributes are stored. 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 `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.

Following this approach, by means of the `hash`, `context` and `result` attributes, the semantics of the *cache* predicate are implemented. Furthermore, thanks to the flexibility of the LDAPQL model described above, we can check for the existence of a rewriting for a given subquery $q_i = (C_i, w_i, C_{i+1})$ simply by means of the following query:

$$R = (\text{“cn=Cache,dc=top”}, \mathit{subtree}, (\&(oc = \mathit{XMLQuery})(hash = hash(w_i)), \{\})$$

where “cn=Cache,dc=top” represents the root node of the cache.

The query returns a list of possible rewriting candidates that need to be tested based on the contents of the `context` attribute following the definitions given in section 3, that is, testing whether or not the contents of the `context` attribute are either a subset or a superset of $C_i$, the input context of subquery $q_i$.

Finally, given a rewriting for a query $r$, its evaluation is simply the retrieval of the `result` attributes that correspond to the input context. This operation is performed in the following way:

$$C_{i+1} = (r, \texttt{base}, \{(\&(oc = \texttt{XMLQuery})(result = (C_i, *)))\}, \{result\})$$

for a (weak) equivalent rewriting, or

$$C_{i+1} = (r, \texttt{base}, \{(\&(oc = \texttt{XMLQuery})(result = (context(r), *)))\}, \{result\})$$

for a partial rewriting.

## 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 [Lev00].

## 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.

[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.

[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.

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