

# On Processing XML in LDAP

Pedro José Marrón

Georg Lausen

Universität Freiburg, Institut für Informatik  
Georges-Koehler-Allee, Geb. 51  
79085 Freiburg, Germany  
{pjmarron, lausen}@informatik.uni-freiburg.de

## Abstract

The proliferation of XML and its settlement as the de facto standard for information interchange in Internet, as well as the development of query languages like XPath, have lead to a paradigm transition in the Internet database world in favor of techniques based on semistructured data models. At the same time, there has been an increase in the deployment of implementations of the LDAP protocol to store the most varied information. Its data model characteristics lie closer to the semistructured model than to the relational or object model used in traditional database systems, while also offering additional benefits for the distribution, parallelization and off-site processing of queries. In this paper, we present the internal data representation and query model of an LDAP-based system used for the storage of XML documents and processing of XPath queries. The feasibility of our approach, as well as its practical relevance is backed up by experimental data.

**Keywords:** XML, XPath, LDAP, Semistructured Databases, Query Processing, Caching

---

*Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.*

**Proceedings of the 27th VLDB Conference, Roma, Italy, 2001**

## 1 Introduction

With the release of the Lightweight Directory Access Protocol (LDAP) v3 [21], the popularity of directories to store information about users, networks, etc., has steadily increased. Even commercial companies offer LDAP support in their browsers and operating systems, making directory services a viable alternative to more traditional systems for the storage and efficient retrieval of information.

At the same time, the Internet community has been moving away from static HTML, towards more dynamic and easily configurable options that allow the decoupling of content, usually represented as XML data [2], and format, usually represented as CSS data [10]. This transition has lead to an increase in the involvement of the database community in issues related to semistructured databases [1, 8], a reevaluation of semistructured data models, and even to the creation of models and mechanisms to efficiently represent and process semistructured data in relational database systems [18, 25]. These relational systems have, nevertheless, limitations due to differences in the representation and query model [19] that support the need for XML processing systems closer to its semistructured nature.

The purpose of this paper is to describe in detail the internal data representation, query model and experimental results of our XML processing system developed to prove the feasibility of general techniques to efficiently evaluate XPath queries using LDAP. Besides providing support for network operations and query distribution over a set of nodes, the LDAP model is so close to the DOM model [6], that we are able to process XML-based sources without the need to incur in cumbersome transformations, like XML to relational representations. The combination of these features, plus

the simplicity of our LDAP-based model to efficiently support XPath queries, make it the ideal candidate for an XML proxy caching system (currently under development at the U. of Freiburg). For more information, see [12, 13].

The rest of this paper is organized as follows: Section 2 presents the internal representation of data and queries, which will be used in section 3 to specify our new query model. Experimental results for the performance of our system are presented in section 4, whereas a comparison with related approaches is made in section 5. Finally, section 6 concludes this paper.

## 2 Representing XML in LDAP

The heart of the efficiency and flexibility of our system with respect to XML lies on the structure of its internal data and query representation. Since our system is based on LDAP, the internal storage model is defined in terms of LDAP classes and attributes [21]. We refer readers not familiar with the LDAP model and protocol to [7] for an informal description of its features, or to [8] for a formal one.

The main goal of our structure is to: (1) Allow for arbitrary XML documents to be stored efficiently; (2) avoid unnecessary changes in the LDAP schema that slow down the storage process; and (3) allow for efficient query processing, partitioning and forwarding.

Based on these requirements, we have designed several representation models that, despite their simplicity, provide an elegant and uniform environment for effective processing of XPath queries in our system. The following sections go into detail about each one of the internal representations: XML documents, in section 2.1; and XPath queries, in section 2.2.

### 2.1 Data Representation

The most obvious way to represent XML documents in LDAP is to provide two mappings: one between the DTDs and the schema definition to provide the structure, and another one between the XML document and the LDAP instance level. This intuitive but naive approach has several disadvantages: (1) The integration of a DTD in LDAP involves changes to the schema that, by definition, is assumed to be static in LDAP [21]; (2) even in the case where extensions to the basic LDAP model to allow dynamic schema changes are used [24], their update is troublesome and can introduce illegal nodes at the instance level; and (3) the DTD for most XML documents is not available, reduc-

ing the number of processable XML documents considerably.

Therefore, we have developed our internal representation in such a way to avoid these limitations without losing the ability to incorporate arbitrary XML documents in our system. Figure 1 describes the contents of the `XMLNode`, `XMLElement` and `XMLAttribute` classes that constitute the basic elements of our representation.

```
XMLNode OBJECT-CLASS ::= {
  SUBCLASS OF {top}
  MUST CONTAIN {oc,oid,name}
  TYPE oc OBJECT-CLASS
  TYPE oid DN
  TYPE name STRING}
XMLElement OBJECT-CLASS ::= {
  SUBCLASS OF {XMLNode}
  MUST CONTAIN {order}
  MAY CONTAIN {value}
  TYPE order INTEGER
  TYPE value STRING}
XMLAttribute OBJECT-CLASS ::= {
  SUBCLASS OF {XMLNode}
  MUST CONTAIN {value}
  TYPE value DN, STRING}
```

Figure 1: XML Data Representation in LDAP

The generality of our model to encode arbitrary XML documents is based on two key factors: (1) The use of the attributes defined in figure 1 to store information about individual XML nodes; and (2) the hierarchical nature at the instance level of the LDAP model, where each node must be assigned a distinguished name that defines its unique location in the hierarchy. Of all the LDAP attributes defined in the figure, `name`, `order` and `value` are needed to explicitly represent XML documents, whereas the `oc` and `oid` attributes are intrinsic to the LDAP model.

```
<country car_code="D", area="356910",
        capital="Berlin">
  <name>Germany</name>
  <population>83536115</population>
  <languages percentage="100">
    German</languages>
  <province id="B-W", capital="cid-9",
            country="D">
    ...
  </province>
</country>
```

Figure 2: Excerpt from a Mondial XML file

In order to illustrate the use of the LDAP

classes in figure 1, we have taken as an example (figure 2) an excerpt of the Mondial database available in XML format from [14]. In it, we can see that the element `country` has attributes `car_code`, `area` and `capital` and several subelements (`name`, `population`, etc), some of which have in turn, attributes and/or textual contents, like `Germany` or `83536115`.

For each element in the XML document, we create an `XMLElement` node where the LDAP attributes `name`, `order`, `value` and `oid` from figure 1 are set to their corresponding values. For example, the `languages` tag in the document would create an `XMLElement` with `name = languages`; `order = 4`, since it is the fourth node in global document order; `value = German`, because the textual content of the node is the string “German” and `oid = (cn=4, cn=1)`, which contains information in the form of an LDAP distinguished name to determine the location of this node in the document hierarchy. The `oc` attribute, which represents the LDAP class the node belongs to, is initialized to `oc = XMLElement`.

The attribute `percentage` from the `languages` node is processed in a similar way. We create an `XMLAttribute` node, where the LDAP attributes are initialized as follows: `name = percentage`, `value = 100` and `oid = (cn=5, cn=4, cn=1)`. We follow the XPath convention that attribute nodes are stored as children of their corresponding element, and therefore, the `oid` LDAP attribute is initialized to represent this fact. Finally, the `oc` attribute is set to `oc = XMLAttribute`.

Figure 3 contains the graphical representation of the LDAP tree generated as a result of applying the transformation algorithm we have just described. An algorithmic representation of this process is described in the XML2LDAP procedure in the Appendix. Its inverse algorithm (LDAP2XML), not shown in the paper due to space constraints, performs the translation in the opposite direction and allows, therefore, the lossless retrieval of XML documents from the LDAP instance directory.

## 2.2 Query Representation

In addition to providing a data representation model, we also need to be able to represent queries to determine whether or not a particular query has already been stored and/or is answerable using only the data obtained from previously evaluated queries. This model is of extreme importance if our system is to be used as a proxy cache for XML documents, since it

is the basis for the solution to the problem of cache answerability.

Following the same schema from the previous section, we have extended the set of standard LDAP classes to include an `XMLQuery` class that contains all relevant information about a particular query and its result set.

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

Figure 4: Query Representation in LDAP

The detailed list of attributes that constitute an `XMLQuery` is shown in figure 4. The meaning of the `oc` attribute is analogous to that of the data representation classes previously defined. All nodes used to represent either a query or part of it, have a value equal to `XMLQuery` in their `oc` attribute.

The `hash` attribute contains an encoded string that uniquely identifies a query. The purpose of this attribute is to serve as a key that can be used among different evaluation nodes in our system to exchange information about the queries already processed and stored at the node. This caching mechanism helps in providing a more efficient query processing system, as we will see later on.

The next four attributes, `context`, `scope`, `xpathquery` and `result` define a query or subquery in terms of the characteristics described in the XPath specification [5].

The `context` attribute is a set of distinguished names defined as the result set of a (possibly non-existing) previous subquery. The contents of the `result` attribute is the set of distinguished names that contain the LDAP nodes resulting from applying the query stored in the `xpathquery` attribute under the `scope` defined in the `scope` attribute on the `context` of the query.

By means of these four attributes, our system is able to provide support for subquery rewriting, remote query processing, cache answerability, and other features that will be dealt with in detail in the next section.

## 3 Query Model

The query model used by our system is very close to the traditional LDAP query model de-

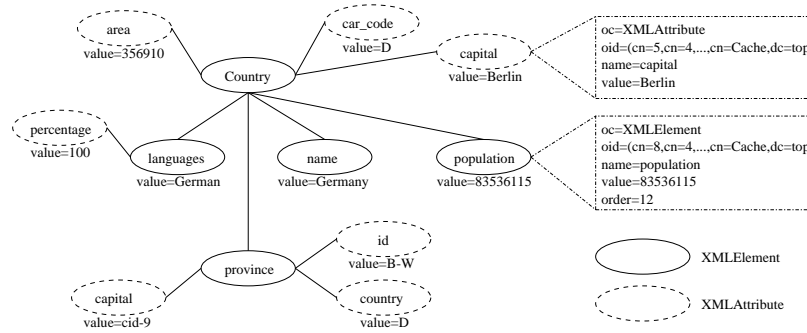


Figure 3: XML Document in LDAP

scribed in the standard specification [21] and in other pieces of the literature [7, 8]. However, our model differs from previous approaches in our desire to limit to a minimum the number of changes to the standard LDAP model so that it can be deployed easily in existing systems, while at the same time providing full XPath [5] support. Other papers [8] provide extensions to the LDAP model that, although very interesting and valid, either go beyond the requirements of XPath, or need considerably more work than our model in order to be deployed in current systems.

By allowing full XPath expressive power, we are able to support not only XML but also a whole breadth of semistructured models whose querying languages are based on variations and extensions of the same idea exploited by XPath.

### 3.1 Traditional LDAP Query Model

In the traditional LDAP model a query is defined in the following way:

**Definition (LDAP Query)** An LDAP query  $Q_L = (b_{Q_L}, s_{Q_L}, f_{Q_L}, p_{Q_L})$  is a 4-tuple filter-based expression where:  $b_{Q_L}$  is the distinguished name of the base entry in the directory instance where the search starts from;  $s_{Q_L}$  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; or `subtree`, if all nodes under the base should be considered by the filter expression.  $f_{Q_L}$  is the filter expression defined as the boolean combination (written in prefix notation) of atomic filters of the form  $(a \text{ op } t)$ , where  $a$  is an attribute name;  $op$  is a comparison operator from the set  $\{=, \neq, <, \leq, >, \geq\}$ ; and  $t$  is an attribute value. Finally,  $p_{Q_L}$  is an (optional) projection of LDAP attributes that define the set of attributes to be returned by the query. If  $p_{Q_L}$

is empty, all attributes are returned.

**Example** The LDAP query  $Q_L = ("cn = Cache,dc = top", subtree, (oc = XMLQuery), \{hash\})$  retrieves the `hash` attribute from all `XMLQuery` nodes under the `cn=Cache,dc=top` node, possibly to report our own cache contents to other nodes in the distributed caching system.

### 3.2 XPath Query Model

As specified in [5], the primary purpose of the XPath standard is to address parts of an XML document, usually represented in the form of a tree containing 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:  $Q_X = q_0/q_1/\dots/q_n$ , where  $q_i$  is an XPath subquery defined below, and `'/'` is 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); and  $C_{i+1}$  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 like `ancestor`, `child`, `sibling`, etc. (see table 1

$n$	$a_i$	$b_{Q_{HL}}$	$s_{Q_{HL}}$	$t_f$
n	ancestor	n	ancestors	oc=XMLElement
n	ancestor-or-self	n	{ancestors,base}	oc=XMLElement
n	attribute	n	onelevel	oc=XMLAttribute
n	child	n	onelevel	oc=XMLElement
n	descendant	n	subtree	oc=XMLElement
n	descendant-or-self	n	{subtree,base}	oc=XMLElement
n	following	root(n)	subtree	(&(oc=XMLElement)(order>order(n)))
n	following-sibling	parent(n)	onelevel	(&(oc=XMLElement)(order>order(n)))
n	parent	n	parent	oc=XMLElement
n	preceding	root(n)	subtree	(&(oc=XMLElement)(order<order(n)))
n	preceding-sibling	parent(n)	onelevel	(&(oc=XMLElement)(order<order(n)))
n	self	n	base	oc=XMLElement

Table 1: BaseScope: Axis Translation Function

for a complete list);  $e_i$  is a node expression that tests either the name of the node or its content type; and  $c_i$  is a boolean expression of conditional predicates that must be fulfilled by all nodes along the path.

**Example** The XPath query:  $Q_X = /child :: mondial/child :: country[attribute :: car\_code = "D"]$  is composed of two subqueries whose combination selects all `country` nodes directly connected to the `mondial` node having an attribute `car_code` with value "D".

### 3.3 LDAPQL Model

In our system, we have extended the traditional LDAP query model described in section 3.1 to include the necessary modifications to support XPath queries.

The gist of the modifications rely on the fact that LDAP only allows searching of the document tree in one direction, from the root to the leaves, whereas XPath contains axis that perform queries in the "upwards" direction.

Therefore, we need to extend the notion of scope in the traditional model to include parents, siblings and ancestors, as defined below.

**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: (1)  $b_{Q_{HL}}$ ,  $f_{Q_{HL}}$  and  $p_{Q_{HL}}$  are defined as in the LDAP Model; and (2)  $s_{Q_{HL}}$  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; `parent`, if only the parent node should be considered; `siblings`, if only the siblings of the node should be considered; and

`ancestors`, if all the ancestors of the node up to the root are to be searched.

Using this new model, we are able to transform any XPath query into a series of LDAPQL queries that achieve the same goal as the original query, as we detail in the next section.

### 3.4 Evaluation of XPath Queries

Given the nature and structure of the XPath model described above, in order to evaluate an XPath query, we compute each input and output context  $C_i$  by means of two types of LDAPQL queries: (1) main queries (M) and (2) refinement queries (R). Figure 5 contains a graphical representation of our XPath2LDAPQL evaluation algorithm (see Appendix), where each XPath subquery is translated in one main query and a set of refinement queries, that uniquely determine the input context for the next step of the computation. Table 1 contains the representation of the scope translation function used by the algorithms in the Appendix as a base template for the main and refinement queries.

The number of refinement queries in each step equals the number of predicates in the conditional expression of the subquery. At the end of the evaluation, the final context  $C_n$  contains the result of the query.

**Theorem 1 (Result Equivalence)** *Given an XML document, the evaluation of an XPath query produces the same result as the evaluation of the equivalent LDAPQL query generated by our XPath2LDAPQL algorithm and performed over the XMLDAP document that results from the application of our XML2LDAP algorithm to that same XML document (figure 6).*

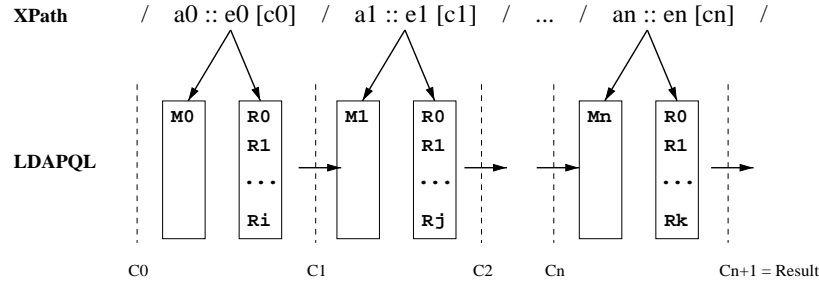


Figure 5: XPath Evaluation

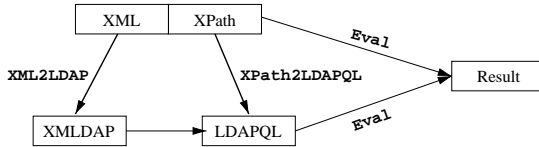


Figure 6: Result Equivalence

**Example** The application of our evaluation algorithm to the XPath query  $Q_X$  from section 3.2 produces the following results:

- $q_0 = /child :: mondial/$ 
  - $C_0 = \{dn(root)\}$ , since we start at the beginning of the document.
  - $M_0 = (dn(root), onelevel, (\&(oc = XML\text{Element})(name = "mondial")), \{\})$
  - $R_0 = \{\}$
  - $C_1 = \{dn(mondial)\}$
- $q_1 = child :: country[attr :: car\_code = "D"]$ 
  - $C_1 = \{dn(mondial)\}$
  - $M_1 = (dn(mondial), onelevel, (\&(oc = XML\text{Element})(name = "country")), \{\})$
  - $R_0 = \{(dn(country), onelevel, (\&(oc = XML\text{Attribute})(\&(name = "car\_code")(value = "D"))), \{\})\}$
  - $C_2 = \{dn(Germany)\}$ , since Germany is the only country in figure 2 whose `car_code` attribute has the value "D".

### 3.5 Advantages of our model

The combination of the internal data representation described in section 2 with the XPath mapping algorithm we have just proposed, offer quite a few advantages that make them suitable for deployment in a distributed query processing or caching environment.

Since an XPath query can be naturally split in a sequence of subqueries, the problem of **cache answerability** becomes an easier one. By using the `context` and `result` attributes, intermediate evaluation results of a query can

be cached independently of their superordinate query, allowing for the development of cache answerability algorithms that work at the subquery level. Since the structure of subqueries, as well as their evaluation process is simpler than that of complete queries, more efficient algorithms that check for containment and rewriting possibilities can be implemented.

Another further benefit of working with XPath queries at the subquery level is the possibility to perform **partial query evaluations**, or to evaluate a query until a specific intermediate result has been achieved. This provides the basis for efficient evaluation algorithms over user queries that rely on previous results or a specific context. It is well known that queries tend to be localized, and that small variations of the same query appear together in time, as a result of users refining, redefining, or extending their queries.

Thanks to our evaluation procedure, we are able to find context matches not only at the beginning, or the end of a query, but more importantly, also within the inner subqueries. This is of extreme importance for queries performed by means of a compiler or graphical interface that translates higher level language queries into XPath, since the same trends, patterns and contexts are used over and over again.

By finding partial matches among the subqueries that compose a request, we are able to do **subquery preprocessing** following a partial match in the same way a CPU does branch prediction, but, as opposed to a CPU, we can benefit from partial subquery results, even in the case where we "mispredicted", or did not have enough information to process the whole subquery. As the evaluation routine trickles down, its result is combined with the precomputed subquery, eventually leading to the correct result. The implementation of this procedure involves an almost trivial change to the XPath2LDAPQL algorithm, where the computation loop for context  $C_i$  is unrolled and paral-

lelized based on the (partially) known  $C_{i-1}$ .

The fact that query evaluation could potentially be performed in **parallel** due to the detection of partial matches in user requests is the most important benefit of our approach that make it suitable for a distributed environment. This evaluation can be made in parallel either on the same machine that received the original query, or in other machines that might be less overloaded at the moment. Since each one of our caches may keep a distributed index of the contents of all other caches in the system, it can decide to send a partial evaluation request to another peer that happens to already have an answer to part of the query, making the evaluation process much more efficient.

## 4 Experimental Results

In order to prove the feasibility of our system as an efficient LDAP-based query processing system for XML, we have performed a series of experiments to determine the following characteristics: average storage and retrieval time; and query execution performance improvement.

Preliminary experiments that involve more than one server and make use of the parallelization capabilities of XPath queries in our system, although promising, have been left out of this work due to lack of space.

### 4.1 Experimental Setup

All experiments have been performed on a Pentium III 450 MHz based computer running Red Hat Linux 6.1, a modified version of the Apache Server v.1.3.12 and a heavily modified implementation of the OpenLDAP Server v.1.2.10. Modifications to Apache involve the integration of an XML specific cache into the `mod_proxy` module, whereas the changes performed on the OpenLDAP server focused on the implementation of the internal representation described in section 2, and a series of modifications to the query processing and network communication module. These last changes, which also involve the addition of a thread pool per connection to the server, have been responsible for an increase in performance of over two orders of magnitude with respect to the original OpenLDAP implementation when performing a rapid sequence of LDAP operations, which is crucial for our translation and evaluation algorithms.

The data files used in our benchmarking experiments come from several sources: the Mondial database [14], that contains geopolitical information about countries, organizations, etc.;

XSLBench [9], a performance benchmark for XSLT [4] maintained by Kevin Jones; the ACM Sigmod Record Database in XML form; and a database of "Great Books" maintained in WML at JollyRogers, which serves as an example of the use of our system with WAP technology, since WML is structurally, nothing but a set of XML documents that conform to the WML Document Type Definition [22].

### 4.2 Average Storage and Retrieval Time

In this experiment we have measured both, the average storage time needed by the XML2LDAP algorithm to incorporate an XML document in our system, and the average retrieval time of our LDAP2XML algorithm used to reconstruct the XML document resulting from a query. For the retrieval experiments, we have computed a worst case scenario, where the whole document must be retrieved, formatted and returned to the user. For the most part, queries request only a subset of nodes in the document, making the data in table 2 a lower bound on the efficiency of our algorithms.

As shown in the table, our system can process almost 4700 store operations per second, which correspond to about 2700 XML nodes, where each node is either an element or an attribute. For the retrieval experiments, 650 operations can be performed per second, which amount to almost 400 XML nodes. The performance of storage operations is so good in comparison to the retrieval operations because the former are performed asynchronously, whereas the latter must be performed in a synchronous fashion, since the order in which nodes are received is an important factor for the reconstruction of the original document. Despite the relative performance disadvantage of read operations overall, the fact that this number is a lower bound on the efficiency of our procedure, and that clients start receiving the document as soon as the first bytes are generated, imply that there is no noticeable overhead for read operations seen from the perspective of the client.

### 4.3 Query Execution Performance Improvement

In the last set of experiments, we have tried to determine the relative performance gain of our query mechanism with respect to similar XPath engines by the proper use of LDAP filters and translation mechanisms detailed in section 3.

For the following set of queries, we have taken a C-based implementation of an XPath

File Name	Ops	Stor.(s)	Ops/sec.	Retr.(s)	Ops/sec.
mondial-2.0.xml	57116	13.34	4281.56	85.86	665.22
europa-2.0.xml	18186	3.88	4687.11	26.84	677.57
dream.xml	6231	1.19	5236.13	10.22	609.69
SigmoidRecord.xml	38518	8.43	4569.16	56.33	683.79
books1.wml	138	0.0098	14081.63	0.18	766.66
<b>Average</b>	–	–	4693.50	–	659.07

Table 2: Average Storage and Retrieval Time

engine [20] developed for the Gnome project by Daniel Veillard. At the point where we started performing our experiments, this engine was the only open source XPath implementation we could find written in C that could be compared to our system (also written in C). We modified his original version of the parser to use our LDAP-based system instead of the DOM backend required for XPath processing, and compared the results of using the original DOM and our LDAP-based representation.

Table 3 shows some of the most representative XPath queries involving simple walk-throughs and predicate testing, and their processing times in seconds using the DOM backend and our LDAP-based system. The queries were performed on a “clean” system loaded only with the `mondial-2.0.xml` file used in previous experiments. As can be seen in the table, our system performs better than the DOM representation in all cases where the query requires more than a mere name lookup. The use of filters to reduce the search space of possible answers as early as possible has a tremendous effect on the performance of our system, not only on simple queries, like the ones in table 3, but more importantly on all complex queries that make use of these simple patterns as their building blocks. Furthermore, the easiness of integration of our system in a distributed environment, like the Internet, enables the evaluation of such queries in parallel, as described in detail in section 3.5.

## 5 Related Work

Although the field of query procedures and optimizations for XML have been extensively researched [3, 15], none of these approaches tries to combine the similar representation models of LDAP and XML to benefit from the distribution capabilities inherent in the LDAP model. In our approach, besides providing evidence that both representation models can work together, we try to benefit from the best of both worlds by taking advantage of the distribution

capabilities of LDAP.

Due to the document and query storage capabilities of our system, our approach could be used as a caching system with additional querying capabilities that falls on the realm of active caching systems like [11]. However, the usual focus of active caching systems lie on applications and applets either in Java or Javascript, whereas our system focuses on semistructured data and how to efficiently query it.

Other approaches, like the one taken by [25] rely on content specification of a Web site to implement their caching strategies. However, this approach lacks the flexibility and applicability of our system, which does not even rely on the presence of DTDs to perform its task. ARGOS [17], on the other hand, takes a similar approach to ours, but its focus is on the integration of materialized views for semistructured data, that, although feasible as a caching solution, lacks the underlying structure to be deployed in a distributed environment with more than one integration point. Furthermore, they rely on a Java-based PDOM implementation that cannot be implemented in a setting where efficiency is a key issue.

Finally, the distributed indexing mechanism implemented in our caching framework relate to the notion of active catalogs in *Nomenclator* [16], where, interestingly enough, the focus was on X.500 directories [23], the precursor to LDAP. Our system differs from *Nomenclator* in its widest breadth of features, its applicability to caching, and its extensibility regarding the maintenance of distributed indexing capabilities over semistructured databases.

## 6 Conclusion

In this paper we have presented the design, internal data representation details and query model of an efficient, LDAP-based system for the processing of XPath queries on XML documents in a network environment.

We have introduced evaluation algorithms that allow our system to provide full support



Query Patterns	Nr. Result Nodes	DOM back-end	LDAPQL
/mondial/country	260	0.69	0.71
/mondial//city	3047	217.67	91.40
/mondial/country[@car_code='D']	1	6.36	4.68
/mondial//city[@is_cap='yes']	230	276.56	116.03

Table 3: Performance for XPath using DOM and HLCaches

for XPath queries, as well as a storage model based on subquery processing that enables the seamless integration of distributed and parallel query execution.

We have further provided experimental data that characterizes the average access and query processing times of our system in comparison to other DOM-based implementations. Although our approach incurs in some additional overhead, mainly in terms of storage space, and the translation to and from LDAP, its advantages outweigh this factor, especially when dealing with applications that use small XML files as their information basis, as it is the case with wireless applications and WML.

## References

- [1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web*. Morgan Kaufmann, 2000.
- [2] T. Bray, J. Paoli, C.M. Sperberg-McQueen, and E. Maler. Extensible markup language (xml) 1.0. <http://www.w3.org/TR/2000/REC-xml-20001006>, October 2000.
- [3] V. Christophides, S. Cluet, and J. Siméon. On wrapping query languages and efficient XML integration. In *Proc. of the SIGMOD Intl. Conf. on Management of Data*, 2000.
- [4] J. Clark. XSL transformations 1.0. <http://www.w3.org/TR/xslt>, 1999.
- [5] J. Clark and S. DeRose. XML path language 1.0. <http://www.w3c.org/tr/xpath>, 1999.
- [6] L. Wood et al. Document object model. <http://www.w3.org/TR/REC-DOM-Level-1>, 2000.
- [7] T. A. Howes, M. C. Smith, and G. S. Good. *Understanding and Deploying LDAP Directory Services*. Macmillan Technical Publishing U.S.A., 1999.
- [8] H. V. Jagadish, L. V. S. Lakshmanan, T. Milo, D. Srivastava, and D. Vista. Querying network directories. In *Proc. SIGMOD Intl. Conf. on Management of Data*, 1999.
- [9] K. Jones. XSLBench. [www.tfi-technology.com/xml/xslbench.html](http://www.tfi-technology.com/xml/xslbench.html), 2000.
- [10] H.W. Lie and B. Bos. Cascading style sheets. <http://www.w3.org/TR/REC-CSS1>, 1999.
- [11] Q. Luo, J.F. Naughton, R. Krishnamurthy, P. Cao, and Y. Li. Active query caching for database web servers. In *Proc. Third Intl. Workshop on the Web and Databases*, 2000.
- [12] P.J. Marrón. Design of a middleware service for scalable wide area network applications. In *Proc. Intl. Symp. on Distributed Objects and Applications*, 2000.
- [13] P.J. Marrón and G. Lausen. HLCaches: An LDAP-based distributed cache technology for XML. Technical report, University of Freiburg, 2001.
- [14] W. May. Mondial DB. [www.informatik.uni-freiburg.de/~may/Mondial](http://www.informatik.uni-freiburg.de/~may/Mondial).
- [15] J. McHugh and J. Widom. Query optimization for XML. In *Proc. 25th Intl. Conf. on Very Large Data Bases*, 1999.
- [16] J. J. Ordille and B. P. Miller. Distributed active catalogs and meta-data caching in descriptive name services. In *Proc. 13th Intl. Conf. on Distr. Computing Systems*, 1993.
- [17] L. Quan, Li Chen, and E.A. Rundensteiner. Argos: Efficient refresh in an XQL-based web caching system. In *Proc. Third Intl. Workshop on the Web and Databases*, 2000.
- [18] A. Schmidt, M.L. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *Proc. 3rd Intl. Workshop on the Web and Databases*, 2000.
- [19] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and oportunities. In *Proc. 25th VLDB Conf.*, 1999.
- [20] D. Veillard. <http://www.xmlsoft.org/>.
- [21] M. Wahl, T. Howes, and S. Kille. LDAP (v3). RFC 2251, 1997.
- [22] <http://www.oasis-open.org/cover/wap-wml.html>.
- [23] Information processing systems - open systems interconnection - the directory: Overview of concepts, models and service. ISO/IEC JTC 1/SC21, International Standard 9594-1, 1988.
- [24] Y. Yaacovi, M. Wahl, and T. Genovese. LDAP (v3): Extensions for dynamic directory services. RFC 2589, 1999.
- [25] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching strategies for data-intensive web sites. In *Proc. of the 26th VLDB Conf.*, 2000.

## 7 Appendix: Evaluation Algorithms

**Algorithm** XML2LDAP ( $D$  /\* XML Document \*/)

```

/* Initialize top of LDAP cache */
CurrentNode = "(cn=Cache,dc=top)"
while there is input  $i$  from  $D$ 
  /* If  $i$  is an opening tag, create the Element and link it */
  if  $i$  is  $< \text{tagName attrName}_0 = \text{attrValue}_0 \dots \text{attrName}_n = \text{attrValue}_n >$ 
    NewNode = XMLElement(tagName)
    link(CurrentNode, NewNode)
    CurrentNode = NewNode
    /* Create and link attributes */
    for each attrName, attrValue pairs
      link(NewNode, XMLAttribute(attrName, attrValue))
  /* If  $i$  is a closing tag */
  if  $i$  is  $< /\text{tagName} >$ 
    CurrentNode = Parent(CurrentNode)
  /* else,  $i$  is the textual content of the node */
  else
    CurrentNode.value =  $i$ 

```

**Algorithm** XPath2LDAPQL ( $Q_X$  /\* XPath query \*/)

```

/* Initialize  $C_0$  to the cache root */
 $C_0 = "(cn=Cache,dc=top)"$ 
For each subquery  $q_i = (C_i, w_i, C_{i+1}) \in Q_X$ 
  /* Create a new XMLQuery node and initialize its attributes */
  XMLQuery.context =  $C_i$ 
  XMLQuery.xpathquery =  $w_i$ 
  XMLQuery.hash = hash( $w_i$ )
  /* For each node in the context, evaluate  $w_i$  on it */
  for each  $n \in C_i$ 
     $C_{i+1} = C_{i+1} \cup \text{EVAL}(\text{PET}(n, w_i))$ 
  XMLQuery.result =  $C_{i+1}$ 

```

**Algorithm** EVAL ( $M$  /\* Main query \*/,  $R$  /\* Set of subordinate queries \*/)

```

Result = LDAP( $M$ )
for each subquery  $R_i \in R$ 
  Result = Result  $\cap$  LDAP( $R_i$ )
return Result

```

**Algorithm** PET( $n$  /\* distinguished name \*/,  $w_i$  /\* an expression of the form  $a_i :: e_i[c_i]$  \*/)

```

Let  $M_{HL}$  be an LDAPQL query (called main query)
Let  $R = \{R_j\}$  be a set of LDAPQL queries (refinement)
/* Translate  $a_i$  into  $M_{HL} = (b_{M_{HL}}, s_{M_{HL}}, f_{M_{HL}}, p_{M_{HL}})$  */
( $b_{M_{HL}}, s_{M_{HL}}, f_{M_{HL}}$ ) = BaseScope( $n, a_i$ )
for each  $nodeName = e_i$ 
   $f_{M_{HL}} = (\&(f_{M_{HL}})(\text{name} = nodeName))$ 
 $p_{M_{HL}} = \{\}$ 
/* Translate each predicate  $cp_j$  into  $R_j = (b_{R_j}, s_{R_j}, f_{R_j}, p_{R_j})$  */
Let  $R = \{\}$ 
for each  $cp_j \in c_i$ 
  Let  $cp_j$  be of the form  $term_j op_j value_j$ 
  ( $b_{R_j}, s_{R_j}, f_{R_j}$ ) = BaseScope(LDAP( $M_{HL}$ ),  $term_j$ )
  for each ( $nodeName, nodeValue$ )  $\in c_i$ 
     $f_{R_j} = (\&(f_{R_j})(\&(\text{name} = nodeName)(\text{value } op_j \text{ nodeValue})))$ 
     $p_{R_j} = \{\}$ 
   $R = R \cup R_j$ 
Return ( $M_{HL}, R$ )

```