Cooperative XPath Caching

Kostas Lillis and Evaggelia Pitoura
Computer Science Department
University of Ioannina, Greece
{klillis, pitoura}@cs.uoi.gr

ABSTRACT

Motivated by the fact that XML is increasingly being used in distributed applications, we propose building a cooperative caching scheme for XML documents. Our scheme allows sharing cache content among a number of peers. To facilitate sharing, a distributed prefix-based index is built based on the queries whose results are cached. In the loosely-coupled sharing approach, each peer stores in its local cache results of its own queries and just publishes the associated queries to the index. In the tightly-coupled approach, each peer is assigned a specific part of the query space and stores in its local cache the results of the corresponding queries. Both approaches result in a dynamic organization of content that evolves over time based on the query load, the number of peers and the overall storage available. We present a number of associated design choices such as using a DHT for distributing the prefix-based index and a proactive cache replacement policy. We also report on a number of experiments that show the benefits of cooperative caching and highlight the pros and cons of loosely and tightly coupled cache sharing.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*distributed systems, performance evaluation*

General Terms

Performance

Keywords

XML, cache, peer-to-peer systems

1. INTRODUCTION

XML is being used widely in data exchange applications in the Internet. As the size and number of XML documents increase, caching results of previous queries can substantially reduce the communication and computational cost of processing them in a distributing setting. Caching schemes for XML databases usually follow a semantic approach [7, 8, 14]: cached data are not organized at the tuple, page or document level but instead at the level of query descriptors. In abstract terms, the answers of previous queries are stored in cache along with the associated query analogously to materializing a view. As new queries arrive, the system checks the cache to determine whether the query can be answered by the cached results of some previous queries.

In many applications, such as when deploying web services, or accessing XML sites from the web, XML documents are queried by a number of peers in close network proximity with each other. The actual documents themselves are located in a large number of remote data sources, thus the cost of locating them and transferring the answers to the peers is expected to be high. Hence, it is central that the results of the queries of each peer can be reused to answer subsequent queries posed by not just the same peer but also by other peers that submit similar queries. This motivates building a cooperative cache.

We consider two fundamental ways of sharing cache content in terms of the degree of cooperation among the participating peers. In the *loosely-coupled approach*, that we call *IndexCache*, each peer locally caches the results of its own queries. A distributed index is built on top of these local caches to facilitate sharing. Each query consults the index to locate any peer whose cached results may be used in answering it. In the *tightly-coupled approach*, that we call *DataCache*, each peer is assigned a particular part of the cache data space. The results of each query are not stored at the peer that posed it, but instead at the peer that is responsible for that part of the data space. While in the IndexCache, the overall content of the cooperative cache is influenced by the local workload of each peer, in the DataCache, the overall content is affected by the aggregated workload over all peers.

In both approaches, caching is dynamic in the sense that the physical organization of data changes; it shrinks and expands based on the current query load, as well as the number of peers and the available overall storage. This leads to a form of self-organization. Documents that are frequently queried are indexed and can be located by looking up the cache entries. In contradistinction, documents that are less popular are not indexed and thus must be searched along all, potentially remote, data sources.

A central issue is how to organize the index in the case of IndexCache or the data content in the case of DataCache. Both the DataCache and the IndexCache are semantic caches and thus, we index them based on the queries that describe their content. We propose using a prefix-based index that takes advantage of the XPath pattern of queries. To distribute the index nodes among the participating peers, we use an approach based on distributed hash tables (DHTs). DHTs support (either directly or indirectly) a hash-table interface of *put*(key, value) and *get*(key) [18]. More than one trie node may be assigned to a peer resulting in balanced load distribution. DHTs are scalable in that both the latency of the basic operations and the local state at each peer typically grow logarithmically with the number of peers. Also, they are equipped with advanced load-balancing and reliability protocols.

Our cache replacement strategies exploit the subsumption relation among queries in maintaining access statistics. We also propose a proactive replacement scheme that is shown to lead to a faster adaptation of cache content in cases of rapid query workload fluctuations. In addition, our strategy allows each peer to dynamically partition its storage among the index nodes that are assigned to it, by letting both their number and their size vary with the workload.

We have fully implemented both schemes on top of a DHT, namely CHORD [18]. We present experimental results that (a) evaluate the benefits and overheads for building a cooperative cache, (b) identify the conditions under which DataCache is preferable to IndexCache and vise versa, and (c) evaluate our replacement strategies from a variety of perspectives.

In a nutshell, in this paper, we:

- propose and evaluate two ways of building a distributed, self-organized, semantic XML storage scheme,
- exploit a prefix-based approach to indexing and storing path queries that uses a DHT interface for distributing the index among the participating peers,
- introduce appropriate replacement strategies, and
- present a thorough experimental evaluation that clarifies the factors that actually affect cache performance.

The rest of this paper is organized as follows. In Section 2, we introduce the problem of building a cooperative XML cache. In Section 3, we present our basic data structure for indexing and storing cache results. In Section 4, we describe our replacement strategies, while in Section 5, we report our experimental results. Section 6 includes a comparison with related work and Section 7 our conclusions.

2. COOPERATIVE CACHE SHARING

In this section, we introduce the problem of cache sharing and the two basic approaches for addressing it.

2.1 Problem Formulation

We assume a network of N nodes or peers, p_i, $1 \leq \mathrm{i} \leq N$, that pose queries on XML documents. The XML documents are located at a large number of widely distributed nodes which are not necessarily the same with the peers posing the queries. That is, the set of peers that cooperate in sharing their caches does not need to coincide with the set of nodes that act as data sources. For instance, data sources may be located in remote and disperse web sites, correspond to wide area sensor databases [9] or constitute distributed hierarchical directory entries [3]. Each one of the N peers offers some storage space C_p for caching query results. As opposed to caching for reducing the I/O or the computational cost, caching in this setting mainly aims at reducing the network cost. First, we save on the communication cost of locating the data sources holding the matching documents. Then, by moving documents closer to their requesters, the cost of transferring any query results is also reduced. The documents at each peer may be either stored inside a native or a relational database or maintained outside the database at the application tier. Our focus in this paper is on the dynamic organization and management of the overall *content* of the cache, so that the most popular items are indexed and stored efficiently.

Our cooperative cache is a semantic one. At any instant, the cache contains (the results of) a set of queries, $S = \{Q_1, Q_2, ..., Q_m\}$. When a new query Q is posed at any of the peers, the peer checks whether Q can be answered from the results of the cached queries. If this is the case, there is a *cache hit*. Otherwise, there is a *cache miss* and Q must be answered from scratch, possibly by initiating an expensive search for data sources having documents matching it. The results of Q may then be added to the set of cached queries S. To achieve efficient cache sharing among peers, the cache content is indexed. The index is built on the set S of cached queries.

To fully benefit from caching, query *subsumption* can be exploited. We say that Q_1 *subsumes* Q_2, if the result of Q_1 contains the result of Q_2 and thus it can be used to answer it. There is a lot of work on the topic of determining a subsumption or containment relation between queries on XML documents [4, 14, 15]. In this paper, we assume that Q is an XPath query. The result of each query are the fragments F_i of the original XML documents for which the path P_i from the root of the document to the root of the fragment satisfies the query. Our tests for subsumption are based on string matching. Initially, we consider queries that are linear path expressions (containing only the child axis) each starting from the root node of the document. Later on, we relax this assumption to handle more general queries. To determine subsumption between queries, we use the following observation:

CLAIM 1. *For linear path queries, query Q_1 subsumes query Q_2, if Q_1 is a prefix of Q_2.*

PROOF. This results as a special case of the answerability criteria in [14]. □

Using as the unit of caching simple XPath queries simplifies checking for subsumptions. This leads to an efficient implementation of distributed lookup. It also reduces the overheads of managing the distributed cache.

2.2 Index Versus Data Sharing

With the *IndexCache* approach, the results of each query are stored locally at the peer that posed it. Next, each of these results (i.e. the corresponding query) is indexed so that it can be located by other peers. *DataCache* follows a tightly-coupled approach to cache sharing: each peer is assigned a specific part of the query space. The results of each query are cached at the peer which is responsible for the corresponding part of the query space. In DataCache, there is control over the placement of cache content and thus there is no redundancy. In particular, the following property holds for DataCache:

Property 1. *In DataCache, if fragments F_i and F_j, $i \neq j$, are cached, there does not exist a subsumption relation between their corresponding path expressions P_i and P_j.*

DataCache induces an additional overhead of moving query results from the peer posing the query to the peer actually caching them. Furthermore, with IndexCache, each peer can exploit better its own query workload temporal locality, since it can answer some of the queries locally from its cache without having to access the index. On the other hand, DataCache has some advantages over the local caching solution. First, in IndexCache, very active peers may replace recently accessed queries from their local caches, while less recently accessed queries will remain in the caches of the less active peers. Second, in IndexCache peers caching popular queries will be overloaded, while in DataCache the popular content will be distributed among a number of peers achieving better load distribution. Third, with IndexCache the lack of control over other peers' caches may result in having fragments whose path expressions subsume that of fragments cached at some other peer, thus creating redundancy and hence reducing cache space availability.

3. PREFIX-BASED XML CACHING

To support efficient substring matching for testing subsumption, we use a prefix-based index. The trie nodes are distributed among the peers forming the cache.

3.1 Distributed Prefix Trie

A trie is a tree for indexing and storing strings. Trie nodes are labeled with prefixes of the indexed strings such that every node corresponds to a distinct prefix of the data domain being indexed. The actual strings are stored in the leaf nodes with which they share a common prefix. Linear XPath queries can be represented as strings, by considering each element of the query as a character. In particular, the labeling of prefix trie nodes, for a prefix trie indexing a query set $S = \{Q_1, Q_2, \ldots, Q_m\}$ is defined as follows. The root node is labeled with the empty string. The labels of the other nodes are defined recursively: given a node labeled $/l$, its n children are labeled $/l/l_1, /l/l_2, \ldots, /l/l_n$, where $l_i \neq l_j$, $i \neq j$, $i, j = 1, \ldots, n$, and there are $Q_i, Q_j \in S$ s.t. $/l/l_i, /l/l_j$ are prefixes of Q_i, Q_j respectively. A special labeling case involves trie leaves: given a node labeled $/l$, one and only one of its leaf children can be labeled $/l/NULL$, where $/NULL$ is a dummy element, if $\exists\ Q_i \in S$ s.t. $/l = Q_i$. We call such leaves $NULL$ leaves and denote their label by $Q_i\perp$. $NULL$ leaves are introduced to handle redundancy among cache content. Their role will be made clear in Section 3.2. It holds that:

Claim 2. *Given the above labeling scheme, for each query $Q \in S$, there exists either exactly one leaf whose label is a prefix of Q or exactly one $NULL$ leaf with label $Q\perp$.*

Proof. Suppose there exist two leaves lf_1 and lf_2 whose labels lb_1 and lb_2 are prefixes of Q. Then, either: (i) $lb_1 = lb_2$ or (ii) lb_1 is a prefix of lb_2 (the case of lb_2 being a prefix of lb_1 is similar). In case (i), lf_1 and lf_2 are children of the same internal node nd. Suppose that the label of nd is l, then $lb_1 = l/l_i$ and $lb_2 = l/l_j$. Since $lb_1 = lb_2$, it follows that $l_i = l_j$ which is not consistent to the labeling of trie nodes, thus $lb_1 \neq lb_2$. In case (ii), lf_2 is a descendant of lf_1. This is not possible since trie leaves have no children. Suppose there exist two $NULL$ leaves, $l_1/NULL$ and $l_2/NULL$, whose label is $Q\perp$. Then it follows that $l_1 = l_2$, and that the two $NULL$ leaves are children of the same internal node. But each internal node has only one $NULL$ leaf child. Suppose there exists one leaf lf_1 whose label lb_1 is a prefix of Q and one $NULL$ leaf lf_2 whose label lb_2 is $Q\perp$. Then lb_1 is a prefix of lb_2, thus lf_2 is a descendant of lf_1. This is not possible, since trie leaves have no children. □

The prefix trie is constructed to adhere to the following properties:

Property 2. *(Trie Properties)*

1. A query Q is indexed/stored at the unique leaf node whose label is either a prefix of Q or $Q\perp$.
2. Each leaf node has a predefined storage capacity C.
3. The leaves in the subtrie of each internal node index/store queries of size at least $C-k$ (k is a predefined number).
4. Each trie node records the labels of its parent and children (if any).

Property 2.1 states how queries are mapped to trie nodes. From this property and Claim 2, it follows that the label of each leaf is a common prefix of all queries indexed/stored in it. Properties 2.2 and 2.3 determine how the trie adapts to the distribution of the cached queries. When indexing/storing a new query, the total size of the queries indexed/stored in a leaf might exceed the leaf capacity C. To enforce Property 2.2, the node is split and its queries are redistributed among the new nodes according to Property 2.1. When deleting a query, the total size of the queries contained in a subtrie may drop below $C - k$. To enforce Property 2.3, the whole subtrie is merged into a single leaf containing all queries in the subtrie. Parameter k determines how aggressive merging is. We have experimentally seen that a suitable value for k is $C/2$, which reduces the number of merges and consequently the overhead induced. Property 2.4 facilitates the lookup procedure.

Initially, the prefix trie consists of just the root node and, as new queries are indexed/stored, nodes are created and split according to the query distribution. Thus, the shape of the trie depends on the distribution of cached queries; it is deep in regions where a larger number of queries are indexed/stored, and shallow in regions where fewer queries are indexed/stored. The trie is stored at the peers that participate in the cooperative cache, by distributing its nodes using their label as the DHT key. Thus, given a label, it is possible to locate the peer storing the corresponding trie node via a single DHT lookup (*get*). The newly created leaves and internal nodes are gradually distributed among the peers as queries are deleted/inserted in the cache. Note, that more than one trie node may be assigned to a peer. Any type of DHT can be used; however, the choice of the DHT will influence some characteristics such as the operation complexity (through the *put*/*get* interfaces), the maintenance cost and the load and fault tolerance characteristics.

Figure 1 provides an example of how the trie is distributed among the peers in the DHT. The leaves and the internal nodes of the trie are assigned to the peers participating in the DHT through its put interface, e.g. $put(/A/C)$ assigns leaf $/A/C$ to peer $P14$. In DataCache (Figure 1(right)), the leaves store the actual fragments. In IndexCache (Figure 1(middle)), the leaves maintain pointers to the peer that

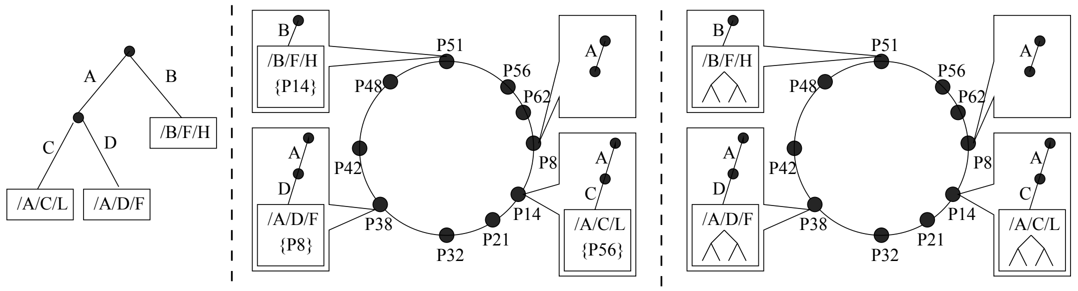


Figure 1: A prefix trie (left) and its distribution in IndexCache (middle) and DataCache (right).

stores the corresponding fragment in its local cache; for instance, peer $P14$ stores a pointer to peer $P56$ that has cached the corresponding fragment.

Our trie resembles the Prefix Hash Tree (PHT) introduced in [6]. The PHT is a binary trie built over a data set consisting of binary strings. There are differences between PHT and the prefix trie used here. First, in the prefix trie the domain being indexed consists of XPath queries instead of binary strings, consequently each internal node may have more than two children. Second, the introduction of $NULL$ leaves due to the semantic nature of the cache significantly differentiates the trie properties and operations.

3.2 Distributed Cache Operations

Next, we describe the basic cache operations for IndexCache and DataCache.

3.2.1 IndexCache Operations

In IndexCache, each peer caches locally the results of its own queries and publishes the corresponding query in the DHT index. When a query is posed, each peer first checks its cache. If the results of the query can be found in its cache, there is a *local hit*. Else, the peer searches the index for relevant results cached at other peers. If such results are found, there is a *global* hit, and the results are retrieved from the caches of the peers storing them.

An important issue concerns redundancy. IndexCache offers a loosely-coupled form of cache cooperation, where peers share the content of their caches, but retain control over the content of their own caches. Assume that peer p_1 caches the results of a query Q_1, and some other peer p poses a query Q that subsumes Q_1. Peer p_1 is not forced to delete the results of Q_1 from its cache. Instead, p caches Q and $NULL$ leaves are used for inserting $Q\perp$ in the DHT index. This adds some redundancy, however, p_1 can still benefit from local hits. However, if p_1 itself poses a query that subsumes Q_1, then the new query replaces Q_1 in the cache of p_1.

We detail next, the basic cache operations, namely: looking up, inserting and deleting a query.

Cache Lookup When a peer poses a query Q, it first checks its own cache. In the case of a local cache miss, Q is looked up in the DHT trie index. According to Property 2.1, Q is indexed at the unique leaf whose label is either $Q\perp$ or a prefix of Q. Thus, initially a DHT lookup is performed for $Q\perp$. If this is not found, then the leaf lf having a prefix of Q as a label is looked up. Since the label of lf is not known in advance, all prefixes of Q must be looked up via the DHT. If the outcome of the lookup for a prefix of Q is an internal node, the labels of its children are checked. If one of them is a $NULL$ leaf, then its prefix is of the form $Q'/NULL$, where Q' is a prefix of Q. By Claim 1 and Property 2.1, Q' subsumes Q, thus the pointer associated with Q' is followed to the peer caching it. If the outcome of a lookup for a prefix is a leaf, then a check is performed whether the leaf indexes Q or any other query subsuming it. In any other case, the lookup fails. The query lookup algorithm for IndexCache is shown in Algorithm 1.

Claim 3. *The lookup operation for query Q will always locate an answer for Q if such an answer exists in cache.*

Proof. There are two cases: (i) Q is cached, or (ii) a query $Q\prime$ which subsumes Q is cached. In case (i), by Claim 2 and Property 2.1, there exists a unique leaf lf where Q is indexed and lf's label is either a prefix of Q or $Q\perp$. The lookup operation initially looks up, via the DHT, $Q\perp$ and next each prefix of Q, thus it will finally locate lf. In case (ii), according to Claim 2 and Property 2.1, $NULL$ leaves index queries that are labels of their parents. Thus, during lookup, when an internal node with a $NULL$ leaf is located, the query indexed in this leaf subsumes Q. For a $NULL$ leaf to index queries subsuming Q, the label of that leaf's parents must be a prefix of Q. Consequently, by looking up all prefixes of Q, the lookup operation locates all the leaves indexing queries that subsume Q until an answer is found. A query subsuming Q may also be indexed in non-$NULL$ leaves. From Claim 2 and Property 2.1, there exists only one non-$NULL$ leaf whose label is a prefix of Q, consequently all queries that subsume Q (are prefixes of Q) are indexed in this leaf. Again as in case (i) this leaf is located by the lookup operation by looking up all the prefixes of Q. $\square$

Caching a New Query In IndexCache, when caching a query Q, a peer p stores the results of Q in its local cache and inserts Q to the trie index. Q must be indexed at the unique leaf lf having a prefix of Q as a label. Leaf lf is located by looking up all prefixes of Q. This lookup may have three outcomes: locate a leaf or an internal node having Q as a label, or none of the above. If a leaf is located, Q is inserted there. If an internal node having Q as a label is located, a $NULL$ leaf is created and Q is inserted in it. If none of the above two cases holds, a new leaf is created. The label of the new leaf is obtained as follows: Let nd_{max} be the node with the longest label located during the lookup for prefixes of Q, having a label of length d. The label of the new leaf becomes the prefix of Q of length $d+1$, and the parent of the leaf becomes node nd_{max}. The new leaf records the label of nd_{max} and vice versa.

When inserting a new query in a leaf, the total size of queries indexed at it may exceed C. In this case, the leaf must be split in two or more leaves having size not greater than C. Specifically, let x be the length of the label of the leaf and m the position of the first element at which any two queries indexed in this leaf differ. $x - m - 1$ internal nodes are created with labels of lengths from x to $m - 1$. The new leaves are assigned labels corresponding to prefixes of the cached queries consisting of m elements. The old leaf is deleted and the queries indexed in it are redistributed among the new leaves according to Property 2.1. The new leaves/nodes record the labels of their parent/children. It can be shown (proof omitted due to space limitations) that:

CLAIM 4. *The operation of caching a new query described above is consistent to the trie labeling definition and preserves Property 2.*

Algorithm 1 Cache index serial lookup.

```
1: $node := DHTlookup(Q/NULL)$
2: if $node$ is a "$NULL$" leaf then
3:    if IndexCache then
4:       SearchInLeaf($node$)
5:    else if DataCache then
6:       $nd$ := DHTlookup(leaf's parent label)
7:       SearchSubtrie($nd$)
8:    end if
9: end if
10: for $i := L$ to 1 do
11:    $node$ := DHTlookup($P_i(Q)$)
12:    if $node$ is a leaf then
13:       result := SearchInLeaf($node$)
14:       return result
15:    else if $node$ is internal then
16:       if exists leaf child labeled $Q/NULL$ then
17:          $lf$ := DHTlookup($Q/NULL$)
18:          result := SearchInLeaf($node$)
19:          if result $\neq$ NULL then
20:             return result
21:          end if
22:       end if
23:    end if
24: end for
25: return NULL
26: // $L$ is the number of elements in $Q$
27: // $P_i(Q)$ is the prefix of length $i$ of $Q$
```

3.2.2 DataCache Operations

In DataCache, along with the indexed query, each leaf of the trie also stores the fragment answering it. There is no distinction between local and global cache hits, since all queries must be looked up at the DHT, so that the appropriate leaves and thus peers storing relevant results are located.

As opposed to IndexCache, in DataCache, there is tight control over the distribution of cache content, and thus, it is possible to avoid redundancy caused by subsumption among queries. Assume that a new query Q that subsumes some query $Q\prime$ is inserted in cache. If Q is inserted in the same leaf lf as $Q\prime$, then $Q\prime$ and the fragment answering it are deleted from the leaf and replaced by Q and its associated fragment. This does not involve any restructuring of the trie.

Consider however, the case in which the outcome of the lookup for Q is an internal node nd. According to trie labeling and Property 2.1, Q is a prefix of all queries stored in the subtrie rooted at nd, and hence subsumes them. Thus, the set $\hat{F}$ of fragments stored in the subtrie rooted at nd is fully contained in F. There are two options for the actions to be taken for preserving Property 1: (a) merge the subtrie rooted at nd into a leaf and store there Q and the fragment F, and (b) create a new $NULL$ leaf as a child of nd with label $Q\perp$ and store the sub-fragment $F - \hat{F}$ there. The leaves in nd's subtrie record the fact that parts of F are stored in the subtrie rooted at nd. The basic advantage of option (a) is that subsequent queries for Q are found in the single leaf storing F, whereas, in option (b) all leaves in the subtrie rooted at nd must be accessed, since they all store part of the answer. On the other hand, the basic advantage of option (b) is that fragment sizes are kept small, helping their better distribution among the peers in the DHT, thus improving cache performance. We evaluated experimentally both options and have concluded that keeping the fragments small overweights the additional cost for lookups and thus we have adopted option (b).

It is easy to show using Claim 1 and Property 1.1 that:

CLAIM 5. *The operation of caching a new query Q in DataCache retains Property 1.*

The query lookup is similar to that in the case of IndexCache, except when Q is stored in a $NULL$ leaf (i.e. in a subtrie of the leaf). In this case, initially a DHT lookup is performed for $Q\perp$. If such a $NULL$ leaf is located, it means that the trie rooted at the leaf's parent contains parts of the fragment that answer Q. Thus, all the leaves in the subtrie are visited to produce the final answer.

3.3 Prefix Lookup Alternatives

Prefix lookup is central for both IndexCache and DataCache. There are three alternatives for looking up a query Q of length L. The first (SP) is looking up all prefixes of Q in parallel. The second one (SS) is looking up all prefixes of Q sequentially, starting from the longest one. The third technique (BI) is to use binary search on the prefix lengths of query Q. Specifically, if the currently located prefix is an internal node, lookup proceeds with longer prefixes. Otherwise, if the current prefix is neither an internal node nor a leaf, lookup proceeds with shorter prefixes.

We compare each technique based on: the number of hops required for locating an answer (Hops), the time required for locating an answer (Response Time), the load each technique induces on the peers maintaining the trie (Load), and the ability of each technique to locate a result stored/indexed in a live leaf regardless of the failure of any other trie node (Fault Tolerance). The comparison results are summarized in Table 1. Each prefix lookup in the DHT takes $log(N)$ hops, where N is the number of peers. For estimating the response time, we assume that each lookup takes t units at most. Regarding load balancing, in SP, each lookup for Q issues DHT lookups for all its prefixes. This leads to overloading peers at the upper levels of the trie, since they are visited for more queries. The other two techniques treat all nodes equally, thus it is expected that they achieve better load distribution in general. Regarding fault tolerance, assume that the result being looked up is stored/indexed in leaf lf. Suppose that lf is alive, while an internal node nd in the path from the root of the trie to lf has failed. If during binary lookup performed by BI, nd's prefix is looked up, the lookup will conclude that there does not exist a leaf or internal node labeled nd. Therefore, the lookup will proceed with a shorter prefix, failing to locate the live lf. Both SP and SS do not rely on internal nodes to lookup the query result.

Table 1: Comparison of SP, SS and BI.

	Hops	Response Time	Load Distr.	Fault Toler.
SP	$\Theta(L \cdot log(N))$	t	Poor	YES
SS	$O(L \cdot log(N))$	$O(L \cdot log(N) \cdot t)$	Good	YES
BI	$O(log(L) \cdot log(N))$	$O(log(L) \cdot log(N) \cdot t)$	Good	NO

3.4 Other Queries

In this section, we extend the basic cache operations, i.e. cache lookup and caching a new query, to handle more general XPath queries. The basic idea is for each such query Q to index/store both the resulting fragments using their corresponding path as index and a description of Q. We consider general queries in which the descendant axis ('//') or the wildcard ('*') may appear any number of times. Let $Q = P//R$ (or $Q = P/*/R$) be such a query, where P is a query containing only the child axis or the empty string and R is any general query.

Let us first consider query caching. The result of Q consists of a set of m fragments described by the path expressions P/F_i, $i = 1,..,m$, where F_i are expressions containing only the child axis. We call the expressions P/F_i *component* queries of Q. To cache Q, we cache its component queries as described previously, since they are linear path queries. Specifically, first, we perform a lookup for P, which is the common prefix of all the component queries. Let the outcome of looking up P be node n_q. Note that if P is the empty string, the lookup will locate the root node of the trie which has the empty string as a label. If n_q is a leaf, the component queries are cached in n_q, since P is a prefix of all of them. Analogously, if n_q is an internal node, the component queries are cached at the subtree rooted at n_q. In both cases, node n_q records the fact that Q is cached at it or at its subtrie. If n_q was a leaf and at some later point, it gets split, the information is passed to the new internal node having P as a label.

For looking up a query $Q = P//R$ ($Q = P/*/R$), initially a lookup for P is performed. If the outcome of the lookup, i.e. n_d, is a leaf, then it is checked for a record of caching Q or a query subsuming it. If n_d is an internal node which has a record of Q or a query subsuming it being cached in the subtrie, a broadcast is performed. For binary lookup, if n_d does not have such a record, the path from n_d's parent (if n_d is a leaf) or from n_d (if n_d is an internal node) to the root is traversed until a record for a query subsuming Q is found. For serial lookup, the internal nodes in the path to the root are examined during the lookup for P, thus the additional traversal of the path is not required.

The lookup cost, for a query containing the descendant axis cached in a leaf is $L_P \cdot log(N)$, where L_P is the length of P. If the query is cached in a subtrie then the cost depends on L_P, the smaller L_P, the higher the subtrie rooted at the node having P as a prefix and the more internal nodes and leaves will the broadcast visit.

4. CACHE REPLACEMENT

We assume that each peer p participating in cooperative caching offers some fixed amount of storage, C_p, for caching. With IndexCache, most of this storage is used by each peer p for caching the results of its own queries, while only a small portion is handled by the DHT for storing the part of the trie index that is assigned to p. With DataCache, the total storage C_p of each peer p is used by the DHT for storing the nodes and leaves of the trie along with the cached fragments associated with the leaves that are assigned to p. We assume that the storage required for maintaining the index is small and replacement decisions are needed only when caching the actual result fragments of each query.

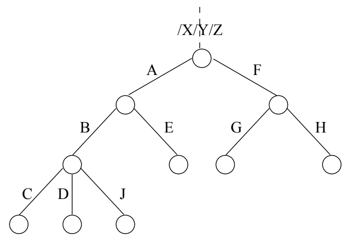


Figure 2: Example of an XML fragment cached as the result of query $/X/Y/Z$.

In particular in the case of DataCache, a number of trie leaves are assigned to each peer p. Let us assume a maximum leaf capacity of C. At each time instance, let l_p be the number of leaves assigned to peer p and L_i ($L_i \leq C$), $1 \leq i \leq l_p$, be the current size of each leaf assigned to p. We make no further assumptions about the number of leaves assigned to each peer p other than that: $\sum_{i=1}^{l_p} L_i \leq Cp$. This allows a dynamic organization of storage among the leaves assigned to each peer, since, depending on the query workload, each peer may have leaves with varying sizes L_i. Furthermore, peers may have different numbers of leaves assigned to them.

4.1 Replacement Policy

When caching a new query, the maximum capacity C_p offered by a peer may be already occupied by previously cached queries. In this case, one or more of these queries must be evicted from cache to make space for the new one. To this end, every time a cached query is used by a peer for answering a query, its utilization value, UV, is updated.

Additionally to answering queries that are the same with the cached ones, cached fragments are also used for answering queries which are subsumed by those cached. Hence, access statistics for *parts* of a cached fragment are also maintained. Therefore, when replacement is needed, individual paths of the fragment answering a query are also considered as candidates for replacement, besides the whole fragment. Such a partial replacement strategy was shown to be beneficial for caching XPath queries [7]. For example, consider Figure 2 that represents the fragment answering the cached query $Q = /X/Y/Z$. Subsequent queries $Q_1 = /X/Y/Z/A/B$ and $Q_2 = /X/Y/Z/F/G$ will be answered using Q, since Q subsumes both Q_1 and Q_2. In this case, the UVs for the paths $/X/Y/Z/A/B/C$, $/X/Y/Z/A/B/D$, $/X/Y/Z/A/B/J$ and $/X/Y/Z/F/G$ are updated.

In the case of IndexCache, cache overflow is caused, when after a cache miss of some local query Q, a peer p whose local cache has reached its maximum capacity C_p, attempts to cache the result fragment of Q. In this case, all paths in the local cache of p are examined and the one with the smallest UV among them is chosen for replacement. This procedure is repeated until enough space is created for the new fragment.

Cache replacement is more involved in the case of DataCache. Cache overflow may be created when either (i) a fragment is assigned to a leaf already stored at p or (ii) a new leaf is assigned to p as a result of some split operation. In case (i), if there is space in the leaf (that is the current size of the leaf is smaller than C) but the peer has reached its full capacity, we check the UV values of all fragments of all leaves stored at p and evict those with the smallest UV values. This allows for a better cache utilization, since leaves get to replace their inactive queries. If, however, the leaf has reached its maximum capacity C, the leaf is split. In case (ii), if peer p has enough capacity, it stores the leaf. If the peer has not enough capacity, it again searches for victims among all entries of all leaves. Note that in this case, even entries at the new leaf are potential candidates for replacement, since the leaf is a result of splitting and its entries may have small UV values.

4.2 Cache Operations Revisited

Partial replacement creates the need for some modifications of the basic cache operations. For example, suppose that the paths $/X/Y/Z/A/B/C$ and $/X/Y/Z/A/B/D$ of Figure 2 are replaced. The resulting fragment does not answer query $/X/Y/Z$ anymore, thus the distributed index must be updated, so that the index entry for query $/X/Y/Z$ includes some information about the replaced paths. The entry is deleted only if all paths of the fragment associated with it are replaced.

Subsequent lookups for a query Q whose fragment has become partial or for queries subsumed by it must determine, through the information for the replaced paths, whether the remaining portion of the fragment is sufficient to produce an answer. Otherwise, we have a cache miss; the answer is looked up in remote data sources and cached. Caching results of such queries differs from that described in Section 3.2. For DataCache, to avoid fragment redundancy, the overlapping portions are stored only once. Specifically, after a cache miss for Q, the new fragment to be cached replaces the old, partial one. After a cache miss for a query $Q\prime$ subsumed by Q, $Q\prime$ is not indexed. Instead the fragment answering it updates the fragment associated to Q by filling the space of the replaced paths. Obviously, this change in caching new queries retains Property 1. For IndexCache, the same actions are performed, when the query being inserted is posed by the same peer caching the partial fragment, otherwise the operation remains unchanged.

4.3 Proactive Replacement

When a specific leaf overflows, it is split, regardless of the UVs of the paths stored/indexed in it. To avoid the overhead of unnecessary splitting leaves containing stale (with low UVs) paths, for DataCache, we propose their proactive replacement. Before performing a split, each peer checks the leaves it stores for paths that can be evicted based on their UVs. To this end, we define a UV threshold such that any path having a UV lower than this threshold will be replaced. The threshold, τ, is defined locally by each peer as a percentage over the average (UV_{avg}) of all the UVs of the paths stored at a peer. Paths having UVs lower than the τ% of UV_{avg} are proactively evicted from cache.

The value of the threshold determines the frequency of proactive replacement: the higher the threshold, the more aggressive the proactive replacement. If the utilization values of all the paths at a peer are similar, then no proactive replacement will be performed. This is desirable when all UVs are high, but not when all UVs are low. To make proactive replacement possible in the second case, the UV_{avg} is increased by a factor f every time proactive replacement is performed in the peer.

5. EXPERIMENTAL EVALUATION

We have implemented both the DataCache (DC) and IndexCache (IC) cooperative cache frameworks. Caches are built on top of a simple simulator of CHORD, over which the distributed prefix trie is constructed. For the purposes of our experiments, we used a number of synthetic tree-structured XML documents of arbitrary complexity produced using the Niagara-Project XML Data Generator [2]. We have also implemented a simple parser which extracts all root-to-leaf paths from an XML document and an XPath query generator which creates queries with different lengths along with their results based on the previously extracted paths. The input parameters are summarized in Table 2.

Table 2: Input parameters.

Parameter	Default Value	Range
# of peers	100	20 - 996
Ring size	128	128 - 1024
Peer cache size	500 Kb	10 Kb - 8 Mb
Leaf size - DC	25 Kb	580 bytes - 183 Kb
Leaf size - IC	400 bytes	40 - 12800 bytes
Query length	8 elements	5 - 8 elements
Avg. entry size - DC	571 bytes	571 - 15417 bytes
Avg. entry size - IC	40 bytes	20 - 40 bytes
Merge coefficient(k)	1/2·leaf size	-
Peer activity rate	Uniform	zipf ($\alpha = 0.4$ - 1)
Local vs. global overlap	No distinction	0-100%
Query overlap	80%	10-90%

We use a relatively small number of peers, since our scheme is expected to involve peers in close network proximity with each other. All our experiments start with an empty (cold) cache. Regarding the lookup procedure, we use the binary lookup algorithm unless stated differently.

5.1 Cooperative vs. Individual Caching

In this set of experiments, we evaluate the benefits of cooperative caching over individual caching. With individual caching, peers locally cache the results of their queries, but do not share them with any other peer. We consider individual caching with a 50% hit ratio. Cooperative caching improves the hit ratio, when there is overlap among the peer queries. If at each step, we increase the overlap between the query workloads of the peers by 10%, starting from 10% up to 40%, our results confirm (figures not shown due to space limitation) that the cache hit ratio increases linearly with this overlap for both DC and IC from 50% (no overlap, only local hits) up to 90% (50% local hits and 40% hits from the cooperative cache). There is a similar linear dependency with the number of peers sharing their caches. In particular, we keep the overlap among the query workloads constant and build cooperative caches starting from 25 peers up to 100 with a step of 25. Again, the hit ratio increases linearly from 50% when there is a single peer up to 90% when we have all 100 peers participating in the cache.

However, cooperation comes with a cost. Cooperative caching is built on the assumption that locating and transferring data among the peers forming the cooperative cache is less expensive than locating and transferring data from remote data sources. Locating data from remote sources involves procedures varying in complexity from exhaustive search to using specialized directories, which we expect to be less efficient than locating data in the cache. Thus, here we only consider the cost of transferring data. To quantify this, we use the ratio $\rho = B_C/B_I$, where B_C is the amount of data transferred for answering queries from the cooperative cache and B_I the corresponding amount of data transferred when the same queries are answered from remote hosts. B_C includes constructing and maintaining the cooperative cache, initially transferring the results from the remote sources, and transferring data among peers participating in caching. B_I is the cost of transferring the results from the remote host, once for each peer posing the query. Our results (Figure 3(left)) show that as the query overlap increases, ρ decreases in both approaches, since the hit ratio increases. In particular, if the cost of transferring a bit from a remote host is more than 3.15(1.1) times higher than transferring it from a peer in the cache, it is cost effective to build a DataCache (IndexCache), even for a 10% query overlap.

5.2 DataCache vs. IndexCache

We compare the performance of DataCache (DC) with the performance of IndexCache (IC) in terms of: (a) the cache *hit ratio*, (b) the *hop lookup cost* (hLC): the number of hops needed for locating the leaf of the distributed prefix trie that stores/indexes the results of the query, (c) the *network lookup cost* (nLC): the network bandwidth needed for transferring the final result fragment of a query to the peer that has posed the query in the case of a hit, (d) the *maintenance cost*: the overall network bandwidth required for maintaining the cooperative caches including the network bandwidth for performing leaf split/merge during inserting/deleting queries in cache, and (e) the *query load*: the number of queries answered by each peer.

Hit Ratio The performance of DC depends on the size of each trie leaf. We relate the size of the leaf with the size of the queries. In particular, we assume that each query must "fit" inside a leaf. Thus, we take the lower value of the leaf capacity to be equal to the size of the larger query. We vary the leaf capacity starting from this value and increasing it by 50% at each step. Our results (Figure 3(middle)) show that, while, as expected, the leaf capacity has no impact on the hit ratio for IC, an increase of the leaf capacity results in a decrease of cache hit ratio for DC. In IC, cached fragments remain in local caches thus their distribution is only influenced by the peer query workload, while in DC, the distribution of cached data depends on the underlying DHT and the distributed prefix trie. Smaller capacity leaves mean more splits thus better distribution of cached data among peers. Consequently, peer caches grow more uniformly, leading to fewer replacements, hence a higher cache hit ratio.

The main factor that influences the hit ratio of DC versus IC is the query rate imbalance among peers. Intuitively, in the case of IC, overactive peers may generate disproportionally more queries and thus influence the content of local caches more than less active ones. To quantify that, we created query workloads where the portion of queries assigned to each peer is generated according to a Zipf distribution $1/a^{id}$ where *id* is the *id* of each peer in the CHORD ring and parameter a determines the degree of skew. We vary a from 0.4 to 1. Our results (Figure 3(right)) show that when the query distribution is uniform, DC and IC have the same hit ratio. As the skew increases, the hit ratio for IC decreases due to the high number of replacements occurring at the most active peers. These replacements cause valuable data to be evicted from cache making the hit ratio for IC up to about 22.4% lower than that for DC when $a = 1$.

Lookup Cost In the IC case, local hits are much cheaper than global ones, since global hits require transferring results from other peers. In this set of experiments, we vary the percentage of local overlap in the queries posed by each peer, i.e. the portion of queries whose answer is subsumed by queries previously posed by the same peer from 0% (no overlap) to 100% (complete overlap). In IC, such results are cached in the local cache of the peer. Our results (Figure 4 (left)) show that for IC as the local overlap and thus the local hit ratio increase, hLC and nLC decrease analogously, while for DC, they remain constant. This is expected since in IC, the more queries overlap locally for a peer, the more answers will be found in its local cache, thus, the fewer times the distributed index will be accessed (less hops) and the fewer results will be transferred from other peers' caches (less bandwidth).

Maintenance Cost In IC, the maintenance cost involves just managing the distributed index, whereas in DC, the cost includes also moving cache fragments, thus, it is considerably larger. In both cases, the number of maintenance actions (splits/merges) mainly depends on the leaf capacity. We vary the leaf capacity from 1142 bytes to 73088 bytes for DC and from 80 to 5120 for IC by doubling the leaf capacity at each step, corresponding to a variation from 1 to 64 queries indexed/stored per leaf. Our results (Figure 4 (middle)) show that the maintenance cost for DC is $\approx$14 times higher than that of IC because the sizes of the leaves that are split or merged are also $\approx$14 times larger ($1142 \div 80$) than those in IC. Furthermore (Figure 4 (right)), for DC the bandwidth consumed for maintenance constitutes $16.9 - 33.25\%$ of the total bandwidth, while for IC only $2.6 - 6.1\%$. In general, the increase in leaf capacity reduces the number of splits (which in this setting is equal in each step for DC and IC), however this depends on the workload which explains the fluctuation in the reported results.

Query Load Distribution While in IC cached fragments remain in local caches, in DC the distribution of cached data among peers is performed by the distributed prefix trie and the underlying DHT. Thus, for IC, the query load that a peer receives depends solely on the popularity of the fragments that the peer caches locally, while for DC, it depends on the popularity of the trie leaves assigned to it by the DHT. To show this, we design an experiment for the extreme case, where one peer queries and then caches a number of popular fragments which are then queried by the other peers in the system. Since the distribution in DC depends on the leaf size, we experimented with two leaf sizes, the second one twice the size of the first one. Our results show (Figure 5) that for IC, the peer holding the popular fragments receives all queries for them producing a significant load, while the remaining peers answer $96.35\% - 91.47\%$ less queries. In DC, for the smaller leaf size the load of answering the popular queries is distributed among all peers, so that the most loaded peer answers 84% less queries than in IC. As the leaf size increases, the load distribution becomes more skewed: a smaller number of peers share most of the query load, but still achieving a better distribution than IC.

5.3 Replacement

Local vs. Global Replacement Decisions In IC and DC, replacement decisions are made based on the local view of each peer. In IC, this view includes the results of its own queries and in DC, the part of the query space assigned to it through the DHT. However, the smallest local UV may not be equal to the globally minimum UV. To investigate the effect of this fact, we compare the performance of DC and IC with that of a central cache, where replacement decisions are made based on the UV values of all queries. Specifically, we implemented a central cache with size equal to the sum of the sizes of caches of all peers in DC or IC. The performance metric in these experiments is the cache hit ratio with varying leaf capacity (DC) and skew (IC). Results (Figure 6(left) and (middle)) show that in the best case (smaller leaf capacity - 0.4 skew), the central cache has ≈7.66% better performance than both DC and IC and as leaf size - skew increase the difference increases up to 30%. Thus, local replacement decisions do not significantly decrease the performance of both approaches, for a good choice of leaf capacity and reasonable activity rates.

Proactive Replacement As already shown, the leaf capacity significantly influences the performance of DC. To keep the leaf capacity small and at the same time avoid the maintenance cost, we have proposed proactive replacement. We performed two sets of experiments which evaluate the impact of the replacement threshold τ on the cache hit ratio and on the network bandwidth overhead caused by leaf splits. The effects of proactive replacement are more evident, when there is a shift of interest in the workload, thus popular items become unpopular and vice versa. To model this shift of interest, after posing a set of queries for a period of time, the "topic of interest" of peers changes. As shown in Figure 6(right), for $10\% \leq \tau \leq 30\%$ the cache hit ratio remains constant while the bandwidth significantly decreases (≈35% for $\tau = 30\%$). Increasing τ further causes replacement of needed paths decreasing the hit ratio. Furthermore, the number of merges increases leading to further splits thus reducing the decrease of the bandwidth.

5.4 Design Alternatives

NULL Leaves vs. Merging We evaluate for DC, the factors that influence the choice between creating a "NULL" leaf (NL) and merging the subtrie (MS) when caching a query that subsumes the queries in the subtrie defined by the query. These factors are the query *lookup cost*, i.e., the number of hops needed for locating the leaf of the distributed prefix trie that stores/indexes the results of the query in the case of a hit, and the cache *hit ratio*.

LOOKUP COST With MS, only one leaf is visited for answering the subsuming query, while with NL, all leaves in the subtrie must be visited increasing the lookup cost. This is confirmed by our experimental results (Figure 7(left)), where as the height of the subtrie increases, so does the lookup cost for the broadcast.

HIT RATIO NL keeps smaller fragments in more leaves allowing for their better distribution between peers. Consequently, peer caches grow more uniformly, leading to fewer replacements, hence a higher cache hit ratio. To evaluate the above intuition, we measured the cache hit ratio for different percentages of subsumption ($10\% - 50\%$). i.e. 10% subsumption means that 10% of the cached queries are stored in subtries subsumed by a new query. Results (Figure 7 (middle)) show that the hit ratio for MS is always smaller than that in NL. The difference increases as the percentage of the subsumption increases. This is expected since the increase of the subsumption corresponds to an increase in the number of leaves created by the merge, consequently to more replacements due to poor fragment distribution.

Binary vs. Serial Lookup Figure 7 (right) shows the average load of the nodes for each level of the trie during serial parallel (SP) lookup and binary (BI) lookup. Serial lookup visits the upper level nodes more often, thus the peers that hold the upper level nodes receive much higher load than those storing the lower levels.

5.5 Scalability

We evaluate the scalability of the cooperative cache with regards to (a) the cache hit ratio, (b) the lookup cost (number of hops) and (c) the maintenance cost. For different number of peers, we consider different ring sizes for the DHT as follows: for 50 and 100 peers the ring size is 128 (2^7), for 114 and 228 peers the ring size is 256 (2^8), for 242 and 484 peers the ring size is 512 (2^9) and for 496 and 996 peers the ring size is 1024 (2^{10}). The above numbers are chosen so that for each ring size (e.g. 2^7) the ring is half-full (e.g. 50 peers) and almost full (e.g. 100 peers). By doing so, we also show how the density of the ring influences the cache performance. All participating peers pose an equal number of queries, thus, as the number of peers increases, the overall number of queries increases accordingly. Each peer selects its queries from a large set of queries, using a zipf distribution with $\alpha = 1$. This creates a global query overlap.

Hit Ratio For both the IC, (Figure 8 (left)) and the DC (Figure 8(right)), the hit ratio increases with the number of peers, since the more peers in the cache, the more storage space is offered for caching the common, frequent queries posed by the peers. Observe that, half-full rings in DC have a lower hit ratio than full rings with about the same number of peers. This is due to the fact that the more peers in the ring, the less ring positions are assigned to each peer by the DHT and consequently, a better distribution of fragments is achieved.

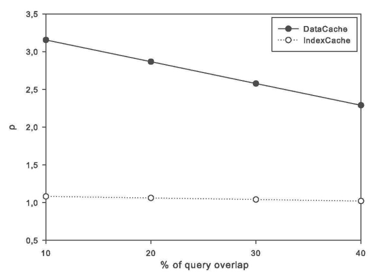
Lookup Cost As expected, the average number of hops required for answering a query increases with the ring size for both DC and IC logarithmically (Figure 8 (middle)).

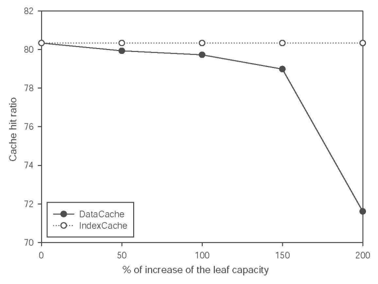
Maintenance Cost We measure the average maintenance cost per peer. As expected, for any number of peers the maintenance cost for DC is higher than that for IC (Figure 8 (right)). For both IC and DC the maintenance cost per peer drops fast as the number of peers increases (e.g from 100 to 996 peers it drops by ≈ 90%), because of the related increase of the hit ratio. A higher hit ratio means that fewer new fragments are cached and thus fewer splits and merges are generated.

5.6 Result Summary

We summarize below, some of the key findings of our experimental evaluation:

1. Cooperative caching significantly increases the cache hit ratio compared to individual caching, if there is overlap in the query workload. Depending on this overlap, the benefits of sharing surpass the maintenance overheads for reasonable ratios of remote vs. local network bandwidth.





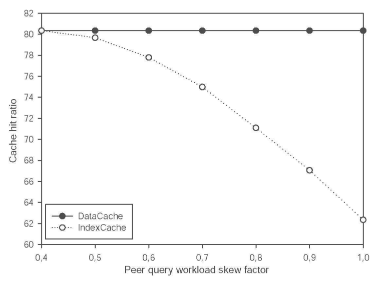
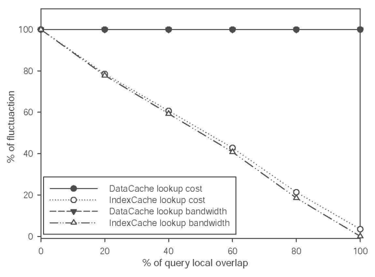
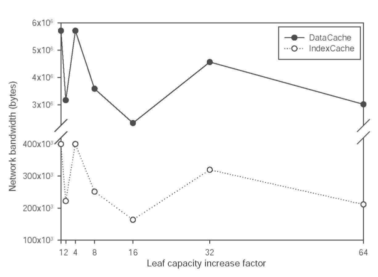


Figure 3: (left) ρ with the overlap among peer query workloads. Cache hit ratio for different leaf capacity (middle), and for different levels of skew in the peer workload activity (right).





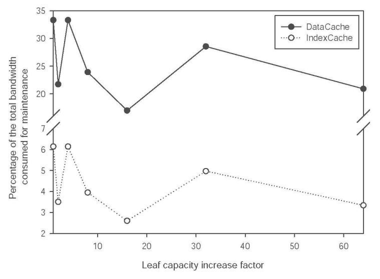
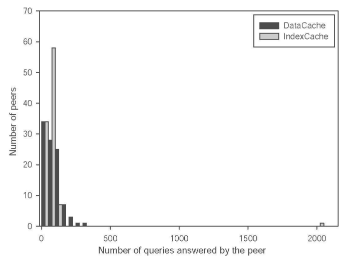


Figure 4: Fluctuation of hop and network lookup with the percentage of local overlap (left). Maintenance cost with leaf capacity (middle). Percentage of network bandwidth for maintenance with leaf capacity (right).



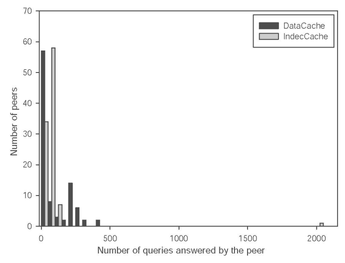
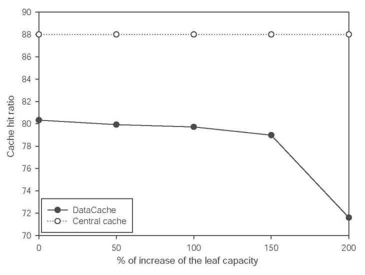
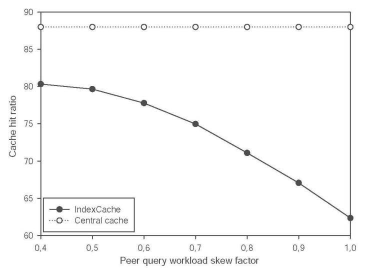


Figure 5: Distribution of the queries answered by the peers (left) and for doubling the leaf size (right).





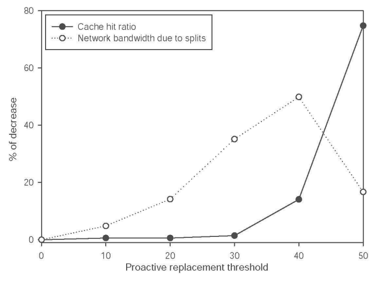


Figure 6: Comparison of hit ratio: central cache vs. DataCache (left) and vs. IndexCache (middle). Decrease of cache hit ratio and the overhead caused by splits for increasing proactive replacement threshold (right).

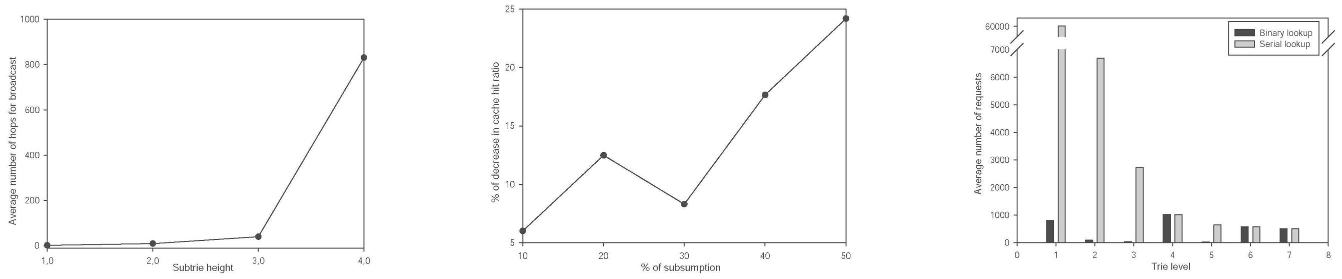


Figure 7: Avg. lookup cost for broadcasting a query in NL (left). Hit ratio for NL and MS for increasing query subsumption (middle). Avg. number of requests served by each peer during lookup (right).

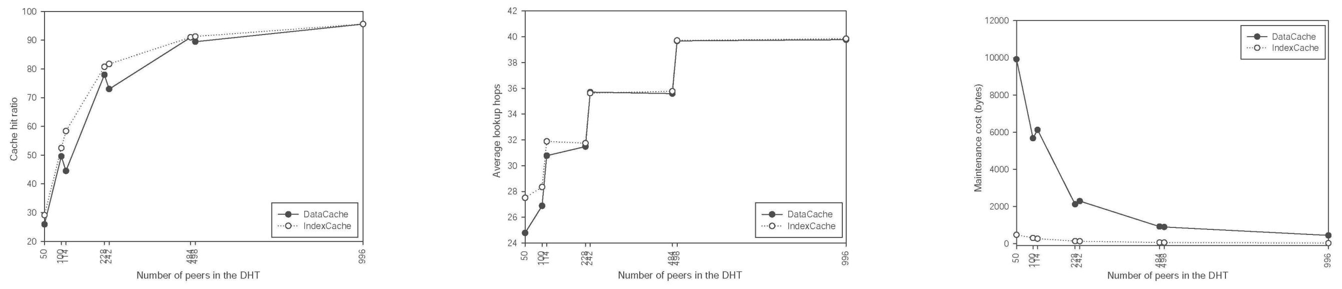


Figure 8: Cooperative cache hit ratio (left), average lookup hops (middle) and averaged maintenance cost for increasing number of peers participating in the DHT (right).

2. DataCache achieves higher hit ratios than IndexCache, when there is substantial query imbalance among the peers. DataCache performance depends on the number of peers in the DHT and the leaf capacity: a small leaf capacity increases the cache hit ratio and achieves a significantly better query load balance than IndexCache, but also increases the maintenance cost since it increases the number of maintenance operations. In general, the maintenance overhead of DataCache is more than 10 times that of IndexCache. IndexCache supports more efficient lookups when there is high intra-peer locality and relatively low inter-peer locality.

3. Replacement decisions based on the local view of each peer do not significantly deteriorate cache performance compared to a global cache, under reasonable conditions. Proactive replacement decreases the maintenance cost in DataCache without affecting its hit ratio, when there is a change of popularity in the query workload.

4. Caching the difference between a query and the set of already cached queries that subsume it versus merging these queries increases the lookup complexity, while it increases the cache hit ratio. These effects are highly correlated to the height of the subtrie merged.

5. Cooperative caching is scalable with the number of peers as long as there is workload locality. For DataCache, the size of the DHT should be close to that of the peers participating in the cache.

6. RELATED WORK

In this paper, we present a framework for cooperative XML caching. XML caching in a typical client-server environment is discussed in [7, 14, 19]. XML caching, as related to view materialization and view subsumption, has been the focus of much current research. Here, we adapt a practical approach that uses XPath expression and the associate fragments as the unit of caching. This reduces answerability problems to simple prefix matching.

In terms of building a cooperative cache, perhaps the work most related to ours is Squirrel [11]. In Squirrel, peers in a corporate LAN environment are organized in a DHT to form a cooperative web cache. The URL of each document is mapped to a peer in the DHT, which is called *home-node*. Two approaches are proposed for what the home-node stores. In the *home-store* approach, the cached document is stored both in the local cache of the peer and in the home-node. In the *directory* approach, peers locally cache the results of their HTTP requests and the home-node only stores pointers to these peers. Directory resembles our IndexCache, while home-store resembles a hybrid combination of IndexCache and DataCache. Besides the difference between home-directory and IndexCache-DataCache, the main distinction of our work with regards to Squirrel stems from the fact that we built a semantic cache. Thus, our unit of caching is a query result as opposed to a document. Further, there is overlap between results. These lead to the need of building an index (e.g. a prefix trie) that allows checking for subsumption and providing more involved cache operations.

In [16], a cooperative cache is also maintained but for supporting full text document search in P2P. Each peer locally stores query results. An index is built over a DHT, by indexing the cache content of each peer through randomly selected terms of the cached queries. The focus is on determining which multi-term queries to cache to avoid intersecting large inverted lists.

There is some work in the context of distributed processing of XML documents. [13] investigates the organization of peers in an unstructured super-peer network for maintaining a distributed index of XML data. Peers are clustered based on their content which is summarized by multi-level bloom filters. This approach is not appropriate for indexing cached XPath queries, since caches and thus content, changes dynamically. In [17], a distributed index for XPath queries is proposed. The index is based on a distributed binary tree and queries are routed based on their prefixes. The main difference with our prefix trie is that [17] is used for indexing the document collections maintained by the peers, while our prefix trie indexes cache content, which significantly changes query processing. Additionally, in [17], P-Grid [1] is used as the DHT, which requires restructuring the overlay to adapt to the indexed dataspace. In a cooperative cache this would occur often, since the cached data changes dynamically according to the query workload.

Other approaches for DHT-based indexing for XML include [5] and [10]. The operations in [5] can not detect fragment overlap, thus if used for indexing the cooperative cache, they would create significant redundancy. The approach presented in [10] indexes XML paths in a DHT by using tag names as keys. A peer responsible for an XML tag maintains a summary with all possible unique paths leading to the tag. Thus, only one tag of a query is used to locate the responsible peer for exact match queries. Although ensuring efficient search and the possibility of looking up query prefixes, the approach introduces considerable overhead for popular tags whose summaries are large. Again, the difference with our work is that we index cached data instead of content, thus, we cannot ensure that the node responsible for a tag would have a summary of all possible documents (instead, it would know only of the cached ones).

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a framework for building a cooperative cache of XML documents. To facilitate sharing, a prefix-based distributed index of the cache content is maintained. Our main focus is on studying the two fundamental approaches to cache sharing. In the IndexCache approach, each peer caches results of its own queries and just publishes a description of the content of its local cache to the index. In the DataCache approach, each peer is assigned and stores a specific part of the query space. The partition of the query space is based on the prefixes of the cached queries. We have evaluated both approaches and identified the conditions under which one surpasses the other.

There are many issues for future work. A promising direction for future work is to investigate whether a prefix-based partitioning of the dataspace can be used for building a distributed native XML database. Considering different overlays for implementing the prefix-trie is also central. A basic reason for building the trie *on top* of a DHT is for achieving transparent distribution of the trie nodes among the peers of the DHT. As our results show, this is achieved. On the other hand, there have been proposals for tree-based overlays (e.g., PGrid [1], BATON [12]), for which, it could be possible to have “built-in” substring search. However, a direct mapping of trie nodes to nodes of such structures would introduce the need for restructuring the overlay itself and thus increase the maintenance cost considerably.

8. REPEATABILITY ASSESSMENT RESULT

All the results in this paper were verified by the SIGMOD repeatability committee.

9. REFERENCES

[1] K. Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In *CoopIS*, 2001.

[2] A. Aboulnaga, J. Naughton, and C. Zhang. Generating Synthetic Complex-Structured XML Data. In *WebDB*, 2001.

[3] S. Amer-Yahia, D. Srivastava, and D. Suciu. Distributed Evaluation of Network Directory Queries. *IEEE Trans. Knowl. Data Eng.*, 16(4), 2004.

[4] A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh. A Framework for Using Materialized XPath Views in XML Query Processing. In *VLDB*, 2004.

[5] A. Bonifati, A. Cuzzocrea, U. Matrangolo, and M. Jain. XPath Lookup Queries in P2P Networks. In *WIDM*, 2004.

[6] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, J. Hellerstein, and S. Shenker. A Case Study in Building Layered DHT Applications. In *SIGCOMM*, 2005.

[7] L. Chen, S. Wang, and E. A. Rundensteiner. A Fine-Grained Replacement Strategy for XML Query Cache. In *WIDM*, 2002.

[8] S. Dar, M. J. Franklin, and B. Jonsson. Semantic Data Caching and Replacement. In *VLDB*, 1996.

[9] A. Deshpande, S. K. Nath, P. B. Gibbons, and S. Seshan. Cache-and-Query for Wide Area Sensor Databases. In *SIGMOD*, 2003.

[10] L. Galanis, Y. Wang, S. R. Jeffery, and D. J. DeWitt. Locating Data Sources in Large Distributed Systems. In *VLDB*, 2003.

[11] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A Decentralized Peer-to-Peer Web Cache. In *PODC*, 2002.

[12] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: A Balanced Tree Structure for Peer-to-Peer Networks. In *VLDB*, 2005.

[13] G. Koloniari and E. Pitoura. Content-Based Routing of Path Queries in Peer-to-Peer Systems. In *EDBT*, 2004.

[14] B. Mandhani and D. Suciu. Query Caching and View Selection for XML Databases. In *VLDB*, 2005.

[15] G. Miklau and D. Suciu. Containment and Equivalence for a Fragment of XPath. In *J. ACM 51(1)*, 2004.

[16] G. Skobeltsyn and K. Aberer. Distributed Cache Table: Efficient Query-Driven Processing of Multi-Term Queries in P2P Networks. In *P2PIR*, 2006.

[17] G. Skobeltsyn, M. Hauswirth, and K. Aberer. Efficient Processing of XPath Queries with Structured Overlay Networks. In *ODBASE*, 2005.

[18] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *In SIGCOMM*, 2001.

[19] H. L. Yang, M. L. Lee, and W.Hsu. Efficient Mining of XML Query Patterns for Caching. In *VLDB*, 2003.