HLCaches: An LDAP-based Distributed Cache Technology for XML

Pedro José Marrón and Georg Lausen
Institut für Informatik
Georges-Koehler-Allee, Geb. 51
79085 Freiburg, Germany
{pjmarron, lausen}@informatik.uni-freiburg.de

March 5, 2001

Abstract

The proliferation of XML and its settlement as the de facto standard for information interchange in Internet, as well as the development of querying languages for XML, like XPath, have lead to a paradigm transition in the Internet database world, where the use of classical database systems has decreased in favor of techniques and standards based on semistructured data models, from which XML is the most prominent example. At the same time, there has been an increase in the support, development and deployment of implementations of the LDAP protocol to store the most varied information. Its data model characteristics lie closer to the semistructured model as to the relational or object model used in traditional database systems, and therefore, its use as a caching system for semistructured databases is fully justified. In this paper, we present the design, internal data representation and query model of `HLCaches`, a hierarchical distributed caching system for semistructured documents based on LDAP technology that brings both, the semistructured data model and the LDAP data model together into a system that provides the ideal characteristics for the efficient processing of XPath queries over XML documents. We also show transformation algorithms and experimental results that prove the feasibility of `HLCaches` as a distributed caching system especially tailored for semistructured data.

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

1 Introduction

With the release of the Lightweight Directory Access Protocol (LDAP) version 3 in 1997 [23], the popularity of directories to store information about users, networks, etc. has been steadily increasing. Even companies like Netscape or Microsoft offer LDAP support in their browsers and operating systems, making directory services a viable alternative to more traditional database systems for the storage and efficient retrieval of information.

At the same time, the Internet community has been moving away from static HTML to describe information on the web, towards more dynamic and easily configurable options that allow the decoupling of content, usually represented in form of XML data [3], and format, usually represented as CSS data [17]. This transition has lead to an increase in the involvement of the database community in issues related to semistructured databases [15], 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 [20, 28]. These relational systems have, nevertheless, limitations due to differences in the representation and query model [21] that support the need for XML processing systems closer to its semistructured nature.

The purpose of this paper is to describe in detail the architectural design, query model and experimental results of HLCaches, a hierarchical distributed caching system based on LDAP technology developed to:

- Improve the performance of data retrieval of sites based on XML technology.
- Improve the current state of caching technology on the Internet by providing a system that supports both older and newer technologies.
- Serve as an example of a real world application on top of Plexor, a generic middleware framework for distributed applications [13].

The rest of this paper is organized as follows. Section 2 provides a detailed description of the overall architecture of our system. Section 3 presents the internal representation of data, queries and meta-data in HLCaches, which will be used in section 4 to specify our new query model. Experimental results for the performance of our system are presented in section 5, whereas a comparison with other related systems is made in section 6. Section 7 deals with our future plans, and finally, section 8 concludes this paper.

2 Overall Architecture

HLCaches is a hierarchical, LDAP-based, distributed caching system that allows the storage and efficient retrieval of static web information, like HTML pages, images, etc., as well as XML documents in a common cache environment. HLCaches has been implemented on top of Plexor, a middleware software that provides support for the distribution of data and group membership management needed by HLCaches.

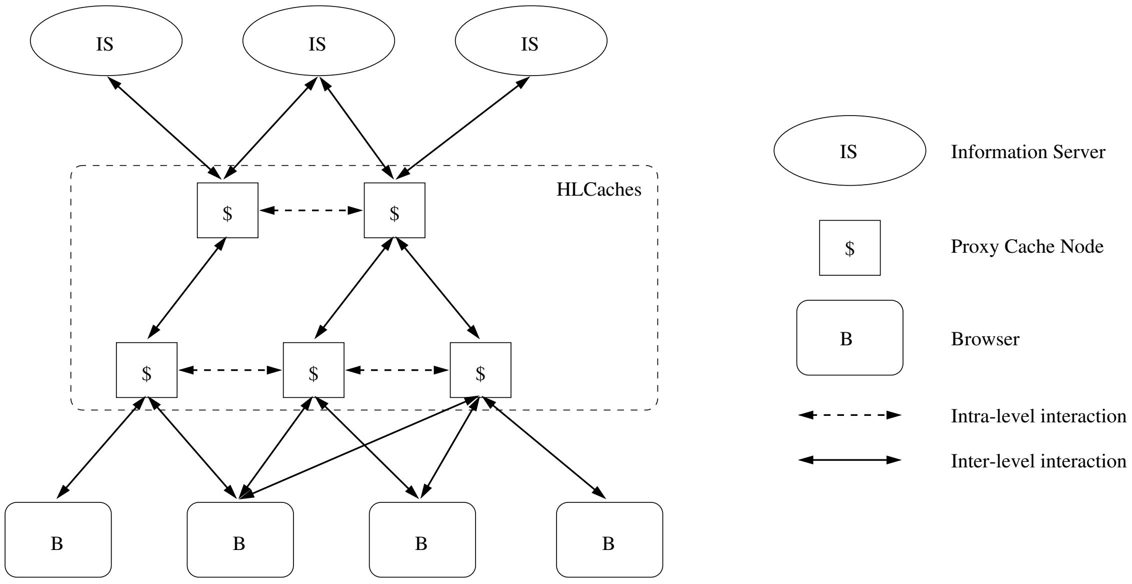

Figure 1: General Architecture

Figure 1 depicts a group of four browsers (B) retrieving information from three different information systems (IS) through our HLCaches system, composed, in this particular example, of two levels of proxy caching servers that exchange information both, at the inter-level and the intra-level domains.

The configuration of the individual proxy cache nodes in the HLCaches system, as well as their belonging to a particular level in the hierarchy are managed by the underlying Plexor system, which also provides facilities for scalability, replication, fault-tolerance and the transparent addition or removal of nodes in the system. A full description of the Plexor system is out of the scope of this paper, but more information can be obtained from [13].

Internally, each one of the proxy cache nodes in figure 1 is made up of the following components, schematically represented in figure 2:

- Proxy Web Server (part of the Apache server)
- Query Engine
- Distributed Cache Engine
- XML Parser
- XMLDAP Cache (OpenLDAP Server)
- Traditional Cache (also part of the Apache server)

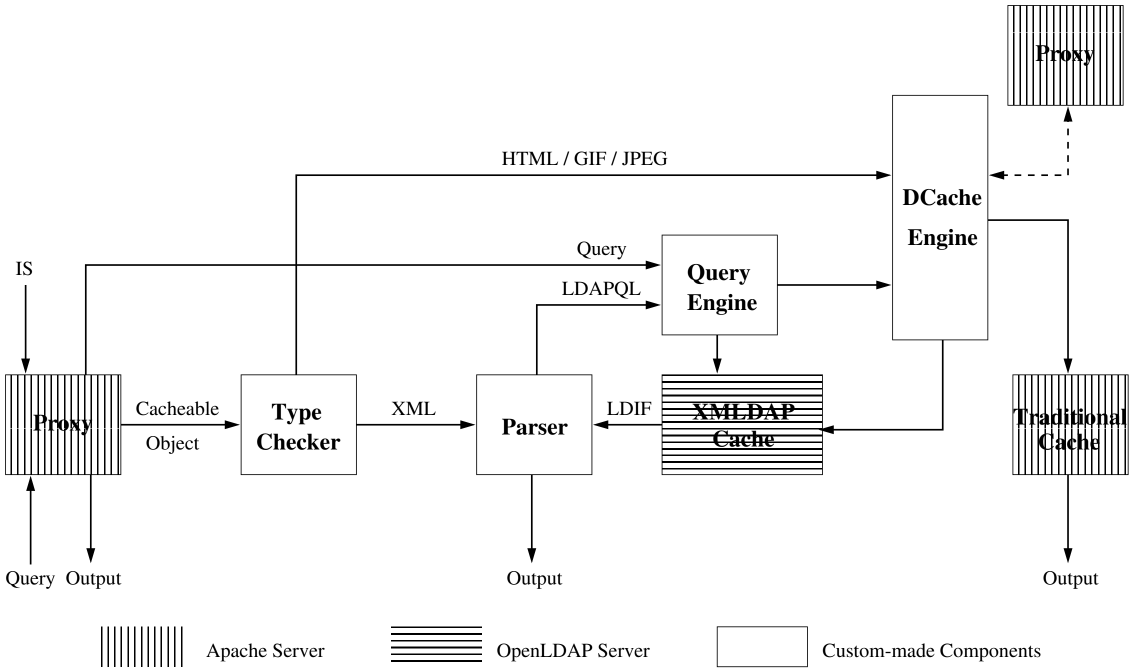

Figure 2: Proxy Cache Server Internal Architecture

A modified version of the popular Apache Server [10] is heavily used in our system to provide the functionality of a caching proxy server that forwards user requests as needed, either to the appropriate node in the hierarchy or directly to the information system after the corresponding processing has been done by the node.

As it is obvious from figure 2, the proxy engine serves as a front-end that collects queries in the form of URL requests, and transmits the result (output) back to the client. Whether or not the node has the necessary information in either one of its caches to provide the answer without incurring in the overhead of contacting other nodes is determined by the **Query Engine** and the **Distributed Cache Engine**.

If, as it may be the case, the node does not have the necessary information to answer the request by itself, the query is forwarded to the next level in the hierarchy, and the answer cached in the node as it is sent back to the client. Whether the object should be cached in the **Traditional Cache** or in the **XMLDAP Cache** is determined by the **Type Checker**.

In the current modified version of the server, functionality specifically designed for our system can be turned on or off via the standard `httpd.conf` configuration file. Therefore, HLCaches can be deployed on existing Apache proxy servers by simply installing the modified executable and adjusting a small number of configuration parameters.

The **Query Engine** is a custom-made component designed to process, decompose and normalize XPath [7] queries into LDAP constructs that the **XMLDAP Cache** can understand.

Any other type of queries, like URL requests that do not contain XPath expressions, are forwarded to the **Distributed Cache Engine** to determine whether or not the system as a whole could process the request.

The decomposition and normalization of queries plays an extremely important role in the cache answerability problem, and more importantly, in the efficient processing and delegation of queries and subqueries, as we will see later on in section 4.

In order to speed up the querying process, the **Query Engine** works in tight cooperation with the **Distributed Cache Engine**, whose only purpose so far, is the management of a distributed index of the contents of both, the **XMLDAP Cache** and the **Traditional Cache** from all nodes in the system. In the future, the **Distributed Cache Engine** will also be responsible for the efficient integration and processing of Document Type Definitions (DTDs) to allow for the correct characterization and utilization of semantically related cache entries with different syntax, as detailed in [13].

Due to the nature of the internal representation in our **XMLDAP Cache** all access to this component is made through the **XML Parser**, which is responsible for the translation of XML data into LDAPQL, the LDAP Query Language described in section 4.

This translation process is what differentiates our caching mechanism from traditional approaches, where the document is treated as a normal file with no additional characteristics. HLCaches, on the other hand, generates an LDAP tree (see section 3) that allows the system to extract additional information used by the **Query Engine** to efficiently direct its efforts to the appropriate place in the document.

As an additional feature, the **XML Parser** also handles the conversion of an LDAP tree structure to XML, allowing for on-the-fly generation of well-formed XML documents from partial documents cached at a previous time.

The **XMLDAP Cache** is based on the last available version of the OpenLDAP server [11] modified to provide the higher performance requirements typical of a caching system. The purpose of this component is to serve as a specialized cache for XML documents, and therefore, we have changed the standard configuration to fulfill the requirements imposed by the storage of generic XML documents, as we will see in section 3.

In the HLCaches system, the **XMLDAP Cache** runs as a separate process from the Apache server, and can therefore be installed either locally to the Proxy server or in another node, which allows for greater configurability and flexibility. In fact, the job of the **Distributed Cache Engine** can be greatly simplified if one of the caches in the system is shared among several of its member nodes.

The **Traditional Cache**, on the other hand runs as a module inside the Apache server in order to provide a caching mechanism for other types of documents besides XML. The implementation of the **Traditional Cache** is based on the `mod_proxy` module which, since its conception has been included in all Apache distributions to cache HTML documents, GIF files, JPEG files, etc.

3 Internal Representation

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, as described in [23]. If the reader is not familiar with the LDAP model and protocol, we refer it to [14] for an informal description of its features, or to [15] for a formal one.

The main goal of our structure is to provide the following characteristics:

- Allow for arbitrary XML documents to be stored efficiently.
- Avoid unnecessary changes in the LDAP schema that slow down the storage process.
- Allow for efficient query processing, partitioning and forwarding.
- Provide support for meta-data that describes the current contents of our system.

Based on these requirements, we have designed three representation models that, despite their simplicity, provide an elegant and uniform environment for effective processing of XPath queries in our system. Since XPath is the basis for more complex querying languages like Quilt [8], support for XPath is a necessary requirement for any system that deals with semistructured data in the form of XML. The following sections go into detail about each one of the internal representations: XML documents, in section 3.1; XPath queries, in section 3.2; and meta-data, in section 3.3.

3.1 Data Representation

Anybody familiar with both, the XML specification and the LDAP model can infer the connection between Document Type Definitions (DTDs) in the XML world and the class and attribute schema definition in LDAP. The most obvious way to represent XML documents in LDAP is, therefore, 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:

- The integration of a DTD in LDAP involves changes to the schema that, by definition, is assumed to be static in LDAP [23].
- Even in the case where extensions to the basic LDAP model are used so that dynamic changes in the schema are allowed [27], their update is troublesome and can introduce illegal nodes at the instance level.
- The DTD for most XML documents is not available, reducing the number of cacheable XML documents considerably.

```
XMLNode OBJECT-CLASS ::= {
  SUBCLASS OF {top}
  MUST CONTAIN {oc,oid,name}                    // required attributes
  TYPE oc OBJECT-CLASS
  TYPE oid DN                                   // dns formed by oids
  TYPE name STRING
}
XMLElement OBJECT-CLASS ::= {
  SUBCLASS OF {XMLNode}
  MUST CONTAIN {order}                          // required attributes
  MAY CONTAIN {value}                           // allowed attributes
  TYPE order INTEGER
  TYPE value STRING
}
XMLAttribute OBJECT-CLASS ::= {
  SUBCLASS OF {XMLNode}
  MUST CONTAIN {value}                          // required attributes
  TYPE value DN, STRING
}
```

Figure 3: LDAP Class Hierarchy for XML Data Representation

Therefore, we have developed our internal representation in such a way to eliminate these problems without losing the ability to incorporate arbitrary XML documents in our system. Figure 3 describes the contents of the `XMLNode`, `XMLElement` and `XMLAttribute` classes that constitute the basic elements of our representation.

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 3 to store information about individual XML nodes.

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.

The meaning of the attributes and classes defined in figure 3 is pretty straight forward. The `oc` attribute denotes whether the node represents an `XMLElement` or an `XMLAttribute`, and is a required attribute of every LDAP node. The `oid` attribute, also part of every LDAP node, contains its distinguished name, and therefore, information about the location of this particular node in the document hierarchy.

The `name` attribute contains the name of the `XMLElement` or `XMLAttribute` defined by the node and, although not a required attribute by the LDAP specification, it is a required attribute of our model, since every XML node needs to have a name. The combination of the `oid` and the `name` attribute constitute an ad-hoc implementation of the namespace concept defined in the XML standard, allowing for the sharing of names at different levels of the hierarchy without introducing name clashes.

The `XMLElement` class differs from the `XMLAttribute` class in that the former requires the presence of an `order` attribute, whereas the latter does not even allow it. This attribute is used to encode the relative ordering of a particular node in the document hierarchy which, as we will see later, is crucial for the correct implementation of queries like `next-node`, `previous-node`, `next-sibling`, etc., as defined in the XPath standard [7].

Finally, the `value` attribute is used to store the contents of either an `XMLElement` or an `XMLAttribute`. Since an `XMLElement` can have an empty body, the existence of the `value` attribute is not guaranteed, as opposed to an `XMLAttribute` that, by definition, requires it.

```
Algorithm XML2LDAP (D)
    Let D be an XML document to processed from left to right
    /* Initialize the current node to the top of the LDAP cache tree */
    CurrentNode = "(cn=Cache,dc=top)"

    while there is input i from D
        /* If we find an opening tag in our input i */
        if i is <tagName attrName_0 = attrValue_0 ... attrName_n = attrValue_n>
            NewNode = XMLElement(tagName)
            link(CurrentNode, NewNode)
            CurrentNode = NewNode

            /* Create the attributes and link them to the new node */
            for each attrName, attrValue pairs
                NewAttribute = XMLAttribute(attrName, attrValue)
                link(NewNode, NewAttribute)

        /* If we find a closing tag in our input i */
        if i is </tagName>
            CurrentNode = Parent(CurrentNode)

        /* else, i is the content of the node */
        else
            CurrentNode.value = i
```

Figure 4: XML-to-LDAP translation algorithm

Given this definition, the `XML2LDAP` algorithm depicted in figure 4 is used whenever an XML document D is parsed to be included in our `HLCaches` system. Its inverse algorithm (`LDAP2XML`) is obvious and not presented here.

In order to illustrate the translation process that occurs whenever an XML document is incorporated into our cache, we have taken as an example (figure 5) an excerpt of the Mondial database available in XML form from [13].

Figure 6 contains the graphical representation of the LDAP tree generated as a result of applying the previous algorithm to the contents of figure 5. In our system, we have adopted the convention that all XML documents are stored under the `(cn=Cache,dc=top)` node (as indicated in the figure), but this does not change the generality of our procedure and representation.

```
<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">
    <name>Baden Wurttemberg</name>
    <area>35742</area>
    <population>10272069</population>
    <city id="cid-9" is_state_cap="yes" country="D"
          province="B-W">
      <name>Stuttgart</name>
      <longitude>9.1</longitude>
      <latitude>48.7</latitude>
      <population year="95">588482</population>
    </city>
  </province>
</country>
```

Figure 5: Excerpt from a Mondial XML file

Theorem 1 *The `XML2LDAP` algorithm is a correct procedure for the translation of arbitrary XML documents into LDAP. Any XML document can be transformed by `XML2LDAP` into a LDAP tree composed of `XMLNode`, `XMLElement` and `XMLAttribute` nodes, and reconstructed by the `LDAP2XML` algorithm.*

3.2 Query Representation

In addition to providing a data representation model, caches also need to be able to represent queries in order to determine whether or not a particular query has already been stored and/or is answerable using only the data obtained from previously cached information.

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

The detailed list of attributes that constitute an `XMLQuery` is shown in figure 7. The meaning of the `oc` attribute is analogous to that of the data representation classes previous defined. 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 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 parts of our distributed caching system to exchange information about their cache contents without having to actually send more than a couple bytes per cached document.

The next four attributes, `context`, `scope`, `xpathquery` and `result` define a query or subquery in terms of the characteristics described in the XPath specification [7].

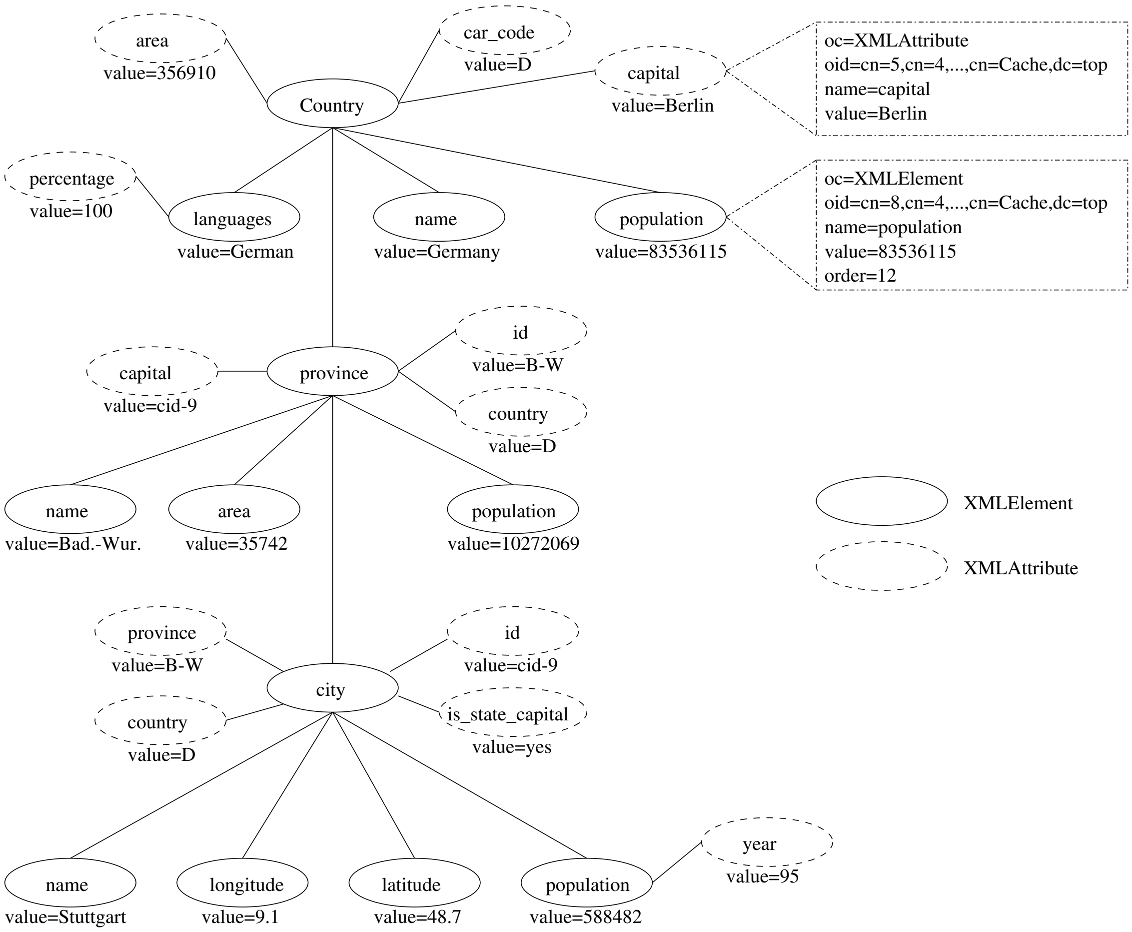

Figure 6: XML Data Integration

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 caching 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 section 4.

Finally, the last three attributes keep information used to determine how long this query (and its results) should be kept in the cache. The `create_time` attribute contains a timestamp with the specific time that this entry was originally created. The `access_time` is analogous to `create_time`, but contains the timestamps of the last two accesses to the query. Periodically, the server purges entries that have not been accessed within a predefined interval. Lastly, the `popularity` attribute keeps a count of the number of times this query has been accessed.

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

Figure 7: LDAP Classes for Query Representation

3.3 Meta-data Representation

The last piece of internal information kept by each proxy node in the system is meta-data, that is, data about the contents of the cache, as well as a set of currently stored queries and the location of their results.

We store meta-data in our system in a special LDAP node called `(cn=admin,cn=Cache,dc=top)` that has the following attributes:

`next`: Contains a monotonically increasing integer used to assign a unique internal name to every new XML document stored in the cache.

`count`: Represents the number of XML documents stored at a given time in the system.

`node_count`: Represents the number of nodes stored in the system.

`maturity`: Represents the lower access interval limit where an entry is considered to be "aging" in the system.

`death`: Represents the upper access interval limit used by the purge routine to decide that an entry must be eliminated.

`low_watermark`: The lower limit of nodes in the system that trigger the purge routine.

`high_watermark`: The absolute hard limit of nodes in the system that render it unusable until the purge routine deletes the necessary number of nodes to drive the current `node_count` below this limit.

These attributes, in collaboration with the last set of attributes defined for `XMLQuery` nodes in the previous section, allow for the management and administration of the cache and its contents, without incurring in any noticeable overhead.

It is worth noting that our caches are strictly read-only and that all write operations are forwarded directly to the information systems, allowing us to relax our representation and avoid a whole set of other administration attributes, like a `dirty` attribute, that unnecessarily complicate the management of the cache. Although this restriction might appear as a serious draw-back from our cache, the high ratio of read vs write operations on the Internet backs our decision up to the point of making it a reasonable design policy. Furthermore, issues of cache consistency among members of our system are easier to solve if we only allow the user to write directly to the server.

4 Query Model

The query model used by `HLCaches` is very close to the traditional LDAP query model described in the standard specification [23] and in other pieces of the literature [14, 15]. 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 [7] support. Other papers [15] 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 in our system, we are able to support not only XML (since XPath has become the de facto querying language for XML), but also a whole breadth of semistructured models whose querying languages are based on variations of the same idea exploited by XPath, e.g. Quilt [8].

4.1 Traditional LDAP Query Model

In the traditional LDAP query 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 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.

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

4.2 XPath Query Model

As specified in [7], the primary purpose of the XPath standard 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 along the path.

Example The XPath 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` node that have an attribute `car_code` with value “D”.

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
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
preceding-sibling	Any same-level node with id lower than the current node
self	The current node

Table 1: Allowed Axis Expressions in XPath

4.3 LDAPQL Model

In our `HLCaches` system, we have extended the traditional LDAP query model described in section 4.1 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, namely, from the root to the leaves, whereas XPath contains axis (see table 1) that perform queries in the "upwards" direction.

Therefore, we need to extend the notion of `scope` in the traditional model to also 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:

- $b_{Q_{HL}}$ 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,

  `parent` if only the parent node should be considered,

  `siblings` if only the siblings of the node should be considered,

  `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 $\{=, \neq, <, \leq, >, \geq\}$; and 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.

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

4.4 Evaluation of XPath Queries using HLCaches

In order to evaluate an XPath query Q_X, we need to first translate the original statement into an LDAPQL query so that our system can compute the result. Figure 8 contains pseudocode for the `XPath2LDAPQL` algorithm that performs the translation of the query.

Algorithm XPath2LDAPQL (Q_X)

```
Let Q_X be an 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}) ∈ Q_X
    /* Create a new XMLQuery node and initialize its attributes */
    NewXMLQuery.context = C_i
    NewXMLQuery.xpathquery = w_i
    NewXMLQuery.hash = hash(w_i)

    /* For each node in the context, evaluate w_i on it */
    for each n ∈ C_i
        C_{i+1} = C_{i+1} ∪ EVAL(PET(n, w_i))

    NewXMLQuery.result = C_{i+1}
```

Figure 8: XPath-to-LDAPQL translation algorithm

The translation algorithm takes care of filling up the necessary information for each of the subqueries in an instance of the `XMLQuery` class, and uses the `EVAL` and `PET` procedures to perform the evaluation and translation of the path expressions found in each subquery. Figure 9 contains the definition of the two subroutines.

The `PET` and `EVAL` subroutines invoke in turn the `LDAP` subroutine to retrieve the result of a query, and the `BaseScope` procedure, whose purpose is to provide the right base, scope and initial type filter for an LDAPQL query given a base node and the axis component of a path expression. This function is best described by means of a table (see table 2), where the first two columns represent the input to the function, and the last three its output.

The integration of built-in functions in `XPath2LDAPQL` involves testing the a_i and c_i expressions in the `PET` subroutine for reserved names of functions that need to be executed before the `BaseScope` function is called. Although present in our `HLCaches` system, the formalization of

Algorithm EVAL (Q, S)

/* Q is an LDAPQL query (called main query) */
/* $S = \{S_i\}$ is a set of LDAPQL queries (subordinate) */
Result = LDAP(Q)
for each subquery $S_i \in S$
    Result = Result $\cap$ LDAP(S_i)
Return Result

Algorithm PET(n, w_i)

/* n is a distinguished name and $w_i = a_i :: e_i[c_i]$ */
Let Q_{HL} be an LDAPQL query (called main query)
Let $S = \{S_j\}$ be a set of LDAPQL queries (subordinate)

/* Translate a_i into $Q_{HL} = (b_{Q_{HL}}, s_{Q_{HL}}, f_{Q_{HL}}, p_{Q_{HL}})$ */
$(b_{Q_{HL}}, s_{Q_{HL}}, f_{Q_{HL}})$ = BaseScope(n, a_i)
for each $nodeName \in e_i$
    $f_{Q_{HL}} = f_{Q_{HL}} \cap (\texttt{name} = nodeName)$
$p_{Q_{HL}} = \{\}$

/* Translate each predicate cp_j into $S_j = (b_{S_j}, s_{S_j}, f_{S_j}, p_{S_j})$ */
Let $S = \{\}$
for each $cp_j \in c_i$
    Let cp_j be of the form $term_j$ op_j $value_j$
    $(b_{S_j}, s_{S_j}, f_{S_j})$ = BaseScope(LDAP(Q_{HL}),$term_j$)
    for each $(nodeName, nodeValue) \in c_i$
        $f_{S_j} = f_{S_j} \cap (\&(\texttt{name} = nodeName)(\texttt{value} = nodeValue))$
    $p_{S_j} = \{\}$
    $S = S \cup S_j$

Return (Q_{HL},S)

Figure 9: XPath-to-LDAPQL algorithm (EVAL and PET subroutines)

built-in functions for their presentation in this paper adds complexity to the discussion without providing any better insight in the inner workings of our algorithm.

The most interesting properties of our `XPath2LDAPQL` algorithm can be summarized in the following theorems:

Theorem 2 *The `XPath2LDAPQL` algorithm exhibits linear time complexity with respect to the sum of the number of path expressions and atomic predicates found in the original XPath query.*

The validity of this theorem is intuitively verified by noticing that our algorithm does not need to backtrack at any point in the process.

Theorem 3 (Result Equivalence) *Given any arbitrary XML document, the evaluation of an XPath query using an appropriate parser 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.*

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 2: Axis Translation Function

In order to prove this last theorem, we need to verify the transformations in figure 10.

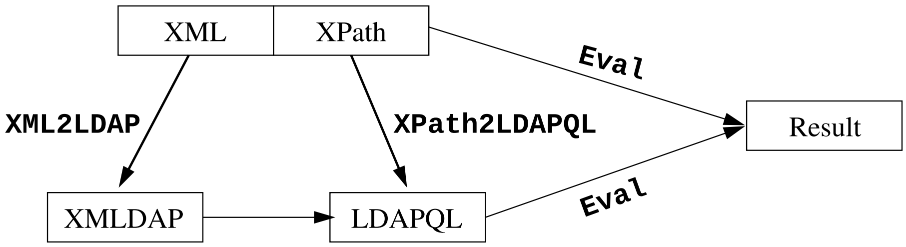

Figure 10: Result Equivalence

Example To illustrate the evaluation and translation process, if we apply our algorithm to the XPath query Q_X from section 4.2, it 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 figure 5 whose `car_code` attribute has the value “`D`”.

where the result of the PET algorithm for w_0 and w_1 is:

- $w_0 = child :: mondial[]$
  - $Q_0 = (dn(root), \texttt{onelevel}, (\&(oc = \texttt{XMLElement})(name = \text{“}mondial\text{”})), \{\})$
  - $S_0 = \{\}$
- $w_1 = child :: country[attribute :: car\_code = \text{“}D\text{”}]$
  - $Q_1 = (dn(mondial), \texttt{onelevel}, (\&(oc = \texttt{XMLElement})(name = \text{“}country\text{”})), \{\})$
  - $S_1 = \{(dn(country), \texttt{onelevel}, (\&(oc = \texttt{XMLAttribute})(\&(name = \text{“}car\_code\text{”})(value = \text{“}D\text{”}))), \{\})\}$

4.5 Advantages of our model

The combination of the internal data representation described in section 3 with the XPath mapping algorithm we have just proposed, offer, despite their simplicity, quite a few advantages that make them suitable for deployment in a distributed caching environment, like `HLCaches`.

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 [13].

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 to match their expectations.

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 that are performed by means of a compiler or graphical interface that translates higher level language queries into XPath queries, 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, with the added benefit that, 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 pre-computed subquery leading eventually to the correct result. The implementation of this procedure involves an almost trivial change to the `XPath2LDAPQL` algorithm, where the loop that performs the computation of the context C_i is unrolled and parallelized based on the (partially) known C_{i-1}.

The fact that a query evaluation could be potentially 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 either 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 keeps 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, thus making the evaluation process much more efficient.

5 Experimental Results

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

We have not considered any kind of replacement policies in our system because the aim of our experiments is to prove the feasibility of `HLCaches` as a caching mechanism for XML, and, for now, not to devise or propose new replacement policies. 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.

5.1 Experimental Setup

All experiments in the following sections 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 the Apache server involve the integration of the **Type Checker** and the **XMLDAP Cache**, as described in section 2, whereas the changes performed on the OpenLDAP server had to do with the implementation of the internal representation described in section 3, and 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.

File Name	Size	Apache Cache	Overhead	HLCaches	Overhead
`mondial-2.0.xml`	1037629	1038094	1.00	3372502	3.25
`europe-2.0.xml`	317913	318384	1.00	1017080	3.20
`dream.xml`	149524	149982	1.00	303613	2.03
`SigmodRecord.xml`	494591	495056	1.00	1401088	2.83
`books1.wml`	3129	3586	1.15	8039	2.57
Average	–	–	1.03	–	2.78

Table 3: Comparison of Storage Requirements

The data files used in our benchmarking experiments come from several sources: the Mondial database [13], that contains geopolitical information about countries, organizations, geographical entities, etc.; XSLBench [16], a performance benchmark of XSLT [6] processors maintained by Kevin Jones; the ACM Sigmod Record Database [1] in XML form; and a database of "Great Books" maintained in WML [25] at JollyRoger [19], which serves as an example of the use of `HLCaches` with WAP technology [24], since WML is nothing but a set of XML documents that conform to the WML Document Type Definition.

5.2 Document Storage Overhead

Table 3 contains data on the storage requirements of the most representative files we have used in our experiments. The "Apache Cache" column shows the storage requirements (in bytes) of the corresponding document under the "normal" caching technique used in the Apache project, whereas the `HLCaches` column contains the size of our internal data representation in compressed form, as it is stored on disk. This comparison is a fair one since `HLCaches` is designed to replace the traditional caching mechanism used by the proxy module in Apache.

As we can see in the table, the overhead of the traditional cache is minimal, since only a header containing information about the caching time, expiration date, size of the file, etc. is stored with the file. Therefore, small files suffer from a greater relative overhead as big files. For our internal representation, the storage requirements are about 2.8 times that of the original size of the file, with no direct correlation between size and relative overhead, as in the previous case. Although our representation has greater overhead than the traditional cache representation, the additional querying capabilities of our system make it a reasonable tradeoff.

5.3 Average Storage and Retrieval Time

Table 4 contains data on the storage and retrieval times of the same files used for the storage requirement experiment. The storage operation involves loading an XML document into `HLCaches` by means of the `XML2LDAP` algorithm, after a query has been formulated, whereas the retrieval operation assumes that the document is already in our system and needs to be reconstructed to

File Name	Nodes/Ops	Stor.(s)	Ops/sec.	Retr.(s)	Ops/sec.
mondial-2.0.xml	39633/57116	13.34	2970.99/4281.56	85.86	461.60/665.22
europe-2.0.xml	12783/18186	3.88	3294.59/4687.11	26.84	476.26/677.57
dream.xml	3361/6231	1.19	2824.37/5236.13	10.22	328.86/609.69
SigmodRecord.xml	15263/38518	8.43	1810.55/4569.16	56.33	270.95/683.79
books1.wml	96/138	0.0098	9795.92/14081.63	0.18	533.33/766.66
Average	–	–	2725.12/4693.50	–	384.27/659.07

Table 4: Average Storage and Retrieval Time

be returned to the client. For this experiment, the absolute size of the file is not so important as the number of element and attribute nodes in the document. The storage and retrieval times have been measured as seen by the Apache server after performing the corresponding operations, and are always greater than the ones seen by the clients, since they start receiving data from the server before the whole operation is complete, which helps perceive even better times from the client perspective.

As depicted in the table, `HLCaches` can process almost 4700 store operations per second, which correspond to about 2700 XML nodes/second, where each node is either an element or an attribute. The performance of storage operations is so good in comparison to the retrieval operations because they are performed asynchronously, meaning that the Apache server does not need to wait for the LDAP server to complete the operation before it sends the next one. On the other hand, retrieval operations are synchronous, since the order in which nodes are received is an important factor for the reconstruction of the original document from its individual nodes. Despite the relative performance disadvantage of read operations overall, the fact that clients start receiving the document as soon as the first bytes are generated, and that most documents are not as big as the ones used in our experiments, imply that there is no noticeable overhead for read operations seen from the perspective of the client.

5.4 Query Execution Performance Improvement

In the third 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 as detailed in section 4.

For the following set of queries, we have taken a C-based implementation of an XPath engine [22] developed for the Gnome project [9] 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 to use our `HLCaches` system as a substitute for the DOM representation needed to perform XPath queries on a XML document.

Table 5 shows some of the most representative XPath queries involving simple walk-throughs

Query Patterns	Nr. Result Nodes	DOM back-end	HLCaches
`/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 5: Performance for XPath using DOM and `HLCaches`

and predicate testing, and their processing times in seconds using the DOM backend and our `HLCaches` system. The queries were performed on a "clean" cache loaded with the `mondial-2.0.xml` file used in previous experiments. As can be seen in the table, `HLCaches` 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 5, but more importantly on complex queries that use these patterns as their building blocks. Furthermore, the easiness of integration of our `HLCaches` in a distributed environment, like the Internet, enables the evaluation of such queries in parallel, as described in detail in section 4.5.

6 Related Work

In the world of caching technologies for the Internet, the two most prominent examples of hierarchical caching systems are Harvest [2, 5] and Squid [12]. Although our system also provides a hierarchical caching structure, it differs from both systems, first, in that it also provides querying facilities, acting more like materialized view layer of a semistructured database than a traditional cache. Secondly, our system can reconfigure its topology on-the-fly, allowing for optimization techniques and methods not available on Harvest or Squid. Finally, our integration approach for arbitrary semistructured documents is generic, as opposed to the wrapper technology used by Harvest to implement brokers for indexing specific data formats.

Our approach is also similar to some active caching techniques in that we offer additional features and capabilities to non-traditional cacheable types, as opposed to HTML, image files, etc. However, the usual focus of active caching systems, like [4], lie on applications and applets either in Java or Javascript, whereas our system focuses on semistructured data and how to efficiently query it, and not take into account the more programmatic cacheable entities that populate the World Wide Web.

Finally, the distributed indexing mechanism implemented in our system relate to the notion of active catalogs in `Nomenclator` [18], where, interestingly enough, the focus was on X.500 directories [26], the precursor to LDAP. `HLCaches` differs from `Nomenclator` in its widest breadth of features, its caching functionality, and in the fact that the distributed index capabilities of `HLCaches` have to support indexing over incomplete semistructured databases that make the process a more challenging one.

7 Future Work

Even though a reasonable amount of work has been put in the development of our `HLCaches` system, there are quite a few experiments and future plans that are either on development at the time, or we plan to include in our system in the future. The most important topics are listed below:

Distributed Indexing: We are in the process of implementing distributed indexing capabilities in `HLCaches` that will improve the guidance and management of the evaluation process of parallelizable queries.

Topology Discovery: Experiments need to be performed on policies and mechanisms to perform the automatic discovery of optimal topologies in our system.

Subquery Analysis: Further experiments are needed with "typical" batches of XPath queries to determine to what degree subquery processing, preevaluation and storage should be encouraged.

Optimization Policies: Although some optimizations already take place in our system, we need to systematically address each link in the processing chain to try to discover new optimization mechanisms and policies, either at the XPath level, or at the internal representation level.

Replacement Policies: Policies to determine what to delete from the cache, or to send to other peers for off-site storage become important as soon as the deployment of `HLCaches` starts to take place into systems with limited caching space. Since these policies should make use of the distributed indexing capabilities of our system, traditional LRU techniques, might not be optimal if we take into account semantic information and the contents of other nodes.

8 Conclusion

In this paper we have presented the design, internal data representation details and query model of `HLCaches`, a hierarchical distributed caching system based on LDAP technology and developed for the efficient storage and retrieval of semistructured data on the Internet.

We have presented evaluation algorithms that allow our system to provide full support for XPath queries, as well as a storage model for XPath based on subquery processing that enables the seamless integration of distributed and parallel query execution on `HLCaches`.

We have further provided experimental data that characterizes the storage requirements and average access times of our system in comparison to more traditional caching schemes. Although our approach incurs in some additional overhead, mainly in terms of storage space, the advantages it provides 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 the WML standard.

To the best of our knowledge, there is no other distributed caching system available nowadays that is able to efficiently cache and provide additional querying capabilities for semistructured data while, at the same time, allow the integration of more traditional data representations, like HTML, graphics, etc. in a common, distributed system.

References

[1] ACM. Sigmod record web edition. `http://www.acm.org/sigs/sigmod/record`, December 2000.

[2] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, Michael F. Schwartz, and Duane P. Wessels. Harvest: A scalable, customizable discovery and access system. Technical report, University of Colorado at Boulder, March 1994.

[3] 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.

[4] Pei Cao, Jin Zhang, and Kevin Beach. Active cache: Caching dynamic contents on the Web. In *Proceedings of the 1998 Middleware conference*, September 1998. `http://www.cs.wisc.edu/~cao/papers/active-cache.html`.

[5] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical internet object cache. In *Proceedings for USENIX '96*, 1996.

[6] James Clark. Xsl transformations (xslt) version 1.0. `http://www.w3.org/TR/xslt`, November 1999.

[7] James Clark and Steve DeRose. Xml path language (xpath) version 1.0. `http://www.w3c.org/tr/xpath`, November 1999.

[8] Quilt: An XML Query Language for Heterogeneous Data Sources. Don chamberlin and jonathan robie and daniela florescu. In *Proceedings of the Third International Workshop on the Web and Databases*, pages 53–62, Dallas, Texas, May 2000.

[9] GNOME. `http://www.gnome.org/`.

[10] Apache Group. Apache web server. `http://www.apache.org/`.

[11] OpenLDAP Group. Openldap server. http://www.openldap.org/.

[12] M. Hamilton, A. Rousskov, and D. Wessels. Cache digest specification - version 5. http://squid.nlanr.net/CacheDigest/cache-digest-v5.txt, December 1998.

[13] Reference removed for double blind reviewing.

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

[15] H. V. Jagadish, Laks V. S. Lakshmanan, Tova Milo, Divesh Srivastava, and Dimitra Vista. Querying network directories. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 133–144. ACM Press, 1999.

[16] Kevin Jones. XSLBench 1.4. http://www.tfi-technology.com/xml/xslbench.html, October 2000.

[17] Håkon Wium Lie and Bert Bos. Cascading style sheets, level 1. http://www.w3.org/TR/REC-CSS1, January 1999.

[18] J. J. Ordille and B. P. Miller. Distributed active catalogs and meta-data caching in descriptive name services. In Robert Werner, editor, *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 120–129, Pittsburgh, PA, May 1993. IEEE Computer Society Press.

[19] Jollyroger.com great books. http://jollyroger.nbci.com.

[20] Albrecht Schmidt, Martin L. Kersten, Menzo Windhouwer, and Florian Waas. Efficient relational storage and retrieval of xml documents. In *Proceedings of the Third International Workshop on the Web and Databases*, pages 47–52, Dallas, Texas, May 2000.

[21] Jayavel Shanmugasundaram, Kristin Tufte, Gang He, Chun Zhang, David DeWitt, and Jeffrey Naughton. Relational databases for querying xml documents: Limitations and oportunities. In *Proceedings of the 25th VLDB Conference*, Edinburgh, Scotland, 1999.

[22] Daniel Veillard. http://www.xmlsoft.org/.

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

[24] http://www.wapforum.org/.

[25] http://www.oasis-open.org/cover/wap-wml.html.

[26] Information processing systems - open systems interconnection - the directory: Overview of concepts, models and service. ISO/IEC JTC 1/SC21, International Standard 9594-1, 1988.

[27] Y. Yaacovi, M. Wahl, and T. Genovese. Lightweight directory access protocol (v3): Extensions for dynamic directory services. RFC 2589, May 1999.

[28] Khaled Yagoub, Daniela Florescu, Valérie Issarny, and Patrick Valduriez. Caching strategies for data-intensive web sites. In *Proceedings of the 26th VLDB Conference*, Cairo, Egypt, 2000.