Cooperative XPath Caching

Η ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Πληροφορικής Εξεταστική Επιτροπή

από τον

Κώστα Λίλλη

ως μέρος των Υποχρεώσεων για τη λήψη του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ

ΣΤΟ ΛΟΓΙΣΜΙΚΟ

Ιούλιος 2007

Dedication

This thesis is dedicated to my nieces Ria and Anny...

Acknowledgments

I would like to thank my supervisor Professor Evaggelia Pitoura for guiding, encouraging and motivating me, and for the time spent during the elaboration of this thesis. Furthermore, I would like to thank my parents and my sisters who have supported me all the way since the beginning of my studies.

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Kostas Lillis. MSc, Computer Science Department, University of Ioannina, Greece. July 2007. Cooperative XPath Caching. Thesis Supervisor: Evaggelia Pitoura.

Motivated by the fact that XML is increasingly being used in distributed applications, in this thesis, 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 distributed 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 nodes 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 policy for cache replacement. We also report on a number of experiments that show the benefits of cooperative caching and highlight the differences in performance between loosely and tightly coupled cache sharing.

Εκτενης Περιληψη

Κώστας Λίλλης του Βάσου και της Ιφιγένειας. MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννινων, Ελλάδα. Ιούλιος, 2007. Cooperative XPath Caching. Επιβλέπουσα: Ευαγγελία Πιτουρά.

Η XML χρησιμοποιείται ευρέως στην ανταλλαγή δεδομένων στο διαδίκτυο. Με την αύξηση του μεγέθους και του αριθμού των XML αρχείων, η λανθάνουσα αποθήκευση των αποτελεσμάτων των ερωτήσεων για αυτά τα αρχεία μπορεί να παίξει σημαντικό ρόλο στη μείωση του υπολογιστικού κόστους και του κόστους επικοινωνίας που απαιτείται για την επεξεργασία αυτών των ερωτήσεων σε ένα κατανεμημένο σύστημα. Τα μοντέλα λανθάνουσας αποθήκευσης αποτελεσμάτων για XML βάσεις δεδομένων συνήθως ακολουθούν μια σημασιολογική προσέγγιση: τα αποθηκευμένα αποτελέσματα δεν οργανώνονται σε επίπεδο πλειάδων, σελίδων ή αρχείων αλλά σε επίπεδο ερωτήσεων. Συγκεκριμένα, τα αποτελέσματα προηγούμενων ερωτήσεων αποθηκεύονται στη λανθάνουσα μνήμη μαζί με την αντίστοιχη ερώτηση. Με την άφιξη νέων ερωτήσεων, το σύστημα εκτελεί μια αναζήτηση στη λανθάνουσα μνήμη για να καθορίσει αν η ερώτηση μπορεί να απαντηθεί με βάση τα αποθηκευμένα αποτελέσματα κάποιας παλιότερης ερώτησης, ή αν το αποτέλεσμα της ερώτησης υπάγεται από κάποιο αποθηκευμένο αποτέλεσμα.

Σε αυτή την εργασία εξετάζουμε το πρόβλημα της κατασκευής μιας κατανεμημένης, σημασιολογικής λανθάνουσας μνήμης για XML αρχεία. Ένα σύνολο κόμβων που βρίσκονται σε κοντινή δικτυακή απόσταση μεταξύ τους υποβάλλουν ερωτήσεις για αυτά τα αρχεία. Τα XML αρχεία βρίσκονται αποθηκευμένα σε ένα σύνολο απομακρυσμένων βάσεων δεδομένων. Έτσι το κόστος για των εντοπισμό και την μεταφορά των απαντήσεων στους κόμβους που υποβάλλουν τις ερωτήσεις αναμένεται να είναι αρκετά υψηλό. Για αυτό το λόγο είναι σημαντικό να μπορούν να ξαναχρησιμοποιηθούν τα αποτελέσματα των ερωτήσεων που έχουν υποβληθεί από κάθε κόμβο, για την απάντηση ακόλουθων ερωτήσεων όχι μόνο από τους ίδιους κόμβους αλλά και από τους άλλους κόμβους στο σύστημα. Αυτό κάνει ελκυστική την συνεργασία μεταξύ των κόμβων του συστήματος για την διατήρηση μιας κοινής λανθάνουσας μνήμης.

Θεωρούμε δύο βασικές προσεγγίσεις για την διατήρηση της κοινής λανθάνουσας μνήμης όσον αφορά τον βαθμό συνεργασίας μεταξύ των κόμβων. Στην προσέγγιση που ονομάζουμε IndexCache , κάθε κόμβος αποθηκεύει στην τοπική του λανθάνουσα μνήμη τα αποτελέσματα των δικών του ερωτήσεων. Για τον διαμοιρασμό του περιεχομένου των λανθανουσών μνημών των κόμβων κατασκευάζεται ένα κατανεμημένο ευρετήριο. Έτσι κάθε ερώτηση που υποβάλλεται από έναν κόμβο προσπελαύνει το ευρετήριο για να εντοπίσει κάποιο άλλο κόμβο που αποθηκεύει αποτελέσματα τα οποία μπορούν να απαντήσουν την συγκεκριμένη ερώτηση. Στην προσέγγιση που ονομάζουμε DataCache σε κάθε κόμβο ανατίθεται ένα συγκεκριμένο κομμάτι του χώρου των δεδομένων. Τα αποτελέσματα μίας ερώτησης δεν αποθηκεύονται πια στην τοπική λανθάνουσα μνήμη του κόμβου που την υπέβαλε, αλλά στην λανθάνουσα μνήμη του κόμβου που είναι υπεύθυνο για το αντίστοιχο κομμάτι του χώρου δεδομένων. Έτσι κάθε ερώτηση μπορεί να προκαλέσει την αναδιοργάνωση της αποθήκευσης των δεδομένων σε φυσικό επίπεδο. Αυτό συνεπάγεται ένα επιπλέον κόστος για την μετακίνηση των δεδομένων αλλά μειώνει τον αριθμό των αντιγράφων για τα αποθηκευμένα αποτελέσματα σε αντίθεση με την IndexCache .

Και στις δυο προσεγγίσεις η αποθήκευση είναι δυναμική. Έτσι, η οργάνωση των δεδομένων σε φυσικό επίπεδο αλλάζει δυναμικά: συρρικνώνεται και επεκτείνεται με βάση τις ερωτήσεις, καθώς και με βάση τον αριθμό των κόμβων και το διαθέσιμο αποθηκευτικό χώρο. Αυτό οδηγεί σε κάποια μορφή αυτό-οργάνωσης.

Τα δεδομένα τα οποία χρησιμοποιούνται συχνά για την απάντηση ερωτήσεων διατηρούνται στην λανθάνουσα μνήμη και μπορούν να εντοπίζονται μέσω του ευρετηρίου. Αντιθέτως τα δεδομένα που δεν χρησιμοποιούνται συχνά απορρίπτονται από την λανθάνουσα μνήμη και για την απάντηση σχετικών ερωτήσεων εκτελείται αναζήτηση στις απομακρυσμένες βάσεις δεδομένων.

Ένα σημαντικό θέμα είναι ο τρόπος οργάνωσης του ευρετηρίου στην περίπτωση της IndexCache ή του χώρου δεδομένων στην περίπτωση της DataCache . Όπως προαναφέρθηκε, η λανθάνουσα μνήμη είναι σημασιολογική, άρα η ευρετηριοποίηση γίνεται με βάση τις ερωτήσεις οι οποίες περιγράφουν το περιεχόμενό της. Για την ευρετηριοποίηση προτείνουμε την χρήση ενός ευρετηρίου βασισμένου σε προθέματα των ερωτήσεων, το οποίο εκμεταλλεύεται το XPath πρότυπο των ερωτήσεων. Για την κατανομή του ευρετηρίου ανάμεσα στους κόμβους του συστήματος, χρησιμοποιούμε μια προσέγγιση βασισμένη στους κατανεμημένους πίνακες κατακερματισμού (DHTs).

Ο φυσικός αποθηκευτικός χώρος της λανθάνουσας μνήμης είναι περιορισμένος, για αυτό κατά καιρούς, για την αποθήκευση νέων αποτελεσμάτων, απαιτείται η απομάκρυνση (αντικατάσταση) παλιών αποτελεσμάτων. Οι τεχνικές που εφαρμόζουμε για την αντικατάσταση, εκμεταλλεύονται τις σχέσεις υπαγωγής (subsumption relations) που υπάρχουν μεταξύ των ερωτήσεων διατηρώντας κατάλληλα στατιστικά χρήσης των αποθηκευμένων αποτελεσμάτων. Επιπλέων, προτείνουμε μια τεχνική εκ των προτέρων αντικατάστασης αποτελεσμάτων η οποία βοηθά σημαντικά στην γρήγορη προσαρμογή του περιεχομένου της λανθάνουσας μνήμης σε αλλαγές στο πεδίο ενδιαφέροντος των κόμβων.

Chapter 1

Introduction

1.1 Scope of Thesis

XML data 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 [9, 14, 32]: 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 query or equivalently whether it is subsumed by any of the materialized views.

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 shared among these peers.

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 their content among all peers. 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 a query are not stored at the peer that posed the query but instead at the peer that is responsible for that part of the data space. In this case, each query may result in triggering a re-organization of the physical storage of data. This involves overheads of moving data around but reduces the degree of replication among caches. Furthermore, 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 infrequent are not indexed and thus must be searched along all potentially remote data sources.

An important aspect is how to actually 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 have been shown to provide efficient lookup in highly dynamic settings. In particular, DHTs are self-organizing, requiring no centralized authority or manual configuration. Most importantly, they are scalable in the sense that both latency (in terms of the number of hops per lookup) and the local state required typically grow logarithmically in the number of peers. DTHs support (either directly or indirectly) a hash-table interface of *put*(key, value) and *get*(key) [44].

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 changes.

We have fully implemented both schemes on top of a DHT, namely CHORD [44]. We present experimental results that (a) quantify the ratio of local versus global communication cost, for which building a cooperative cache is beneficial, (b) clearly identify the conditions under which one approach outperforms the other, (c) show that our replacement decisions that are based on local access statistics produce comparable results with those based on global access statistics, and (d) evaluate design alternatives that concern cache operations.

In a nutshell, the thesis makes the following contributions:

- Proposes and evaluates two ways of building a self-organized distributed semantic caching scheme for XML,
- Exploits a prefix-based approach to indexing and storing path queries that uses a DHT interface for distributing the nodes of the index among the participating peers,
- Introduces appropriate replacement strategies,
- Presents a thorough experimental evaluation that clarifies the issues that actually affect cache performance.

1.2 Thesis Outine

In chapter 2, we introduce the problem of building a cooperative XML cache and the two approaches for handling it. In chapter 3, we present our basic data structure for indexing and storing cache results along with the related operations. Furthermore, we discuss the impact of peer dynamicity on the cache operations. In chapter 4, we describe our replacement strategies, while in chapter 5, we report our experimental results. chapter 6 includes a comparison with related work. chapter 7 concludes this thesis with a sumary of our contributions and directions for future work.

Chapter 2

Cooperative Cache Sharing

In this chapter, we introduce the problem of sharing cache content among a number of peers and the two basic approaches for supporting sharing. To provide a better understanding of the issues handled in this chapter and in the rest of the thesis, we initially present some background on XML and P2P systems.

2.1 Background

2.1.1 XML and XPath Queries

The Extensible Markup Language (XML) [46] has evolved as the new standard for data representation and exchange in the internet. XML is a self-descriptive language that was designed so as to focus on the description and the structure of the data. Its flexibility in describing heterogeneous data makes it very popular for distributed applications and systems where the data are either native XML documents or descriptions of data or services that are represented in different formats in the underlying sources. Furthermore, XML supports more expressive query languages that address both the structure and the content of the data and therefore, they convey more semantic queries.

XML employs a tree-structured model for representing data. The main building blocks of an XML document are the elements that may contain other elements, attributes and content. Elements are hierarchically structured and provide information about the data they describe. Attributes are used to provide additional information about the elements. Typically, an XML document is modeled as a node-labeled tree $T(V, E)$ (Figure 2.1). Each node $e \in V$ corresponds to an XML element with a label assigned from some string literals alphabet that captures the element's semantics. Edges $(e_i, e_j) \in E$ are used to capture the containment of element e_j under e_i and leaf elements in T typically contain values.

XML documents can be considered as consisting of a set of fragments. XML fragments are subtrees of XML documents. A fragment is defined by the path from the root of the XML document to the node of the tree where the fragment is rooted. For example, in Figure 2.1, the fragment defined by *catalog/book/author* is the subtree rooted at the node *author*, and contains the *first name* and the *last name* of all authors of all books in the catalog.

XPath [4] is a query language that offers an expressive way to specify and select parts (subtrees) of an XML tree, i.e. XML fragments. It uses path expressions which are structural patterns that can be matched to nodes in the XML data tree. A path pattern is a sequence of one or more location steps. In its basic form, a location step specifies a node name (i.e., an element name), and the hierarchical relationships between the nodes are specified using child ("/") axis (i.e., at adjacent levels) and descendant ("//") axis (i.e., separated by any number of levels). For example (Figure 2.1), the query $/catalog/book//last\ name$ selects the last names of all descendants of element *book* which itself is a direct child of the root element *catalog* in the document(i.e. the names of all authors and editors of all books in the catalogue). XPath also allows the use of a wildcard operator ("*") to match any element name at a location step. For example (Figure 2.1), the query $catalog/*/title$ selects the titles of all children of element *catalog* (i.e. the titles of all books and journals in the catalog). Each location step can also include one or more predicates to further refine the selected set of nodes. However, in this thesis we do not consider such predicates, thus we will not deepen in how they are described and handled in the context of XPath.

```
<xml>
  <catalog>
    <book>
      <title>...</title>
      <author>
        <first name>...</first  name>
        <last name>...</last name>
      </author>
    </book>
    <book>
      <title>...</title>
      <editor>
        <first name>...</first name>
        <last name>...</last name>
      </editor>
    </book>
    <journal>
      <title>...</title>
    </jounal>
  </catalog>
```

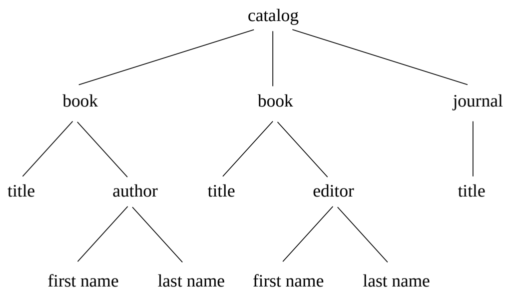

Figure 2.1: Example of an XML document (left) and its corresponding XML tree (right).

2.1.2 P2P Systems

Peer-to-Peer (P2P) systems are massively distributed, decentralized and highly volatile systems for sharing large amounts of resources. The component nodes of such systems participate in similar roles, and are therefore peers to each other. Peers communicate through a self-organizing and fault tolerant network topology which runs as an overlay on top of the physical network. The main purpose of the overlay network is the routing of queries for discovering peers in the P2P system that hold the resources requested. Such routing must be achieved by contacting as "small" a number of peers as possible and by maintaining as "little" state information at each peer as possible.

There are two basic types of P2P overlays: structured and unstructured. Structured overlays ([1, 39, 40, 44]) map data items to peers in the overlay. In unstructured overlays ([17, 25, 35]) there is no correlation between peers and data items. In structured overlays the mapping is usually done through hashing the key space of the data items to the id space of the peers, therefore structured overlays are commonly called Distributed Hash Tables (DHTs). Thus, each peer in the overlay maintains a partition of the data space. In structured overlays, lookup reduces to locating the peer in the overlay that is responsible for the corresponding data partition. Most such overlays guarantee lookup operations that are logarithmic in the number of peers. Generally, DHTs support a hash table interface of *put*(key, value) and *get*(key). Unstructured overlays usually provide no guarantees for the performance of lookup. In the lack of any additional information, search in unstructured overlays is done through some form of flooding, where each peer forwards the lookup to its neighbors until the requested resource is located. To assist in selecting an appropriate neighbor for forwarding the query, often some additional information is kept at each peer in the form of routing indexes.

2.2 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 distributed among a large number of widely distributed data sources 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 data sources. 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 peers holding the matching documents. Then, by moving documents closer to their requestors, 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 a database at the application tier. Our focus on this thesis 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 cache of query results. The cached queries are basically materialized views. At any moment, the cache contains a set of queries, $S = \{Q_1, Q_2, ..., Q_m\}$. When a new query Q is posed at any peer, the cache is checked to see whether Q can be answered from the results of the cached queries. If this is the case, we have a *cache hit*. Otherwise, we have 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 fully benefit from caching, query *subsumption* must be exploited. We say that Q_1 *subsumes* Q_2 if the result of Q_1 fully contains the result of Q_2 and thus they can be used to answer it. A lot of work has been done on the topic of determining a subsumption relation between queries on XML documents [3, 32, 33]. To determine the subsumption relation between queries, we shall use the following Property:

Property 2.1. *Query Q_1 subsumes query Q_2, if Q_1 is a prefix of Q_2.*

Proof. From [32], two conditions are defined for determining when Q_1 subsumes query Q_2. The first condition defines the relation that should exist between the path expressions of Q_1 and Q_2. In particular, it states that Q_1 answers Q_2 if Q_1 is a prefix of Q_2. The second condition defines the relation that should exist between the predicates of Q_1 and Q_2. Since in this work only XPath queries that contain no predicates will be considered, the second condition holds automatically. ∎

In this thesis, we assume that Q is an XPath query. The results of all queries are fragments of the original documents. Initially, we assume that we cache queries that are linear path expressions (containing only the child axis) each starting from the root node of the document. Next, we will relax this assumption to include queries containing the descendant axis, wildcards or non root-to-leaf queries.

2.3 Index Versus Data Sharing

Traditionally, caches are maintained locally at each peer. Each peer caches results of its local queries for future use and has access only to its own local cache. Other peers posing the same or similar queries will have to go through an expensive lookup procedure to obtain the required answers. Thus, when there is overlap between the peers' query workloads it seems benefitial for peers to share the contents of their caches. To achieve efficient sharing of cached results among peers, the cache content of each peer should be indexed so that peers looking for a cached result can locate it efficiently. Then, each peer can look up in cache for results of previous queries posed by any peer in the system, and use these results for answering new queries. The cache index is built on the set S of the cached queries.

A central issue is the organization of the cache content. 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 of the system. Each peer handles exclusively the storage space and the content of its local cache. With the *DataCache* approach, the peers' caches are handled globally by the system. Specifically, 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. 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 and transfer the results. 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. As a consequence, the effectiveness of caching may be significantly diminished. 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 a better load distribution. Third, with IndexCache the lack of control over other peers' caches might cause more than one peer to store the same fragments answering queries in a subsuming relation, increasing redundancy and hence reducing cache space availability. Fragments answering queries in a subsumption relation have a containment relation themselves: one fragment is a subtree of the other, hence the path expressions defining them are in a subsumption relation. In DataCache, the index has total control on data placement, thus the subsumption property between all cached fragments can easily be checked and redundancy avoided. Therefore, the following property holds by construction for DataCache:

Property 2.2. $\forall$ *cached fragment* F_i, F_j, $i \neq j$, *there does not exist a subsuming relation between their corresponding path expressions* P_i *and* P_j.

The above comparison of the advantages and drawbacks of the two approaches is confirmed by the results of the experiments we performed (Section 5.2).

Chapter 3

Prefix-Based XML Caching

To allow sharing of cached results among peers, efficient indexing of the cached queries is needed. We use a distributed prefix trie to index these queries and a DHT to distribute the index. Using a prefix-based structure supports efficient prefix matching useful for testing subsumptions among cached queries.

3.1 Prefix Trie

For indexing the contents of the peers' caches we use a prefix trie. A trie [18] is a tree for indexing and storing strings. The trie nodes are labeled with prefixes of the indexed strings. The actual strings are stored only in leaf nodes with whose label they share a common prefix. XPath queries can be represented as strings, by considering each element of the query as a character.

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 $l_i \neq l_j \forall i \neq j$, $i, j = 1, \ldots, n$, and $\exists Q_i, Q_j \in S$ s.t. $/l/l_i, /l/l_j$ are prefixes of Q_i, Q_j respectively. We introduce a special labeling case involves trie leaves only: 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 $\exists\ Q_i$ s.t. $/l = Q_i$. We will call such leaves "$NULL''$ leaves and denote their label by $Q_i\perp$. "$NULL''$ leaves are introduced to handle subsumption among queries. Their role will be made clear in Section 3.3. For our prefix trie the following property is valid:

Property 3.1. *Given the above labeling scheme, for each query $Q \in S$, there exists either exactly one trie 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 a prefix of Q. Then there exist two cases: (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).

If (i) is the case then 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 $lb_1 \neq lb_2$. If (ii) is the case then lf_2 is a descendant of lf_1. This is impossible 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 impossible since trie leaves have no children. ∎

Property 3.2. *(Trie Properties). Our prefix trie has the following additional 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 3.2.1 states how queries are mapped to trie nodes. From this property, it follows that the label of each leaf is a common prefix for all the queries indexed/stored in it. In the case of the DataCache, a query Q and the XML fragment answering it are both stored at the corresponding leaf. In the case of an IndexCache, a query Q and a pointer to the peer caching its results are stored at the corresponding leaf. Properties 3.2.2 and 3.2.3 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 3.2.2, the node splits in a number of child nodes and its queries are redistributed among these nodes according to Property 3.2.1. When deleting a query, the total size of the queries contained in a subtrie may become smaller than $C - k$. To enforce Property 3.2.3, the whole subtrie is merged into a single leaf containing all queries in the subtrie. Thus, the shape of the trie depends on the distribution of the cached queries; it is deep in regions where a larger number of queries are indexed/stored, and conversely, "shallow" in regions where fewer queries are indexed/stored. Property 3.2.4 facilitates the lookup procedure.

3.2 Distributed Prefix Trie

The prefix trie is distributed among the peers participating in the cooperative cache through a DHT. Thus, all peers that cooperate in sharing their caches join the DHT and participate in the distribution. The distributed index inherits the DHT's nice load-balancing and fault-tolerance properties, and the efficient lookup it offers. Additionally, since DHTs partition the data space among the participating peers, they can easily achieve the assignment of a part of the query space to each peer in the case of DataCache. 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.

The distributed prefix trie is built gradually as queries are inserted in cache. Initially, it consists of only the root node and, as new queries are indexed/stored, leaves are created and split according to the query distribution. The nodes of the trie are distributed among the peers of the system using their label as the DHT key. This way, 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 of the system as queries are deleted/inserted in the cache. Thus, an even distribution of queries among leaves results in an even distribution of queries among peers, balancing their load in answering lookup requests (IndexCache) and storing XML fragments (DataCache). The distribution of trie nodes through the DHT corresponds to distributing the index among the peers. Additionally, in the case of DataCache, it corresponds to assigning each peer a part of the query space consisting of the queries that are or will be stored at the leaves asigned to it.

As an example of how the trie is distributed among peers in the DHT, consider Figure 3.1. The leaves and the internal node of the trie are assigned to different peers participating in the DHT through its put interface, e.g. $put(/A/C)$ assigns leaf $/A/C$ to peer $P14$ and so on. In DataCache (Figure 3.1(right)) the leaves store the fragments answering the queries stored in the leaves. In IndexCache (Figure 3.1(middle)) the leaves maintain pointers to the peer storing the fragments in their local caches. Specifically, the fragment answering query $/B/F/H$ is cached in peer's $P14$ cache, the one answering query $/A/D/F$ in peer's $P8$ cache and fragment answering $/A/C/L$ is stored in peer's $P56$ local cache.

Our trie resembles the Prefix Hash Tree (PHT) introduced in [8, 38]. The PHT is a binary trie built over a data set consisting of binary strings. There exist a number of 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.

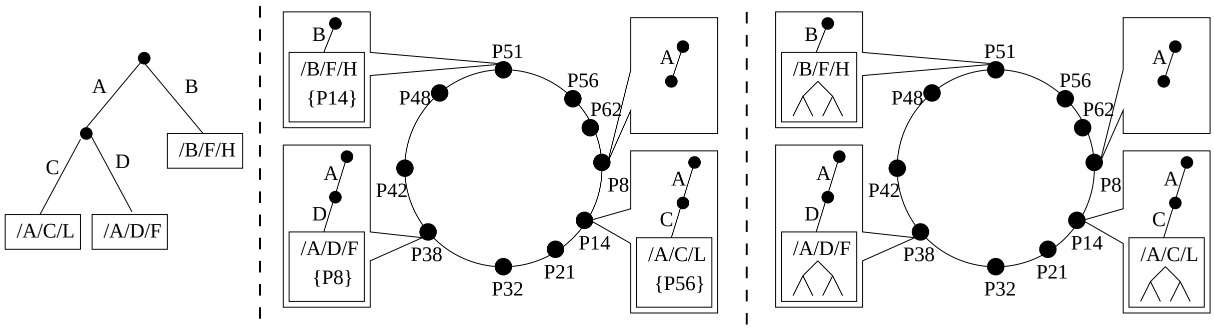

Figure 3.1: A prefix trie (left) and its distribution over the DHT in IndexCache (middle) and DataCache (right).

3.3 Distributed Cache Operations

For answering a query, each peer initially looks for answers in the cooperative cache. If there is a cache miss, i.e. the needed answer is not cached; the peer searches for results in remote data sources and caches/indexes these results. Next, we describe the basic cache operations for IndexCache and DataCache.

3.3.1 IndexCache Operations

In IndexCache, each peer caches locally the results of its queries. In addition, it publishes the corresponding query in the DHT index so that any subsequent query by any peer can use these results. When a query is posed, each peer first checks its cache. If the results of the query can be found locally, then there is a local hit. Else, the peer searches the DHT for relevant results cached at other peers. If such results are found, there is a global hit, and the results are transmitted from the caches of peers storing them. Note, that in the case of an IndexCache, local hits are much more efficient than global ones, since they involve only local processing.

An important issue is handling subsumption. Assume that peer p_1 has cached the results of a query Q_1, and that peer p caches a query Q that subsumes Q_1. IndexCache offers a loosely-coupled form of cache cooperation, in which peers just share the content of their caches but maintain control over the content of their caches. Consequently, peer p_1 is not forced to delete from its cache the results of Q_1. This adds some redundancy, however, p_1 can still benefit from local hits. 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 3.2.1, 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 Property 2.1 and Property 3.2.1, any query indexed in the "NULL" leaf subsumes Q, thus it is located by a DHT lookup and the pointer associated to 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 3.1. *The lookup operation for query Q will always locate an answer for Q if such an answer exists in cache.*

Proof. There exist two cases when looking up Q: (i) Q is cached, or (ii) a query $Q\prime$ which subsumes Q is cached. We will consider each of these cases separately. In case (i), according to Property 3.1 and Property 3.2.1, there exists a unique leaf lf where Q is indexed and lf's label is either a prefix of Q either $Q\perp$. 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 Property 3.1 and Property 3.2.1, "$NULL''$ 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 the 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 Property 3.1 and Property 3.2.1, 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. ∎

Since all the prefixes of the query are looked up and considering the cost for a DHT lookup to be $log(N)$ hops (where N is the number of peers in the system), the cost of the cache lookup operation for a query with L elements is $L \cdot log(N)$. The additional cost for locating "$NULL''$ leaves is also $L \cdot log(N)$ hops, since for each prefix of a query an internal node with a "$NULL''$ leaf may be located, requiring a DHT lookup. Thus, the total cost of a cache lookup is $L \cdot log(N)$. The lookups for each prefix are independent, i.e. a lookup for a prefix does not depend on the outcome of the lookup for any other. Consequently, the DHT lookups for each prefix of a query can be executed in parallel (i.e. can be issued at the same time) thus decreasing the cost of the lookup for a query in terms of latency.

Caching a New Query In IndexCache, a peer p caching a query Q, 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. Thus, before inserting Q in the prefix trie, the unique leaf lf must be located through the lookup operation for Q. This lookup may have three outcomes: locate a leaf, locate an internal node having Q as a label, and none of the above. If a leaf is located, then Q is inserted there. If an internal node having Q as a label is located then a "$NULL''$ leaf is created and Q is inserted in it. If none of the above two cases holds, then a new leaf is created. The label of the new leaf is obtained as follows: When the prefixes of Q are being looked up, zero or more internal nodes having these prefixes as labels may be located. If no such internal node is located, then the first element of Q becomes the label of the new leaf, and the root node becomes the leaf's parent. If some internal nodes are located, then let nd_{max} be the node with the longest label between them corresponding to a prefix of Q 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 the node nd_{max}. During the creation of the new leaves, the label of their parent is known, thus it is recorded by them.

When inserting a new query in a leaf, the total size of queries indexed at it may exceed leaf capacity 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 created with 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 3.2.1. Every time a new internal node or leaf is created it records the label of its parent and sends its own label to its parent.

Algorithm 1 Cache index serial lookup

```
node := DHTlookup(Q/NULL)
if node is a "NULL" leaf then
  if IndexCache then
    SearchInLeaf(node)
  else if DataCache then
    nd := DHTlookup(leaf's parent label)
    SearchSubtrie(nd)
  end if
end if
for i := 1 to L do
  node := DHTlookup(P_i(Q))
  if node is a leaf then
    result := SearchInLeaf(node)
    return result
  else if node is internal then
    if exists leaf child labeled Q/NULL then
      lf := DHTlookup(Q/NULL)
      result := SearchInLeaf(node)
      if result ≠ NULL then
        return result
      end if
    end if
  end if
end for
return NULL
// L is the number of elements in Q
// P_i(Q) is the prefix of length i of Q
```

Claim 3.2. *The operation for caching a new query is consistent to the trie labeling definition, and preserves Properties 3.1 and 3.2.*

Proof. Initially we will prove that the operation is consistent to the trie labeling. For "$NULL$" leaves it means that when creating such a leaf during the insertion of Q, it is the only such leaf for the internal node nd. Suppose that nd already has a "$NULL$" leaf. Then, by Property 3.2.1 query Q is already indexed in this leaf. This means that when looking up Q, there will be a cache hit, hence query Q

can not be cached again. During leaf splitting a number of internal nodes and leaves are created. All the internal nodes between the parent of the split leaf to the parent of the new leaves are labeled with increasing lengths as demanded. Additionally, all the new leaves are labeled by appending to the label of their parent node, the next common element in the queries they contain. Thus, even the labeling of the new leaves is consistent to the definition.

After the creation of new leaves ("$NULL''$ or otherwise) the existing and the new query are indexed in the leaf having their prefix as a label, thus preserving Property 3.2.1. The purpose of the leaf splitting operation is to preserve Property 3.2.2, while at the same time it preserves Property 3.2.4 since during splitting all new internal nodes and leaves record the labels of their parents and children. The operation of caching a new query only increases the subtrie sizes, thus obviously it preserves Property 3.2.3.

Property 3.1 would be violated if a new leaf is created for query Q despite the existence of a leaf lf whose label is a prefix of Q. This can not be the case since, as already proved; if lf existed then the lookup operation would have located it. Suppose a leaf lf_1 is created during leaf splitting and there already exists a leaf lf_2 having the same label as lf_1. The label of lf_1 is longer than label of the old leaf lf_{old} and has the latter as a prefix, thus the label of lf_{old} is also a prefix of lf_2. Since the operation of caching a new query is consistent to the trie labeling definition, lf_2 is a child of lf_{old} which is impossible. ∎

Query Deletion When the results of a query are removed from a cache, the query must also be deleted from the index. Deleting a query may trigger merge operations; if its deletion makes the total size of the leaves rooted at some internal node smaller than $C - k$, then a merge operation must be performed to preserve Property 3.2.3. For this purpose, each internal node nd keeps track of the total size of the leaves in its subtrie and is responsible for the merging procedure: the whole subtrie is merged into a single leaf with nd's label and containing all the queries in the subtrie. The choice of merging parameter k determines how aggressive the merging is. A small k causes merging to be triggered more often than a larger k. We have experimentally seen that a suitable value for k is $C/2$, which reduces the number of merges and consequently the overhead induced.

Claim 3.3. *The delete operation is consistent to the trie labeling definition, and preserves Properties 3.1 and 3.2.*

Proof. The delete operation may violate the labeling and properties of the trie only in the case of a merge. When a merge occurs a subtrie is deleted and the node where it was rooted is replaced by a leaf having the same label. Thus, if the trie was consistent to the trie labeling definition before the merge it will remain so after the merge.

The label of the leaf resulting from the merge is a prefix of all the leaves in the merged subtrie. Thus, all the queries stored/indexed in the new leaf have its label as a prefix preserving this way Property 3.2.1. Clearly, since the size of the subtrie is smaller than $C - k$, the size of the new leaf is smaller than C, thus preserving Property 3.2.2. Properties 3.2.3 and 3.2.4 are clearly preserved.

For the Property 3.1 to be violated after the merge, (i) the label of the new leaf must be a prefix of an existing leaf or (ii) an existing leaf must be a prefix of the new leaf. The leaves in the merged subtrie are the only ones having the label of the new leaf as a prefix. These leaves are deleted during the merge thus (i) can not be the case. If an existing leaf is a prefix of the new one then it also was a prefix of the internal node where the subtrie was rooted before the merge. Thus, from the labeling definition, the internal node is a descendant of the leaf which is impossible. As a consequence, case (ii) does not hold and hence, the deletion operation preserves Property 3.1. ∎

3.3.2 DataCache Operations

In DataCache, along with the indexed query, each leaf of the trie also stores the fragment answering it. Each peer is assigned by the DHT and stores locally a number of the trie leaves and internal nodes, which corresponds to assigning it a part of the query space. There is no difference 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 the 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. As in IndexCache, the position in the trie where the new query Q will be inserted is looked up. If the outcome of the lookup is a leaf lf, and Q subsumes some query $Q\prime$ stored in lf, then $Q\prime$ and the fragment $F\prime$ answering it are deleted from the leaf and replaced by Q and the associated fragment F. This does not involve any restructuring of the trie.

Algorithm 2 Cache index binary lookup

```
node := DHTlookup(Q/NULL)
if node is a "NULL" leaf then
  if IndexCache then
    SearchInLeaf(node)
  else if DataCache then
    nd := DHTlookup(leaf's parent label)
    SearchSubtrie(nd)
  end if
end if
lo = 0, hi = L
while lo ≤ hi do
  mid = (lo + hi)/2
  node = DHTlookup(P_mid(Q))
  if node is a leaf then
    result = SearchInLeaf(node)
    if result = NULL then
      parent := DHTlookup(node's parent)
      result = SearchForNull(parent)
      return result
    end if
  else
    if node is internal then
      lo = mid + 1
    else
      hi = mid − 1
    end if
  end if
end while
return NULL
```

Consider however, the case in which the outcome of the lookup is an internal node nd. According to the trie labeling definition and Property 3.2.1, 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 2.2 (i.e. avoid redundancy): (a) merge the whole subtrie rooted at nd into a single 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 there the sub-fragment $F - \hat{F}$, where the set $\hat{F}$ is obtained by including all the paths in F that have any of the labels of nd's children as prefixes. The leaves in nd's subtrie record the fact that parts of F are stored in the subtrie rooted at nd.

Both options have some advantages. In option (b), there is a cost for notifying the leaves in the

Algorithm 3 Functions used in Algorithms 1 and 2

```
function SearchInLeaf(lf)
if Q is stored in lf then
    if IndexCache then
        return ptr_Q
    else if DataCache then
        return F_Q
    end if
else if Q' is stored in lf and Q' subsumes Q then
    if IndexCache then
        return ptr_Q
    else if DataCache then
        return F_Q
    end if
else
    return NULL
end if
// ptr_Q is the pointer to the peer caching the fragment answering Q
// F_Q is the fragment answering Q

function SearchSubtrie(node)
for all leaf child of node labeled lb do
    lf := DHTlookup(lb)
    F = the set of fragments stored in lf
    result ∪= F
end for
for all node child of node labaled lb do
    nd := DHTlookup(lb)
    result ∪= SearchSubtrie(nd)
end for
return result

function SearchForNull(parent)
if exists leaf child lf of parent labeled Q/NULL then
    lf := DHTlookup(Q/NULL)
    result = SearchInLeaf(lf)
else if parent ≠ root then
    pr := DHTlookup(parent's parent)
    result = SearchForNull(pr)
end if
return result
```

subtrie, but this cost is equivalent to the cost of merging the subtrie in option (a). However, in option (a) consequent queries for Q are found in the leaf storing F, whereas in option (b) since each of the leaves in the subtrie rooted at nd stores a part of the answer, they must all be accessed to produce answers, increasing the lookup cost. On the other hand, in option (a), all consequent queries for fragments in the deleted subtrie will be served by the peer storing the new leaf, possibly overloading it. Additionally to better query load balancing, in option (b) fragment sizes are kept small and the number of leaves bigger, helping their better distribution among the peers in the DHT. Better fragment distribution increases cache performance, as our experimental results show. The above observations, which are confirmed by the results of the experiments we conducted (Section 5.4.1), lead us to the adoption of option (b).

Claim 3.4. *The operation of caching a new query Q in DataCache retains Property 2.2.*

Proof. For Property 2.2 to be violated, there must exist at least two cached fragments whose path expressions are in a subsuming relation. Here we are dealing with simple queries containing only the descendant axis, thus the path expressions of the fragments are exactly the queries which the fragments answer. Hence, the property would be violated if the results of two queries in a subsuming relation are cached. If the operation of caching a new query Q detects a subsumption relation between Q and any cached queries, it stores the common fragments only once. Thus, to prove Claim 3.4 it is sufficient to prove that the caching operation locates all the queries subsumed by Q. All queries subsumed by Q have Q as a prefix (Property 2.1).

First, we consider the case when Q is inserted in an existing leaf lf. Since the label of a leaf is a common prefix for all the queries it contains, all queries having a common prefix with Q are stored in lf. Consequently all queries subsumed by Q are stored in lf.

When a "$NULL''$ leaf is created, all leaves in nd's subtrie have the label of nd as a prefix. Additionally, according to Property 3.2.1, all queries stored in the subtrie have the labels of their leaves as a prefix. Since nd and Q have the same label, it follows that all the queries in the subtrie have Q as a prefix, hence are subsumed by it. If a new leaf is created, it means that there is no leaf having a prefix of Q as a label, thus in this case Q subsumes no cached queries. ∎

The query lookup is similar to that in the case of IndexCache, with the exception of the case 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. This is implemented by recursively visiting the children of internal nodes in the subtrie as shown in Algorithm 1. The additional lookup cost in this case can increase up to $n \cdot log(N)$, where n is the number of leaves and internal nodes in the trie. Query deletion is identical with that in the case of IndexCache.

3.3.3 Binary Lookup

As already discussed, the lookup algorithm presented executes parallel DHT lookups for all the prefixes of a query to achieve low latency in query execution. Thus, all the trie nodes in the path from the root of the trie to a leaf or internal node are visited in a lookup. Consequently, this may lead to a situation where peers responsible for the upper level trie nodes will be overloaded despite the load balancing mechanisms of the underlying DHT, since they are visited in (almost) every lookup. This can be significantly improved by using a binary search on prefix lengths of query Q. Specifically, if the currently located prefix is an internal node, the lookup proceeds with longer prefixes. Alternatively, if the current prefix is not an internal node or leaf, the lookup proceeds with shorter prefixes. If a leaf node containing the answer is located the lookup terminates. Otherwise, as in linear, lookup "$NULL''$ leaves must be checked, for queries subsuming Q. In this case, since the binary lookup does not visit all internal nodes from the root to the leaf, all the internal nodes in the path from the leaf to the trie root are checked until a result is found or the root is reached. The algorithm for binary lookup is given in Algorithm 2.

Binary lookup does not always visit the trie nodes corresponding to each prefix of a query, avoiding the overloading of the upper level nodes, consequently of the corresponding peers (as confirmed by our experimental results of Section 5.4.2). Additionally, binary lookup reduces the index lookup cost since it visits only $O(logL)$ trie nodes while looking up a query with L elements. Hence the cost of the binary lookup operation is $O(logL \cdot logN)$ hops which is lower compared to the linear lookup. The additional cost for visiting the internal nodes from the leaf to the root is $L \cdot log(N)$ hops. This additional cost seems to make the performance (cost and load) of binary lookup similar to that of linear lookup. This is not the case for a number of reasons. First, when caching is employed the expected cache hit ratio is expected to be more than 50% for caching to be beneficial. Thus less than half of the lookups may travel the root to leaf path. Second, “$NULL''$ leaves are generally expected to be “spread” on the path and not concentrated near the root. Third, the cost is decreased for DataCache since the internal nodes in the path from the leaf to the trie root are checked only if the leaf located by the lookup, and not containing the queries result, has a record for the creation of a “$NULL''$ leaf.

In binary lookup the operation proceeds sequentially, i.e. each step depends on the outcome of the previous. Specifically, each DHT lookup has to "wait" the outcome of the previous to determine the direction it will proceed. Therefore, the tradeoff for a lower lookup cost and better query load distribution is the increase in query execution latency due to the sequential lookups.

3.4 Other Queries

The discussion so far involved simple root-to-leaf XPath queries containing only the child axis ("/"). Queries containing the descendant axis ("//"), wildcards ("*") or non root-to-leaf queries can easily be supported by our cooperative cache, keeping the cache operations very similar to those described above. In this section we will discuss how this is achieved for the cooperative cache.

3.4.1 Queries Containing the Descendant Axis

We will examine how the two basic cache operations, i.e. cache lookup and caching a new query, are performed to cope with caching of queries containing the descendant axis.

Caching a New Query We consider general queries in which the descendant axis may appear a number of times. Formally, we consider queries of the form $Q = P//R\{//T\}$, where P and T are queries containing only the child axis or empty strings, R is any query (containing child and/or descendant axis, or R is the empty string), and the part contained between the "{}" is optional. As in the case of queries containing the child axis, after a failed cache lookup for Q, the query is looked up at some remote data sources. The result of Q consists of a set of m of fragments described by the path expressions $P/F_i\{/T\}$, $i = 1,..,m$, where F_i are expressions containing only the child axis. We call the expressions $P/F_i\{/T\}$ component queries of Q. To cache Q (for both DataCache and IndexCache), first, we perform a lookup for P. The outcome of the lookup, node n_q, is either a leaf or an internal node of the trie (note than if P is the empty string the lookup will locate the root node of the trie which has the empty string as a label). All the component queries $P/F_i\{/T\}$ are cached as queries containing only the child axis. If n_q is a leaf, these queries are cached in n_q since P is a prefix of each $P/F_i\{/T\}$. Whereas, if n_q is an internal node, these queries are cached at the subtree rooted at n_q since P is a prefix of each $P/F_i\{/T\}$. In any case, 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 internal node that is created by the split having P as a label records the fact that Q is cached in the subtrie.

Claim 3.5. *The operation for caching a new query that contains the descendant axis is consistent to the trie labeling definition, and preserves Properties 2 , 3 and 4.*

Proof. The caching of a query containing the descendant axis involves only caching a number of component queries which contain only the child axis, and keeping a small amount of information in leaves and internal nodes. We have already proved this claim for caching new queries containing only the child axis. Additionally, keeping the information in the trie leaves and internal nodes is equivalent to keeping the additional information of caching a query that subsumes queries in a subtrie for DataCache. Thus, the claim holds. ∎

The cost for caching a query containing the descendant axis is $Nr \cdot C_Q$, where Nr is the number of the components of the query being cached and C_Q is the cost of caching a query containing only the child axis.

Cache Lookup For answering a query Q containing the descendant axis we can only use cached results of Q or another cached query Q_S which contains the descendant axis and subsumes Q.

For looking up a query $Q = /P//R\{//T\}$, initially a lookup for P is performed (as in Algorithms 1 and 2). If the outcome of the lookup, i.e. nd, is a leaf then the leaf is checked for a record of caching Q or a query Q_S subsuming it. The subsumption check is performed locally at the leaf by checking all the records that the leaf holds about queries it caches. If nd is an internal node then a check is performed if it has a record of Q or a query subsuming it being cached in the subtrie. If so, a broadcast is performed as in the case of looking up a query stored in a "$NULL$" leaf in DataCache. For binary lookup, if nd does not have a record for Q or a query subsuming it, the path from nd's parent (if nd is a leaf) or from nd (if nd is an internal node) to the root is traversed until such a record (i.e. 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 transversal of the path is not required. For performing the subsumption check on queries containing the descendant axis a number of different techniques may be used ([3, 32, 33]). We are not concerned with which technique to use. The only property of the subsuming relation that is substantial to the lookup operation is:

Property 3.3. *If query $Q_1 = P_1//R_1\{//T_1\}$ subsumes query $Q_2 = P_2//R_2\{//T_2\}$, then P_1 is a prefix of P_2 [32].*

Claim 3.6. *The lookup operation for query Q that contains a descendant axis will always locate an answer for Q if such an answer exists in cache.*

Proof. There exist two cases when looking up $Q = P//R\{//T\}$: (i) the components of Q or $Q_S = /P_S//R_S\{//T_S\}$ subsuming Q are stored in the same leaf or (ii) the components are stored in different leaves. In case (i), the lookup for P will locate the unique leaf lf having a prefix of P as a label. If Q is cached, then, from Properties 3.1 and 3.2.1, its components will be stored in lf since they have P as a prefix. If Q_S is cached then the lookup for P will locate the leaf lf_S where the components of Q_S are stored since, from Property 3.3, P_S is a prefix of P and, from Properties 3.1 and 3.2.1, lf_S is the unique leaf having a prefix of P as a label. In case (ii), if Q is cached then its components are stored in the subtrie rooted at the internal node having P as a label. The lookup for P will locate this node and successfully complete the lookup. If Q_S is cached, again its components are stored in the subtrie rooted at the internal node having P_S as a label. The lookup for P will locate either a leaf or an internal node nd_1 which does not have a record for Q. Then, the path from the leaf's parent, nd_2, or from nd_1 to the root will be traversed. All the ancestors of nd_1 or nd_2 have prefixes of P as labels and, from Property 3.3, P_S is a prefix of P. Thus, the node recording information about Q_S will be located and the broadcast performed. ∎

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. Thus, in the worst case, the cost can increase up to $n \cdot log(N)$ when P is the empty string, n being the number of leaves and internal nodes in the trie.

3.4.2 Queries Containing Wildcards

We consider queries of the form $Q = P/*/R\{/*/T\}$, where P and T are queries containing only the child axis or empty strings and R a query containing wildcards and the child axis, or the empty string. Similar to the case of descendant axis, the result of Q is a set of m of fragments defined by the expressions $P/F_i\{/T\}$, $i = 1, .., m$, where F_i are expressions containing only the child axis. Thus, the operations described for queries containing the descendant axis can be also applied for queries containing wildcards. As for queries containing the descendant axis, the only property of the subsumption relation between queries containing wildcards used during the cache operations is:

Property 3.4. *If query $Q_1 = P/*/R\{/*/T\}$ subsumes query $Q_2 = P_2/*/R_2\{/*/T_2\}$, then P_1 is a prefix of P_2 [32].*

3.4.3 Non Root-to-Leaf Queries

We consider queries of the form: (i) $Q = //Q_P$, (ii) $Q = Q_P//$ or (iii) $Q = //Q_P//$. Obviously the query type (i) is identical to queries containing the descendant axis when $Q_P = R$, P is the empty string and the //T part is omitted. Similarly, the query type (ii) is identical to queries containing the descendant axis when $Q_P = P//R$ and T is the empty string. Finally, the query type $Q = //Q_P//$ is identical to queries containing the descendant axis when $Q_P = R$, and P and T are the empty strings. Consequently, all three types of queries can be treated as queries containing the descendant axis. Thus, the operations for queries containing the descendant axis are used even for the case of non root-to-leaf queries, without requiring any change.

3.5 Discussion

3.5.1 Peer Dynamicity

So far we have assumed that peers participating in the cooperative cache are stable. In this section we investigate how a dynamic behavior of the peers (peers joining/leaving or failing) would influence the cooperative cache and its operations.

Peers Joining/Leaving the Cooperative Cache

The joining (leaving) of a peer in the cooperative cache means it will acquire (give up) a part of the distributed prefix trie. This is corresponds to the peer joining (leaving) the underlying DHT. DHTs provide mechanisms to efficiently handle peers joining or voluntary leaving the system. When a new peer joins the DHT, these mechanisms ensure that the peer becomes responsible for a part of the data space indexed/stored by the DHT. In the case of the cooperative cache, the data space consists of the distributed prefix trie. Thus, the DHT assigns to the new peer a number of trie nodes (leaves or internal), which are transferred from the peers previously holding them to the new peer. When a peer voluntary leaves the DHT, the DHT distributes its content, i.e. the trie nodes that the peer holds, to the appropriate remaining peers.

From the above discussion it follows that the procedures of peers joining/leaving the cooperative cache do not disrupt its operation. Both procedures only create some additional network traffic for the redistribution of some trie nodes between the peers of the DHT. This traffic is almost insignificant in the case of IndexCache since trie nodes store a small amount of information. However, in the case of DataCache where trie leaves store actual XML fragments, the demand for network bandwidth will be increased.

Peer Failure

When a peer participating in the cooperative cache fails, the part of the distributed trie it maintains and the fragments it caches are lost. What this implies for the cooperative cache depends on which approach is adopted, DataCache or IndexCache.

DataCache When a peer fails in DataCache the internal nodes and the leaves of the trie along with the fragments stored in them are lost. The loss of the trie leaves will cause the following queries for fragments stored in them to fail, i.e. a cache miss. Hence, the query will be answered from remote data sources and its result cached again, restoring the cache state. Consequently, the loss of trie leaves does not significantly affect the cooperative cache, apart from a possible temporary slight decrease in its hit ratio. On the other hand, the loss of an internal node influences the cache operations. Specifically, during cache binary lookup the loss of an internal node may cause the failure of the lookup even if the peer maintaining the leaf that contains the fragment being looked up is alive (see Algorithm 2). This is not the case for linear lookup, since the proper leaf is guaranteed to be located if the peer holding it is alive (see Algorithm 1). However, in both linear and binary lookup the location of NULL leaves will wrongly fail as a consequence of a lost internal node (see Algorithm 1 line 16 and Algorithm 2 line 18). Additionally, the lookup for queries cached in a subtrie rooted at the missing internal node will also fail (see Algorithm 1 and Algorithm 2 lines 1-9). All the above failures will cause the fragments not located by the lookup to be re-cached, creating a redundancy which is not acceptable for DataCache.

IndexCache When a peer fails in IndexCache the internal nodes, the leaves of the trie assigned to it and the content of its local cache are lost. The loss of the internal nodes has the same consequences for the cache operations as in the case of DataCache. The loss of local cache contents has the same effect as the loss of leaves in DataCache, with the difference that now some leaves of the trie contain pointers to failed peers. However, these pointers will be updated after the lost cache content is restored by following queries. The loss of trie leaves will cause the cached fragments indexed by these leaves to be inaccessible by peers other than the ones storing the fragments in their local caches. Thus, a following query (not form the later peers) will be answered from remote data sources and its result cached again. While this restores the cache state, it creates additional redundancy since the fragment contained in the result is cached by the peer posing the query and the peer where the lost leaf pointed to. The redundancy will further reduce cache space availability and consequently reduce cache performance.

As discussed above, in both approaches the loss of cached fragments is not harmful for the operation and performance of the cooperative cache. On the other hand, the loss of internal trie nodes has a significantly negative impact. Additionally, for the case of IndexCache, even the loss of trie leaves deteriorates the cache performance. Thus it is substantial to ensure the availability of the information of internal nodes and IndexCache leaves. This can be achieved through replication of nodes and leaves, taking advantage of the replication mechanisms offered by the underlying DHT. The amount of data to be replicated is not significant since the information maintained by trie nodes and IndexCache leaves is very small. Thus, the replication can be achieved easily without any significant increase in the demand for network bandwidth.

3.5.2 Choice of Indexing Structure

As the indexing structure for the cached queries we used the prefix trie. The trie supports efficient prefix based search which is significant in determining subsumption relations among XPath queries. The determination of such relations is extremely important in maximizing cache utilization and achieving a better cache space utilization by avoiding redundancy of cached results. Tries are often used as indexing structures for XML documents in a centralized setting [13]. Other indexing choices choices involve using trees ([20, 34, 30]) or indexing based on path expressions of different parts of XML documents ([7, 42, 19]). In the case of IndexCache, replacing the prefix trie with any of the aforementioned approaches would affect only the complexity of cache operations. On the other hand, in DataCache, cached queries along with their results are distributed among peers based on the indexing structure, i.e. the distributed prefix trie. Consequently, using another indexing approach may differentiate the distribution and consequently the overall cache performance. However, it is not obvious which approach offers the best distribution. Furthermore, the choice of the best possible indexing structure is not the scope of this thesis.

Chapter 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 Cp 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 space for maintaining the index is small and replacement decisions are needed only when caching the actual result fragments of each query.

In particular in the case of DataCache, a number of 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 a 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 us for 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. The queries chosen for eviction must be the ones that will be less likely used by the actual peer or other peers in the system for answering subsequent queries. The most common techniques for choosing the appropriate queries are LFU and LRU. In this work, LRU will be used and for that purpose every time a cached query is used by a peer for producing answers its utilization value UV is set at the time of access.

At this point the issue of when to update the utilization value of a cached query arises: every time it is used by *any* peer in the system (global hit) or only when it is used by the very peer that cached it (local hit)? The first, non-selfish approach corresponds to a higher level of cooperation among peers where the cache is managed based on the aggregated query workload of the peers. The second, selfish approach corresponds to a lower level of cooperation where each peer manages its local cache based only on its own query workload. DataCache corresponds to a high level of cooperation where the cache content is distributed among the peers and globally handled based on the distributed trie, thus the non-selfish approach for updating the UVs seems the natural choice. On the other hand, in IndexCache a peer handles its cache based exclusively on its query workload, hence the selfish approach seems more appropriate. However, the selfish approach may diminish the benefits of cooperative caching. Specifically, an overactive peer p may continuously replace cached queries which are popular to other peers in the system but not to p, dropping this way the overall cache performance. Thus, to get the most of the cooperative cache the non-selfish approach would be beneficial even in the case of IndexCache. On the other hand, this approach may lead to *mistreatment* problems. Specifically, an increased number of remote hits in a peer's p local cache by peers having high query rates, will cause the corresponding queries to remain longer in p's cache since their UVs are continuously increased. As a consequence, queries that p would want to keep in its cache will be often chosen as the replacement victims, dropping p's cache performance. However, as shown in [29], IndexCache-like cooperative caches that adopt the non-selfish approach are fairly resilient to the onset of mistreatment, as long as all the participating peers are in close network proximity and the network distance between each pair of peers is similar. This is the case for our setting, thus we adopted the non-selfish approach for updating UVs for both DataCache and IndexCache.

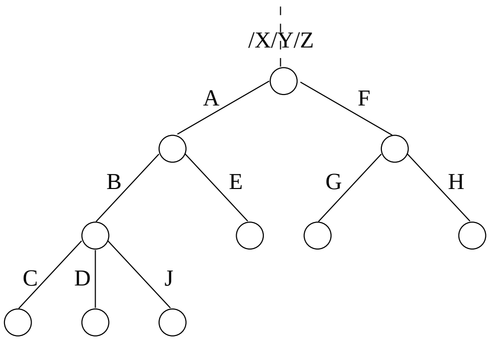

Figure 4.1: Example of an XML fragment answering query $/X/Y/Z$.

Additionally to answering exact match queries, cached fragments can be also used for answering other queries which are subsumed by the cached query. Hence, access statistics for parts of a fragment used for answering subsumed queries should also be maintained. Therefore, when replacement is needed paths of the fragment answering a query will be considered instead of the whole fragment. [9] shows that this partial replacement strategy is beneficial for caching XPath queries. Specifically, separate access statistics are maintained for each root-to-leaf path of a cached fragment used for answering a query. Hence, when such a path is used for answering queries subsumed by the query corresponding to the fragment, its UV is updated. The replacement unit now is a root-to-leaf path of each cached fragment instead of the whole fragment. For example, consider Figure 4.1 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 the fragment corresponding to Q, since Q subsumes both Q_1 and Q_2. In this case, the access statistics updated are those for the subpaths $/X/Y/Z/A/B/C$, $/X/Y/Z/A/B/D$, $/X/Y/Z/A/B/J$ and $/X/Y/Z/F/G$, used in answering Q_1 and Q_2.

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 the leaves stored at p and evict those with the smallest UV values. This policy allows for a better 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

Evicting fragment parts from the cache will affect cache utilization. For example, consider Figure 4.1. Suppose paths $/X/Y/Z/A/B/C$ and $/X/Y/Z/A/B/D$ are replaced. The resulting fragment does not answer query $/X/Y/Z$ anymore, thus the distributed index must be properly updated. A straightforward approach is to delete the index entry for query $/X/Y/Z$ and instead create entries for all the remaining paths in the fragment. However, although leaving the cache operations intact, this approach will result in a substantial increase of the number of indexed queries leading to an increase of the index operational (insert, delete) and maintenance (split, merge) cost. The increase in the split cost is more obvious in IndexCache where the leaf size is determined on the indexing information maintained. More indexed queries entail increase of the indexing information and consequently increase of the number of splits. Thus, a better approach is keeping the index entry for query $/X/Y/Z$ along with some information about the replaced paths. The index entry is deleted only if all the paths of the fragment associated with it are replaced. As a tradeoff for better performance, this approach necessitates some modifications of the distributed cache operations that we describe next.

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. If this is not the case, we have a cache miss; the answer is looked up in remote data sources and cached. Caching results of these queries is different from that described in Section 3.3. For DataCache, when such a result is inserted in a leaf, 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 a cached query 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 preserves Property 2.2. 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 the same as in Section 3.3.

Path replacement causes an additional change in the lookup and insert operations in DataCache, for a query Q stored in a "$NULL$" leaf. When initially caching Q some of the leaves in the subtrie rooted at the "$NULL$" leaf's parent may store partial fragments. Thus, the missing parts of these fragments are restored after contacting the corresponding leafs. Additionally the UVs of all the paths stored in the subtrie are updated. The refreshing of the UVs and the fragment restoring are performed at the same time with the notification of the leaves in the subtrie for the insertion of Q, thus not increasing the query insertion cost. After the insertion of Q in the "$NULL$" leaf, some of the leaves in the subtrie may again replace some paths, making Q impossible to be answered by cache. For subsequent lookups for Q to be able to detect this fact, the leaf performing the replacement notifies the "$NULL$" leaf storing Q. Thus, based on this information, lookups can quickly determine the cache miss without needing to visit the whole subtrie.

Path replacement has the same effect on cache operations for queries containing the descendant axis, wildcards or non root-to-leaf queries, when their component queries are cached in a subtrie. As above, fragment restoring is applied for the component queries, and when paths of component queries are replaced the corresponding trie node is notified.

4.3 Proactive Replacement

When a leaf of the prefix trie is full and another query must be inserted in it, the leaf must be split regardless the UVs of the queries stored/indexed in it. A problem with this approach is that even leaves containing some queries/fragments with low UVs must be split when they overflow. This leads to unnecessary leaf splitting, since the trie does not adapt to the *real* query workload, and consequently to an unnecessary trie expansion which entails unnecessary additional movement of fragments in DataCache, i.e. an increased maintenance cost. The above may be avoided, if before splitting a leaf, some form of proactive replacement of paths in the cache of the peer storing the leaf is performed. By doing so, stale paths will be evicted, probably making the leaf actual size lower than C.

Algorithm 4 Calculation of UV_{avg}

Initially $UV_{curr} = UV_{last} = UV_{prev} = UV_{avg} = 0$
Every time proactive replacement is executed at a peer:

```
UV_curr = avg(UVs)
if UV_prev = UV_curr then
    UV_last = UV_last + (UV_curr/100) · f
    UV_avg = UV_last
else
    UV_last = UV_curr
    UV_avg = UV_curr
end if
UV_prev = UV_curr
```

To achieve proactive replacement, before performing a split each peer must check the part of the cache it stores for paths that can be evicted based on their UVs. As opposed to cache replacement where the number of fragments to be replaced is determined by the space that needs to be freed, the number of fragments to be replaced is not fixed. Instead, we base the replacement decision on the relative values of the UVs stored at the leaf. Specifically, we define a UV threshold such that any path having a UV lower than the threshold will be replaced. Setting a global threshold value for all peers is not sufficient, since such a value may be high for some parts and low for some other parts of the cache. Therefore, the threshold must be defined locally by each peer according to the UVs of the paths it stores. To achieve this, the threshold τ is defined as a percentage over the UV_{avg}, which is the average of all the UVs of the paths stored at a peer. Paths having UVs lower than the $\tau\%$ of the 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 the needed property when all the UVs are high but not when the UVs are all low. All UVs being low means that none of the paths is accessed recently hence UV_{avg} will remain the same for a long time for this peer. To make the proactive replacement of these paths possible the UV_{avg} is increased by a factor f every time proactive replacement is performed in the peer. This way, after a number of steps, the UV_{avg} will increase enough so that the UVs of some paths will be smaller than $\tau\%$ of the UV_{avg}. Algorithm 4 presents the procedure of choosing the proper value of UV_{avg}. Specifically, if the UV_{avg} is not increased between two steps of proactive replacements, it means that the paths in the peer's cache are not used recently. Thus, the UV_{avg} is increased by f and used for proactive replacement.

Chapter 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 for the experiments are summarized in Table 5.1.

We use a relatively small number of peers, since our scheme is expected to involve peers in close network proximity with each other. The ring size of the DHT has 2^8 virtual peers. Cache size is chosen to be smaller than the total cache space needed (i.e. the sum of all queries that are cached), to permit the evaluation of our replacement policies. An important issue in the case of DC is determining a good value for the maximum leaf capacity C for the prefix trie. IC is not sensitive in the choice of C, since leaves in IC store only queries and not the actual fragments as is the case with DC. We discuss the choice of C further in Section 5.2. All our experiments start with an empty (cold) cache. Regarding the lookup procedure, we use the binary lookup algorithm except from the case when we compare binary vs. serial lookup.

Next, we first evaluate the actual benefits of employing a DC or IC cooperative caching scheme over a scheme in which there is no sharing among the local caches of the peers. Then, we identify the parameters that affect the performance of DC and IC caching and present results characterizing their relative performance. Finally, we evaluate our replacement strategies and the design choices we made.

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

Table 5.1: Input parameters

Parameter	Default Value	Range
# of peers	100	20 - 100
Ring size	128	-
Peer cache size	500 Kb	-
Leaf size - DC	25 Kb	2.885 - 182.720 Kb
Leaf size - IC	400 bytes	200 - 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%

other peer. Clearly, the benefits of cooperation depend on the overlap among the query workloads of peers: the larger the overlap among queries at different peers, the higher the benefits of cooperative caching. Our experimental results show that this dependency is linear. In particular, we consider individual caching with a 50% hit ratio. 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 (Figure 5.1 (left)) show that the cache hit ratio increases linearly with the overlap in both the DC and IC case 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. 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 sharing content in the cache (Figure 5.1 (right)).

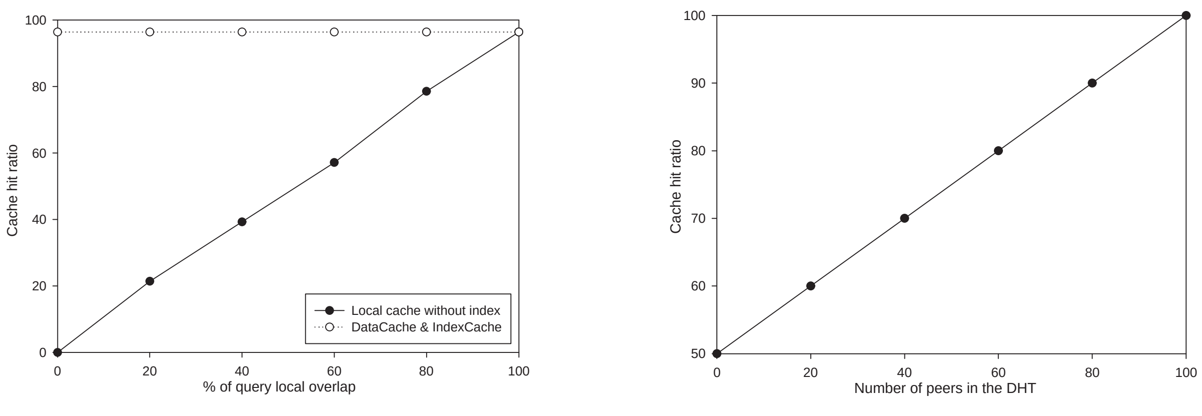

Figure 5.1: Cache hit ratio for increasing query overlap (left) and for increasing number of peers in the cooperative cache (right).

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. We use ρ to quantify that, in particular, we set $\rho = B_C/B_R$, where B_C is the total cost (in bytes) for answering queries from the cooperative cache and B_R the cost (in bytes) required for the same queries to be answered from remote hosts. B_C includes the cost for constructing and maintaining the cooperative cache, the cost for initially transferring the results from the remote host, and the cost of transferring results from other peers participating in caching. B_R includes the cost of transferring the results from the remote host, once for each peer posing the query. Our results (Figure 5.2) show that as the query overlap increases, ρ decreases in both approaches, since the hit ratio increases therefore less results need to be located and transferred from remote data sources. This result leads to the following conclusion: if the cost of locating and transferring a bit from a remote host is more than 25(3) times higher than transferring it from a peer in the cache, it is cost effective to build a DataCache (IndexCache). The situation for DataCache improves fast as the query overlap increases, needing only a 5.33 cost ratio for a 40% query overlap.

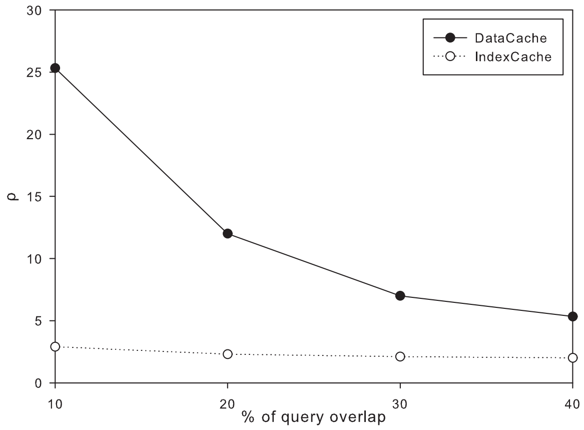

Figure 5.2: ρ for different percentage of overlap among peer query workloads.

5.2 DataCache vs. IndexCache

In this set of experiments, we compare the performance of DataCache (DC) with the performance of IndexCache (IC). We consider the following parameters: (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 in the case of a hit, (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. The experimental results of comparing DataCache vs. IndexCache are summarized in Table 5.2.

Hit Ratio In this set of experiments, we vary the leaf capacity starting from its lower value and increasing it by 50% at each step. Our results (Figure 5.3 (left)) show that as expected the leaf capacity has no impact on the hit ratio for IC, while an increase of the leaf capacity results in a decrease of cache hit ratio for DC. This behavior is due to the way cached fragments are distributed among participating peers. In IC, cached fragments remain in local caches thus their distribution is only influenced by the peer query workload. On the other hand, 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 (more uniform) distribution of cached data among peers. Consequently, peer caches grow more uniformly, leading to fewer replacements, hence higher cache hit ratio. In the next set of experiments, we use this small value as the default leaf size for DC.

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. In this experiment, we investigate the influence of such skew in peer activity. 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 (almost uniform activity rates) to 1 (totally skewed distribution of activities). Our results (Figure 5.3 (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 20% lower than that for DC when $a = 1$.

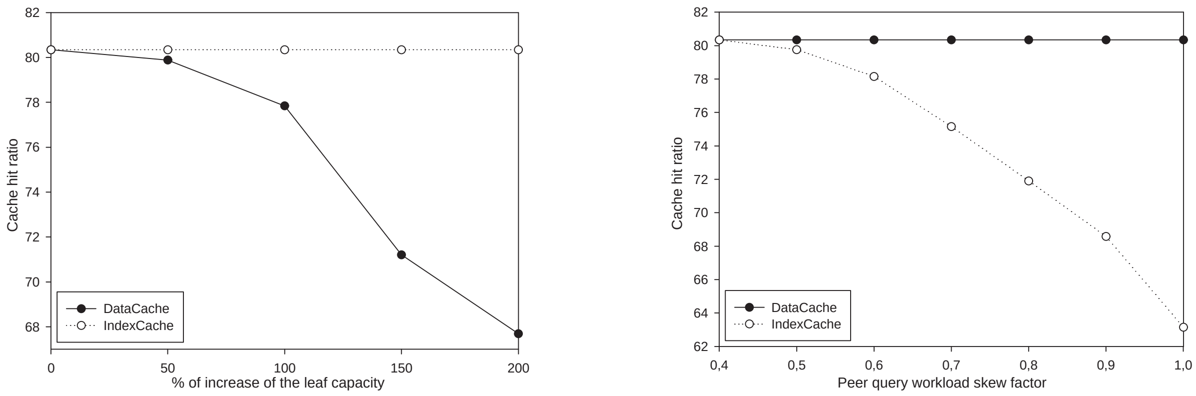

Figure 5.3: Cache hit ratio for different leaf capacity (left) and for different levels of skew in the peer workload activity (right).

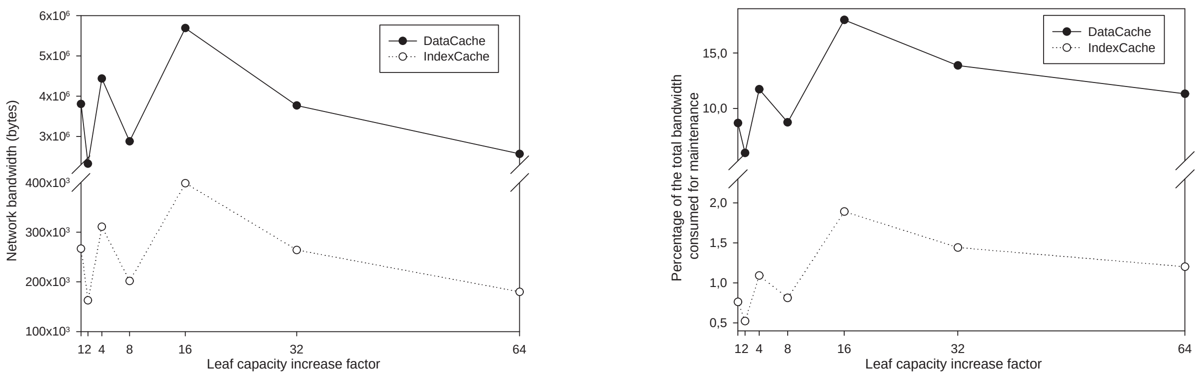

Figure 5.4: Cooperative cache maintenance cost for increasing leaf capacity (left). Percentage of the network bandwidth consumed for maintenance for increasing leaf capacity (right)

Lookup Cost In the IC case, local hits (i.e., when a peer finds results in its local cache) are much cheaper than global hits, since global hits require transferring results from some other peers. This creates the main difference between the lookup cost: IC is better than DC, when there are many local cache hits. Note that both IC and DC have similar hit ratios, what differs is their cost. To quantify that, 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). Our results (Figure 5.5) 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.

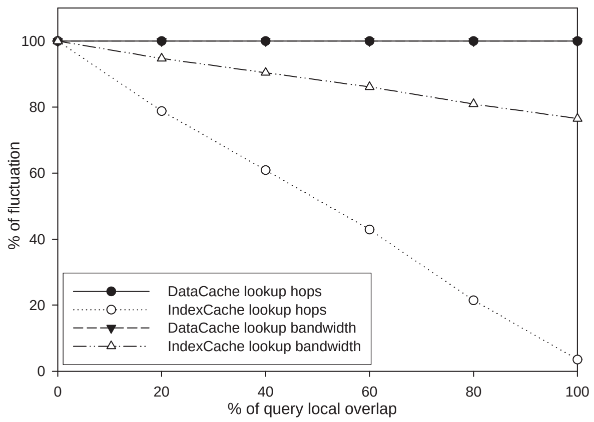

Figure 5.5: Fluctuation of hop and network lookup cost for different percentages of overlap in the queries posed by each peer.

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 less will it access the distributed index (less hops) and the less results will be transferred from the caches of other peers (less bandwidth). On the other hand, for DC, the index is accessed every time a query is posed and almost always the result is transferred from other peers.

Maintenance Cost In IC, the maintenance cost involves just managing the distributed index of cached results, 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 2885 *kbytes* to 182720 *kbytes* for DC and form 200 to 12800 for IC by doubling the leaf capacity in each step. Note, that, in DC, the leaf includes both the queries and their results, while in IC, the leaf includes just the queries. The variation in leaf capacity corresponds to a variation from 5 to 320 queries indexed/stored per leaf. The capacities are chosen, so that, we get a similar number of split/merge operations in both cases. Our experimental results (Figure 5.4 (left)) show that the maintenance cost for DC is ≈14 times higher than that of IC because the capacities of the leaves that are split or merged are also ≈14 times larger (2885÷200) than those in IC. Furthermore (Figure 5.4 (right)), for DC the bandwidth consumed for maintenance constitutes $8.7 - 18\%$ of the total bandwidth, while for IC only $0.8 - 1.9\%$. The increase in leaf capacity in general reduces the number of splits (which is equal in each step for DC and IC). The fluctuation in Figure 5.4, when the leaf capacity increases from a factor of 2 to 4 and from 8 to 16, is due to the fact that the increase of leaf capacity in both cases does not cause a decrease in the number of splits/merges. Thus, the same number of leaves, but of twice the size is moved between peers.

Query Load As already discussed, in IC cached fragments remain in local caches while in DC the distribution of cached data is performed by the distributed prefix trie and the underlying DHT. Thus, for the case of IC the query load a peer receives depends on the popularity of the fragments it caches locally, while for DC it depends on the parameters that govern the trie (i.e. the leaf size). As a consequence it is expected that by properly adjusting the trie parameters, DC achieves a better distribution of query load than IC, especially when some peers cache very popular data.

To evaluate the above intuition we performed a set of experiments where one peer queries and then caches a number of popular fragments for which 2000 queries are issued by the other peers. 8000 additional queries are issued for randomly selected fragments, other than the popular ones, which are cached by random peers in the system. For each experiment we double the leaf size and record the number of queries that each peer answers. As the results show (Figure 5.6), for IC the peer holding the popular fragments has to answer all the queries for them, which produces a significant load (≈2050 queries). All the other peers in the system answer $75 - 175$ queries. In DC the popular fragments are distributed among a number of peers, thus spreading the load for answering the corresponding queries. The smaller the leaf capacity the better distribution of the fragments and consequently of the load is achieved. This is confirmed by the results, where for the smaller leaf size (Figure 5.6(top-left)) the load of answering the popular queries is distributed among all the peers (resembling a gaussian centered at 150 queries), so that the most loaded peer answers ≈450 queries, while 83% of the peers answer $75 - 225$ queries. As the leaf size increases, the load distribution becomes more skewed (Figure 5.6), so that a smaller number of peers share most of the query load. However, even when the leaf size is increased by a factor of 8, the most loaded peer answers ≈1240 queries which is almost half of the load of the peer caching popular fragments in IC.

We can conclude that by properly adjusting the leaf size in DC the query load is spread evenly among the peers. The benefits of DC on the query load are more evident for peers that cache popular queries, since they are not overloaded by queries issued by other peers in the system as in IC. Thus, DC makes cooperative caching more attractive by avoiding mistreatment due to the popularity of a peer's interests.

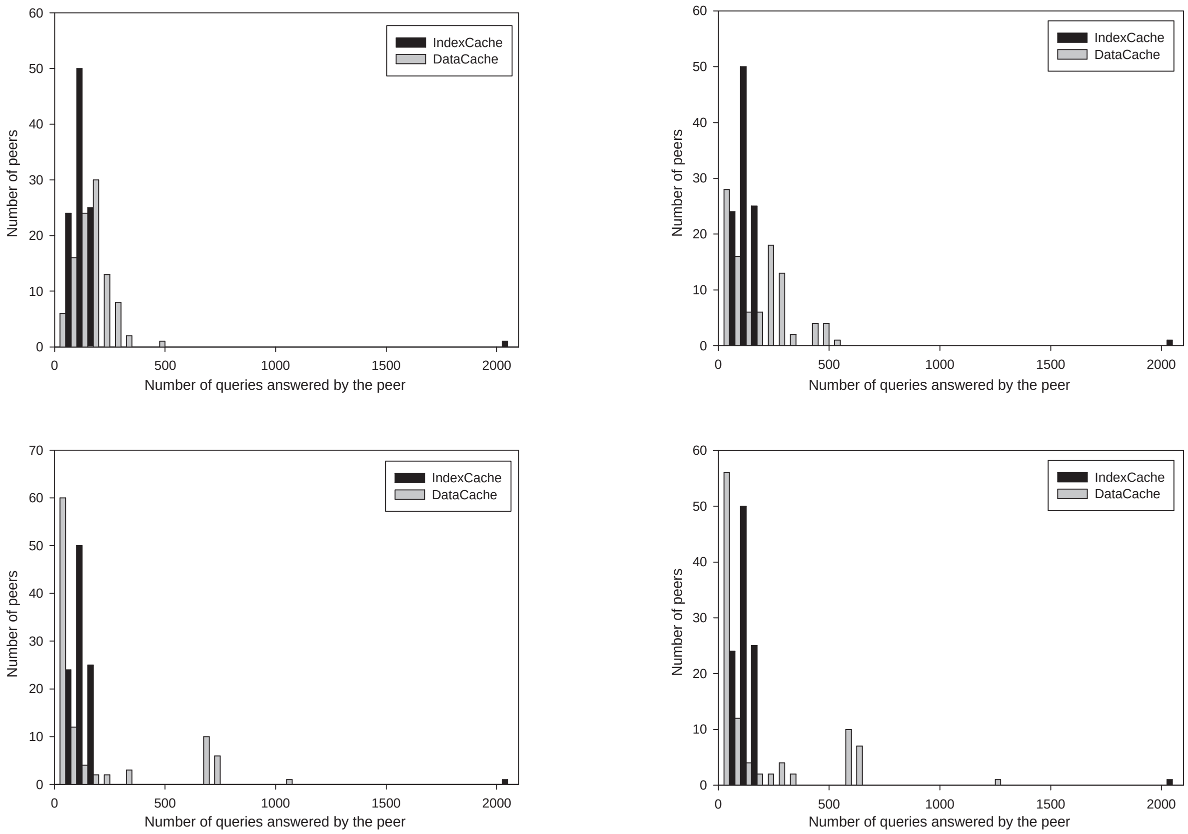

Figure 5.6: Distribution of the queries answered by the peers for increasing leaf capacities by a factor of 1 (top-left), 2 (top-right),4 (bottom-left) and 8 (bottom-right).

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 (utilization value) may

Table 5.2: DataCache vs. IndexCache summary of results

	Hit Ratio		Hop and Network Lookup Cost	Maintenance Cost	Query Load
	skew in query workload	*leaf capacity*	*local query overlap*	*DataCache 14x higher*	*leaf capacity*
IndexCache	$a = 0.4$, hit = 80% $a = 1$, hit = 63%	No Influence	overlap 60% ↑ hops 57% ↓ network 13.9% ↓	$0.9 - 1.8\%$ of the total cost	popular 2000 qrs others 75-175 qrs
DataCache	No Influence	4x leaf capacity hit 16% ↓	No Influence	$8.7 - 18\%$ of the total cost	*1x leaf capacity* most loaded 450 qrs others 75-225 qrs *8x leaf capacity* most loaded 1240 qrs others 75-175 qrs

not be equal to the globally minimum UV. To investigate the effect of this fact, we conducted a set of experiments comparing 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. As already shown, in DC the cache hit ratio is influenced by the leaf capacity, while in IC by the skew in the peer activity rate. Figure 5.7 compares the performance of DC and IC to that of a central cache, with varying leaf capacity and skew respectively. 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. This result scales with the number of peers as Figure 5.8 (left) shows.

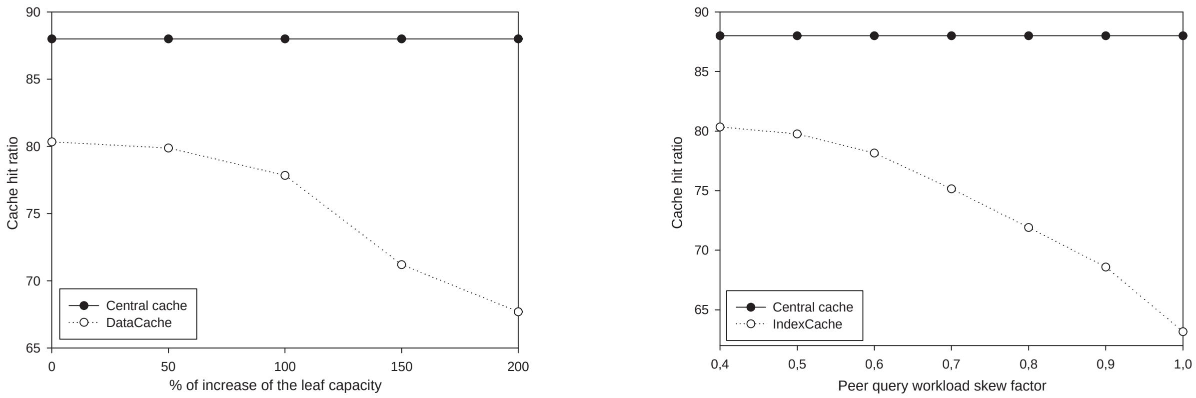

Figure 5.7: Comparison of cache hit ratio between central cache and DataCache for increasing leaf capacity (left) and between central cache and IndexCache for different levels of skew in the peer query workload (right).

Proactive Replacement As already shown, the leaf capacity significantly influences the performance of DC. A large leaf capacity decreases the cache hit ratio, while a small leaf capacity increases the number of splits, hence the maintenance cost. To keep the leaf capacity small and at the same time avoid the maintenance cost, we have proposed proactive replacement. Next we evaluate the impact of proactive replacement on the cache hit ratio and on the network bandwidth overhead caused by leaf splits. We performed two sets of experiments which evaluate the impact of the replacement threshold τ on both metrics. The effects of proactive replacement are more evident when there is a shift of interest

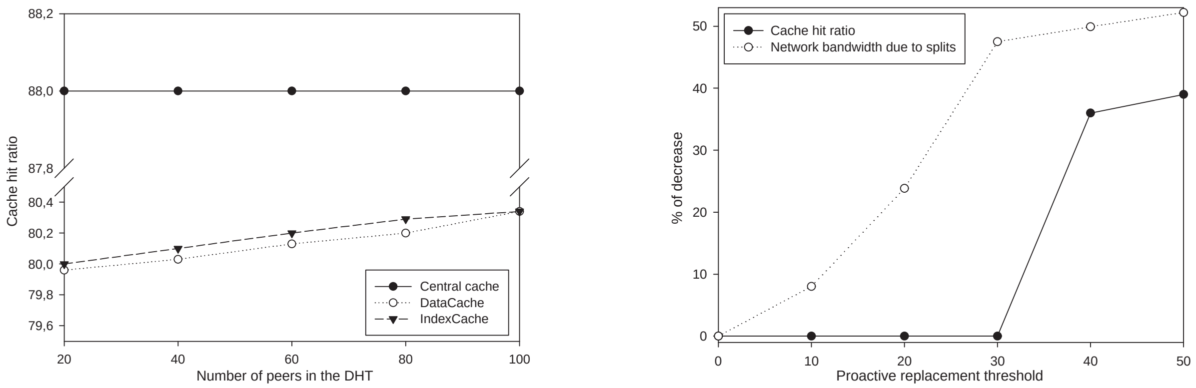

Figure 5.8: Comparison of cache hit ratio between central cache and cooperative cache for increasing number of peers in the DHT (left). Decrease of cache hit ratio and the overhead caused by splits for increasing proactive replacement threshold (right).

in the workload, thus popular items become unpopular and vice versa. For example for a short period of time before the UEFA Champions League final there is an increased interest on soccer. After the completition of the final the topic is not popular anymore and the interest changes to something else (e.g. the Eurovision Song Contest). 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 5.8 (right), for $10\% \leq \tau \leq 30\%$ the cache hit ratio remains constant while the bandwidth significantly decreases ($\approx$52.5% for $\tau = 30\%$). Increasing τ further causes replacement of needed paths decreasing the hit ratio.

5.4 Design Alternatives

5.4.1 NULL Leaves vs. Merge Subtrie

In this set of experiments, we evaluate for DataCache the factors that influence the choice between creating a "$NULL$" leaf (NL) or merging the subtrie (MS) when caching a query that subsumes the queries in the subtrie defined by the query. As already discussed these factors involve the query *lookup cost*: 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, the *load*: the number of queries a peer processes (in the case of a hit) and the cache *hit ratio*.

Lookup Cost When NL is adopted, answering the subsuming query involves visiting all the leaves in the subtrie (broadcasting), while when MS is adopted only one leaf needs to be visited. Thus, the adoption of NL is expected to incur an additional cost (number of hops) when answering such queries. To evaluate this experimentally, we vary the height of the subtrie in which the subsumed queries are cached, from 1 to 4. The increase of the height of the subtrie is achieved by decreasing the length of the subsuming query while keeping the length of the subsumed queries constant. In each case we measure the average number of hops needed to produce the result for a subsuming query in case of a hit. When a miss occurs in NL, the broadcast is not performed as described in Section 4.2, thus the lookup cost in case of a miss is the same for both approaches. The results (Figure 5.9) show that as the height of the subtrie increases, so does the lookup cost for the broadcast. However, the increase is not exponential as would be expected. Specifically, when going from height 1 to height 2, the cost increases $\approx$4 times, while when going from height 2 to 3 and from 3 to 4 the cost increases $\approx$3 and $\approx$2 times respectively.

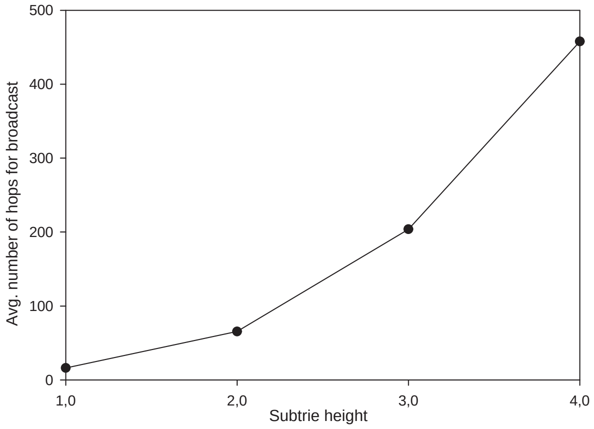

Figure 5.9: Average lookup cost for broadcasting a query in NL.

Load In the case of MS, all the requests for the results previously stored in the subtrie will be answered by the peer holding the leaf that resulted from the merge. This is expected to increase the load of peers holding such leaves. To evaluate this experimentally, we measure the load of each peer for different subtrie heights (1-3). As shown in Figure 5.10, as the height of the subtrie increases the load of the peers becomes less skewed for NL while it becomes more skewed for MS. Additionally, for NL the load values "move" to lower values as the subtrie height increases while they "move" to higher values for MS. In the case of NL, the increase of the height of the subtrie means the distribution of the queries cached in it between more leaves (consequently more peers). Thus, the load spreads better between peers. On the other hand, in the case of MS, the increase in the height of the subtrie means that a larger number of queries is subsumed by the subsuming query. As a consequence more queries will be processed by the peers holding the leaf that results from the merge. Hence, these peers will face the aggregate load of the whole subtrie merged in the leaf they hold.

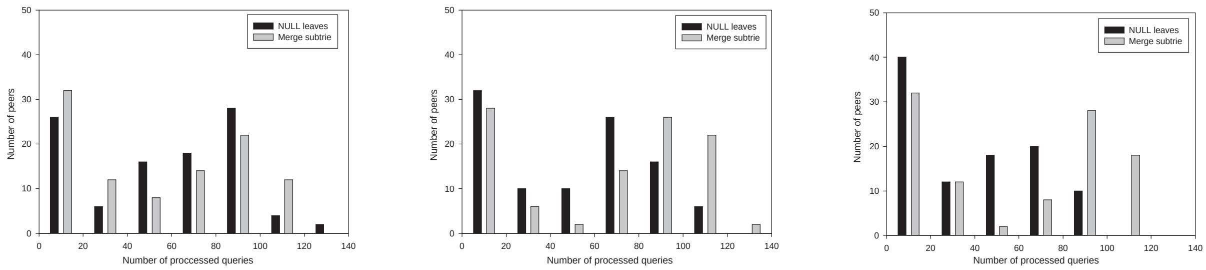

Figure 5.10: Distribution of queries processed by peers in NL and MS for subtries of heights 1 (left), 2 (center) and 3 (right).

Hit Ratio The merging of the subtries in MS creates larger fragments in single leaves in the cache. In NL this is avoided by keeping the smaller fragments in a number of leaves in the subtrie. Smaller fragments in more leaves allow for their better distribution between the peers. Consequently, peer caches grow more uniformly, leading to fewer replacements, hence higher cache hit ratio. To evaluate the above intuition, we measured the cache hit ratio for different percentage of subsumption ($10\% - 50\%$). i.e. 10% subsumption means that 10% of the cached queries are stored in subtries subsumed by a new query. The results (Figure 5.11) 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 merges, and consequently more replacements due to poor fragment distribution.

The above results, advocate in favor of NL. Thus, even if it incurs an increased cost for broadcasting queries, it keeps the load of the peers in lower levels and offers higher cache hit ratio when compared to MS.

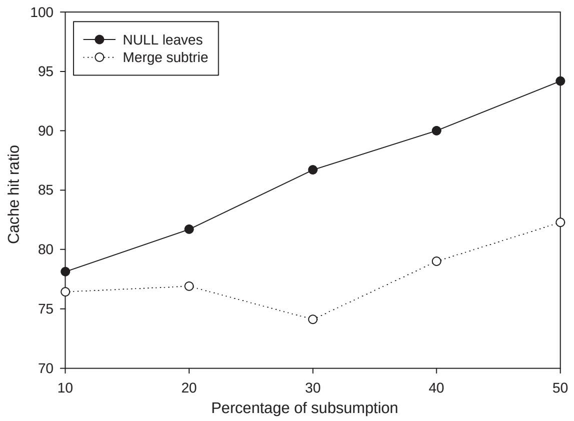

Figure 5.11: Comparison of cache hit ratio between NL and MS for increasing query subsumption.

5.4.2 Binary vs. Serial Lookup

In this set of experiments we compare binary versus serial lookup. The metric for the evaluation is the load, i.e. the number of requests that peers process during the lookup procedure. We measured the load for the nodes of each level of the trie and averaged it on the number of different peers that store them. Figure 5.12(left) shows the average load of the peers for each level of the trie. For serial lookup all the prefixes of a query are looked up, thus all the corresponding levels of the trie are visited. Hence, as shown in Figure 5.12(left), the peers that hold the upper level nodes (i.e. level 1, 2, and 3) receive much higher load than the peers holding the lower levels. Specifically, the peer holding the root of the trie receives almost more load than all the other peers in total. For binary lookup the situation improves dramatically. Hence, the peer holding the root receives a relatively low load, even lower than the load of peers holding nodes in levels 6 and 7 where reside the leaves of the trie (Figure 5.12(left)). Additionally the peers holding nodes in levels 2, 3 and 5 receive notably lower load than the rest of the peers. In general, except for levels 4, 6 and 7 where the load is equal for the two cases, the load for binary lookup is significantly lower than that for serial lookup (Figure 5.12(right)). As a result, we can conclude that binary lookup has notably better load properties than serial lookup, despite the fact that binary lookup in the case of a miss needs to visit some nodes in a trie path (Algorithm 2 line 18). It mainly concentrates the load of requests on the trie leaves (levels 6 and 7) where actual fragments (DataCache) or pointers to the peers' local caches (IndexCache) are stored. In general, the load on the internal trie nodes is kept low except for the "middle" level of the trie where the load is equivalent to that of serial lookup, however still remaining relatively low. This gives us a hint that in case of replication of internal nodes, the ones in the "middle" of the trie are more significant and should be replicated more.

5.5 Discussion

In the experiments we performed we used CHORD as the underlying DHT. How would the choice of another DHT influence our experimental results? As already discussed, the only role of the DHT in the cooperative cache is that of distributing the prefix trie among the peers and, for DataCache, the role of assigning to each peer a part of the query space. Thus, the only characteristic of a DHT

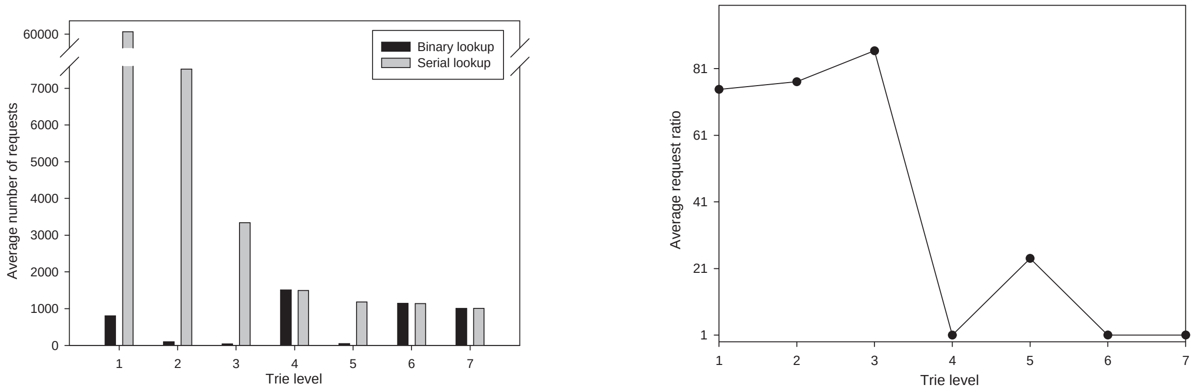

Figure 5.12: Average number of requests served by each peer for different levels of the trie during lookup (left) and the ratio of requests between binary and serial lookup (right).

that would affect our experimental results is how good it distributes data among the participating peers. Specifically, using a DHT that achieves a worse/better distribution than CHORD would decrease/increase the performance for DataCache. The distribution of leaves among peers affects the performance of DataCache in two ways. First, a poor/good distribution decreases/increases the cache hit ratio due to more/lees replacements caused by the poor/good distribution of leaves (i.e. cached fragments). Second, the distribution of load in answering queries for DataCache depends on the distribution of fragments between peers. Thus, a poorer/better distribution of fragments leads to a poorer/better distribution of query load. Consequently, for a different choice of DHT we may observe a greater/smaller quantitative difference in performance between DataCache and IndexCache. However, the trend in our results when comparing the two approaches would remain the same since the parameters that govern the performance of each approach vs. the other are not influenced. Specifically, the skew in query workload and the leaf capacity will still have the same impact on the performance of IndexCache and DataCache. Similarly, the lookup cost for IndexCache will still be lower than that for DataCache since the local hits are not influenced by the DHT. Finally, as shown by the experimental results, the maintenance cost for the two approaches depends only on the leaf capacity, thus the choice of the DHT has no impact on it.

5.6 Summary of Results

In summary, the basic results of our experiments show that:

- As expected, cooperative caching significantly increases the cache hit ratio compared to individual caching, when there is overlap among the query workload of the peers. Depending on this overlap, the benefits of sharing surpass the maintenance overheads for reasonable ratios of remote vs. local communication bandwidth.

- DataCache achieves higher hit ratios than IndexCache, when there is substantial query imbalance among the peers, that is, some peers generate disproportionably more queries than others. Index-Cache supports more efficient lookups when there is high intra-peer locality or overlap and relatively low inter-peer locality. In general, the maintenance overhead of DataCache is more than 10 times that of IndexCache.

- DataCache performance depends on the leaf capacity. A small leaf capacity increases the cache hit ratio but also increases the maintenance cost, since it increases the number of split and merge operations.

- For a smaller leaf size DataCache achieves a singnificantly better query load balance than Index-Cache. The difference is more evident when there is a skew in the popularity of data that peers cache.
- 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.
- The choice of creating a "$NULL$" leaf when a new query subsumes queries cached in a subtrie increases the lookup complexity, while it increases the cache hit ratio and achieves a better query load balance than the choice of merging the subtrie. The aforementioned impact the two approaches have on the cache performance is highly correlated to the height of the subtrie considered.
- Serial cache lookup overloads peers that hold higher level trie nodes, by visitng them in almost every lookup. Binary lookup additionally to requesting less hops for locating a cached result, avoids the overloading of the higher level nodes (i.e. of the peers holding them) by achieving a better load distribution.

Chapter 6

Related Work

In this chapter, related work is presented. First, we present how processing of XML documents is peformed in a local setting. Next, we discuss work that has been done in processing XML documents in P2P systems, distinguishing between structured and unstructured systems. We follow our discussion by examining caching of results of queries on XML data in local databases. Finally, we review work on P2P cooperative caching.

6.1 Centralized Processing of XML Documents

Many approaches exist that deal with processing of XML data in a local setting. Most of them try to improve the query-answering performance by designing an indexing structure with respect to local data processing.

A well-known and influential approach to indexing semistructured data is the DataGuide [20]. A DataGuide is essentially a compact representation of the document tree, in which all distinct label paths appear exactly once. In a DataGuide multiple instances of the same document label path collapse to form a single index label path. Therefore the resulting index tree, which serves as a path index, is usually much smaller than the document tree (although theoretically its size is linear in that of the document tree). Hence it is supposed to be held in main memory even for large document collections. Without references to individual document nodes, however, the index tree only allows to find out about the existence of a given label path, but not its position in the collection. To this end, every index node is annotated with the IDs of those document nodes it represents (i.e. those reached by the same label path as the index node). The annotations of all index nodes are stored on disk in an annotation table. Together the index tree and the annotation table encode nearly all structural information which is present in the document collection. Only parent/child relations between document nodes cannot be reconstructed from the DataGuide, due to the merging of multiple document paths in a single index path.

[34] presents three indexing mechanisms, namely 1-index, 2-index and T-index, aimed to assist in evaluating query paths in semi-structured data. Like DataGuides, 1-indexes are intended to be used by

queries that search the database from the root for nodes matching some arbitrary path expression. A 1-index therefore, represents the same set of paths that DataGuides do, although using a different approach. The basic idea behind the index construction is the generation of a non-deterministic automation to get a more compact structure than dataguides. 2-indexes are intended to be used by queries that search the database for pairs of nodes matching some arbitrary path expression. 2-indexes are constructed in a similar way as 1-indexes. Since 1-indexes and 2-indexes are not tailored to specific paths in the database, when faced with very irregular, cyclic data the index may become too large and inefficient. Restricting the class of queries supported by the index structure can reduce the index complexity and yield better performance. To this end, T-index is proposed. In order to define T-indexes, the concept of path template was introduced. Once defined a path template, a T-index is constructed to support path queries that are instances of the template. Templates are used to guide the indexing mechanism on the more frequently queried part of the database. Like the DataGuide, the indexes proposed in [34] only index the XML document structure.

The Index Fabric [13] is conceptually similar to the DataGuide in that it indexes all raw paths starting from the root element. It encodes paths as strings and inserts those strings into a special index, namely Patricia Trie [26], which is highly optimized for long and complex keys. Patricia tries are unbalanced, main memory structures that are rarely used for disk-based data. However, the Index Fabric utilizes the aggressive key compression inherent in Patricia Tries to index a large number of strings in a compact and efficient structure, and at the same time it is balanced and optimized for diskbased access like B-trees. The fabric uses a novel, layered approach: extra *horizontal* layers of Patricia tries allow a search to proceed directly to a block-sized portion of the index that can answer a query. Every query accesses the same number of layers, providing balanced access to the index. The Index Fabric defines the concept of *raw paths* which index the hierachical structure of the XML data by ecoding simple root-to-leaf paths, and the concept of *refined paths* which are specialized paths through the XML data that optimize special, frequently occurring access paths. Refined paths can support queries that contain wildcards or the descendant axis. Knowledge about the queries and structure of the data is required to create refined paths, while no such knowledge is needed for raw paths. Both kinds of paths are encoded as strings and inserted into the Index Fabric. As a result, Index Fabric can answer general queries efficiently using raw paths, while certain special queries can be accelerated using refined paths. A tree-structured query not in the set of refined paths, however, has to rely on join operations.

XISS [30] is an XML indexing and querying system designed to support regular path expressions. It uses the common representation of XML data objects by tree structures. In each such tree, each node is associated with a pair of integers, based on extended preorder trasversal, enabling the determination of ancestor-descendant relationships among nodes. Each node in the tree (atom expression) is indexed according to its value and the document to which it belongs. Further, all parent-child edges are stored in an index. Processing a query consists of the following steps: (1) decompose the query into parent-child edges or ancestor-descendent paths; (2) for each query edge, use the index to find the corresponding edges in the data trees; (3) for a query path containig a wildcard ($/A/*/B$), locate data node pairs (A, B) corresponding to the ancestor-descendant of the query path and determine whether A is an ancestor of B by using the pair of integers associated with A and B; and (4) combine the results to determine whether there is a match to the query path. One of the merits of the XISS approach is its flexibility, since all kinds of structural queries, including regular expression queries, can be constructed using the most basic building block: the atom expressions.

APEX [12] is an adaptive path index for XML data which has three main goals: First and analog to T-index and Index Fabric, frequently occurring queries should be accelerated more than general queries. Secondly, path expressions starting with the descendant axis that lead to a full evaluation of most structural summaries should be supported in an effective manner. Third and different to the T-Index, APEX can be updated incrementally according to the changes of query workloads. Instead of building an index that is efficient for all possible path expressions, APEX tries to change its structure according to the frequently used path expressions that are determined using data-mining techniques. APEX consists of two structures, a graph G_{APEX} with the structural summary storing references to elements, and a tree of hashtables H_{APEX} representing the incoming paths to nodes of G_{APEX}. Each node of G_{APEX} corresponds to one extent of nodes of the XML data. H_{APEX} is a specific index optimized for frequent queries. When the query workload changes, APEX is incrementally updated minimizing the overhead of reconstructing the index. The order of the hashtables is inverse to the order of nodes in a DataGuide. This means that a query is evaluated from the tail to the head, thus APEX can easily support descendant queries.

The works discussed above pioneered the indexing of semistructured XML data to improve perforlance in the local processing of XML data. However, they are not designed to support a large-scale distributed XML storage due to the large amount of global knowledge they require.

6.2 Distributed Processing of XML Documents

Our cooperative caching scheme provides a scheme for distributing XML documents. There is some work in the context of distributed processing of XML documents. The main focus of this work is the efficient processing of queries on XML documents in a P2P setting. Next we categorize the related work in processing of XML documents on unstructured and structured P2P systems.

Unstructured P2P Systems In unstructured p2p systems, research efforts focus on building space efficient routing indexes for XML documents. Most approaches build path indexes with the use of aggregation and suitable encoding schemes for the paths.

Two architectures for distributing the routing indexes, namely, the open and the agreement model, that differ in the degrees of shared knowledge among the peers, are proposed in [28]. In the open model, each peer is allowed to know about and potentially communicate with every other peer, while in the agreement model, each peer enters into bilateral agreements with some other peers called its neighbors. Path indexes are used as the internal organization of the routing indexes which maintain pointers from each path to the corresponding peers that contain it. Since the information included in a path index can grow excessively, aggregating paths with common prefixes is proposed to accommodate the routing index in a given space overhead.

In [21, 23, 24] the systems use routing indexes that are incrementally updated based on query feedback so as to adapt to changing workloads and topologies. Indexes are used to index elements on instance level and on schema level. Indexing on schema level means the data is identified by the XPath expressions describing the objects. On instance level refers to the physical objects actually stored, using according filter predicates. Path trees are used as routing indexes, which are pruned to accommodate lower memory constraints. Two dimensional histograms are used for the efficient evaluation of predicates, where the first is the classical division into buckets and the second is a time axis of expected returned results within a number of hops.

In MediaPeer [15, 16] every peer that takes part in the network may share XML data by providing an XML abstracted view of the data it is willing to share. This abstraction is a set of tree-like structures, one for each collection of queriable XML documents, and it is called a treeguide. Based on the treeguides, peers are clustered to SuperPeers and SuperPeers may be in turn clustered forming overlay trees. SuperPeers are responsible for the routing of queries and hence maintain a distributed index structure that contains summaries of the relevant paths of the documents, i.e., parts of the treeguides. The treeguides are summarized as tries for being able to index more XML paths. In a trie that indexes the treeguide of an XML document, the vertices represent partial (or entire) strings of element names and the nodes give information about the position of the element names in the paths. The adaptation of a trie also consists in adding to every node the identifiers of the peers that contain the path (starting from the root). An integer is used for representing the ID of a peer and the translation to the real IP address is made using a separate redirection table.

In [27], each peer maintains a local index, summarizing its local content and one or more merged indexes summarizing the contents of its neighbors. The peers form hierarchies in which each peer stores summarized data for the peers belonging to its subtree. The roots (SuperPeers) are interconnected and store additional summaries for all other roots. Each peer that receives a query first checks its local index for any matches. Then, if there is no local answer, it checks the merged index to find the appropriate peer to forward the search. The indexes used are based on Bloom filters that are compact data structures for the representation of a set of elements. To support the evaluation of regular XPath expressions, multi-level Bloom filters are introduced that preserve hierarchical relationships between the inserted elements. These relationships are preserved by inserting the elements of the XML tree to a different level of the filter according to their depth in the tree (Breadth Bloom filters), or by using paths of different lengths as keys and inserting them to the corresponding level of the filter according to their length (Depth Bloom filters).

All the above indexing and clustering techniques proposed for processing XML documents, provide no bounds on the search complexity. Depending on the query workload, the complexity may reach values close to flooding. Additionally, content-based clustering of peers is not appropriate for indexing cached XPath queries since peers' caches change dynamically, thus content is highly dynamic. The above observations piont out that unstructured P2P are not the appropriate choice in the context of building and maintaining a cooperative XML cache.

Structured P2P systems Structured P2P systems are self-organizing, requiring no centralized authority or manual configuration. Most importantly, they are scalable in the sense that both latency (in terms of the number of hops per lookup) and the local state required typically grow logarithmically in the number of peers. An important issue related to processing of XML documents is providing an appropriate mapping of such documents to peers of the system. Structured P2P systems map and search the documents based on exact-match of key values extracted by the documents. This entails a difficulty in including structural information in the mapping of documents to peers.

In [45] the system presumes that all the peers know the schema of the XML data. It encodes the hierarchical structure information of the distributed XML data in an overlay network. The overlay network is a virtual network consisting of logical nodes each corresponding to a coordinate in a multi-dimensional Cartesian space (similar to CAN). The multi-dimensional space and the coordinates in it are defined based on the hierarchical structure information of the paths published by the peers; logical nodes in the overlay network are connected to each other based on their corresponding geometric relationship in the multi-dimensional Cartesian space. To be more specific, each dimension of the multi-dimensional space corresponds to either a path level (i.e. a level of the path expression corresponding to element names), or a unique attribute name on a specific element path level. The entire coordinate space can be represented as a hyper-rectangle with dimension D, D being the number of levels and attributes in the schema. Each distinguished path corresponds to a logical node in the overlay network. The overall hyper-rectangle is disjointly partitioned among sub-hyper-rectangles, and each sub-hyper-rectangle corresponds to exactly one logical node whose coordinate is contained in the sub-hyper-rectangle. Queries containing only the child axis and consisting of all the levels and attributes of the schema, are answered by defining the coordinates of the logical node (by hashing each level and attribute) and routing to it similar to CAN. Queries containing the descendant axis or wildcards are transformed into one or more absolute paths based on the known schema and the same mechanism as before is employed in answering them. Finally, for queries that do not contain all levels or attributes, the missing dimensions are replaced by a default value and the queries are answered by propagating the query in a part of the overlay. The indexing described in [45] has a number of drawbacks. The most important one is the demand all the peers to know the schema of all the data. Additionaly, answering queries with missing dimensions (for example queries in a subsuming relation with stored fragments) becomes more costly as the number of missing dimensions increases.

XP2P [7] extends Chord to support XML data. The system assumes that each peer stores a set of

XML fragments (subtrees of XML data). In addition, each peer stores the local content of the user's fragments and their related path expressions that are the lists of each fragment's child fragments and their super fragment (a path expression of the fragment which is the ancestor of the current fragment). These expressions are hashed into the Chord virtual space. The hashing technique used is different from that used in Chord. In particular, a fingerprinting technique is proposed based on [37]. The produced fingerprints are shorter than the hash keys used in Chord and support a concatenation property that allows the computation of the tokens associated with path expressions to proceed incrementally. Partial and full match lookups are supported, where in the first case, a match to a fragment is returned without unfolding its child fragments, while in the latter case, all the sub tags of the fragment are unfolded and the corresponding child fragments are retrieved. The queries are fingerprinted as well and when the fingerprint of a query (either in full or partial lookup) matches the fingerprint of a data fragment, the results are located by the lookup functionality of Chord. If the system cannot find a match, for instance if the query can be answered by a fragment subsuming it, the query is gradually pruned and a bottom-up search for each query prefix is performed. A more costly bottom-up technique is applied for answering queries containing the descendant axis. The above approach is not appropriate for the purpose of a cooperative cache for a number of reasons. First, there hasn't been described any technique for partitioning a result fragment in a number of related paths. Second, lookup and insert operations are not appropriate to avoid duplications, since when inserting a fragment it is not possible to detect if a fragment subsumed by the new one is already indexed. Third, the echaustive lookup of all prefixes of a query overloads the peers to which the DHT assigns shorter prefixes, since they are looked up for almost every query.

In [42] a distributed index for XPath queries, in structured P2P systems is proposed. The index is built over P-Grid [1] which is a structured overlay network based on the DHT approach. In P-Grid a binary trie is maintained in a distributed way, such that each peer is associated to a root-to-leaf path of the trie and is responsible for storing references to the peers that store data items indexed by keywords for which the path is a prefix. The keys consist of binary strings extracted by the data being indexed, using a prefix-preserving hash function. For inserting a path expression all its suffices are hashed and inserted in P-Grid. Each data item associated to the suffices stores the original fragment and the URI to the source XML document/fragment. For answering a query the longest simple path (containing only the child axis) is hashed and looked up in P-Grid. Broadcasting in sub-tries is deployed to retreive all the results. The final results are checked against the original query before being returned. It is based on a distributed binary tree and queries are routed based on their prefixes which is a desired property for XML documents.

In [19], a distributed catalog framework based on a structured p2p system is proposed. The system uses Chord as the overlay network. The distributed catalog stores sets of key-summaries information for all the peers. The keys for XML data are either element or attribute names. The summaries that correspond to each key are either structural or value summaries. The structural summaries of a key are all possible paths leading to that key. For each new peer that enters the system, each key-summary pair is inserted in the system by the Chord protocol according to the hash values of the keys. The structural part of the query is handled using the structural catalog information, while the value predicates use the value summaries. For query routing, first all the simple paths are extracted from the query. The peer responsible for the last element of each simple path is found and the set of candidate peers for the query are retrieved using the catalog of that peer. The intersection of all the candidate peers that are produced after all the simple paths are processed is the set of peers that should receive the query. Although ensuring high search speed and the possibility of looking up query prefixes, the approach introduces considerable overhead for popular tags, when the data summary is large. Additionally for a single XML fragment entries are created for each of its prefixes increasing the amount of indexing information maintained. The lookup cost of our distributed prefix trie is similar to that of [19] while avoiding its overhead.

The approaches in [42] and [19] seem attractive for being used for indexing the cooperative cache contents. However, both hash each path multiple times (once for each element or suffix), with the goal of increasing the lookup speed not only for the path itself but also for its subpaths. This is not the goal of a cache index since it is a partial index, in the sense that it only indexes content that was previously returned as a result. Thus, we can not use the result of a cached path query to fully answer subpath queries, since results matching them may not be in cache. Additionally to the above difference, there exist some other, practical problems which make using the two approaches for indexing cache contents less attractive. In [42], the usage of P-Grid requires a restructuring of the overlay to adapt to the indexed dataspace. In a cooperative cache this would occur often, causing a large overhead since the cached data changes dynamically acording to the peers' query workload. Additionally, in [19], for the DataCache approach the catalog framework would have to either store a cached fragment in each peer where is hashed each of its elements creating extensive redundancy, or somehow choose one of the peers and properly modify the index information.

6.3 Client-Server and Local XML Caching

Caching has played a key role in client-server databases because network traffic and slow remote servers can lead to long delays in the delivery of answers. Similarly, for local databases the goal is to avoid expensive query execution in large databases by finding as many answers as possible by quick lookups in small caches. The main focus of work in this area is on determining a containment relation between cached and posed queries, and on determining the best set of queries to maintain in cache for increasing its performance.

[14] proposes a semantic model for data caching and replacement in a client-server database system. Specifically, the data cached at the client is modeled semantically as a set of queries, rather than at the physical level as a set of data pages or tuples. Hence, cached queries and their results can be used to answer subsequent queries by reasoning about their containment relationship. To answer a query, it is split in two disjoint portions: one that can be completely answered using the results present in the client cache (probe query), and another that requires results to be shipped by the server (remainder query). Thus, only results of remainder queries need to be fetched from the server improving query performance. Replacement information in the semantic cache is maintained with *semantic regions* which can be dynamically adjusted to the needs of the current queries. Specifically, the result of a cached query defines a semantic region. When a newly cached query Q_1 overlaps with a semantic region Q_2 in cache, the overlaping part and the semantic region defined by Q_1 are coalesced, followed by a shrinking of Q_2 so that it becomes disjoint with Q_1. This dynamic adjustment allows for a fine-grained replacement strategy, which allows recently used results to remain longer in cache. Even if the semantic cache presented in [14] does not concern XML, it has inspired all following works on building semantic XML caches.

ACE-XQ [9, 10, 11] provides a practical framework for XQuery-based semantic caching systems. In other words, it gives the general solutions to the main issues involved in building a semantic caching system for XQuery [6], namely, answering XQuery queries using cached ones, designing an appropriate replacement strategy that improves the cache space utilization, and efficient cached view maintenance upon data updates. The heart of ACE-XQ is the establishment of a containment mapping between the variables in a new XQuery expression and a cached one. Query containment decisions are made depending on whether one-to-one containment mappings can be established. Similarly to [14], based on the containment relationship, the new query is divided into a probe and a remainder query. Additionally, semantic regions are defined by query descriptors of cached views and access statistics are maintained for each of them. Partial replacement decisions (similar to the ones for our cooperative cache) are adopted by utilizing access statistics recorded at the path level to perform a fine-grained region purging, as opposed to a total replacement strategy which replaces a complete query and its associaded cached view. To make better use of cache storage space, "adjacent" queries are discovered and merged into a combined region.

[47] takes a different approach on deciding which queries to be cached. It presents FastXMiner, an algorithm for discovering frequent XML query patterns and cache their results to improve the performance of XML management systems. It considers queries which are regular path expressions or tree patterns with selection predicates on multiple elements. Queries are modelled as trees called *query pattern trees*. Thus, given a set of query patterns, the algorithm determines all the frequent subtrees in the query tree patterns. The determination of the frequent patterns guides the replacement decisions, since the results of frequent queries are favoured by the cache replacement algorithm. Mining the frequent patterns can be expensive since the search space of the tree matching and containment proccess that is employed is exponential to the size of the XML schema. FastXMiner reduces this cost, since only a small subset of the candidate patterns is chosen to undergo costly tree containment tests. Addittionally to mining frequent paths for replacement purposes, [47] presents a heuristic for improving cache space utilization by reducing redundancy of cached data. Specifically, when a new query Q_1 is not fully answered from cache, FastXMiner determines the cached query Q_2 that is most similar to Q_1. Thus, when Q_1 is answered (outside the cache) instead of caching Q_1, the query that results from merging Q_1 and Q_2 is cached.

[32] describes a technique for employing a semantic cache of materialized XPath views. The focus is on detecting when a query can be fully answered using one of the indivdual materialized views and finding a rewriting of the query in terms of such a view. To this end, an answerability criteria is defined to determine when a cached view can be used to answer a new query. Specifically, a query Q can be answered by a cached view V if, after rewriting them in normal form, the path expression of V is a prefix of Q and all the predicates of Q are contained in the predicates of V. This approach, reduces the answerability check in simple string matches and is the approach we adopted in our work. To speed up cache operations the cache itself is stored in a database and cache lookups are performed by issuing SQL queries. To achieve this, information is maintained for the path expressions of the cached views (i.e. their normal form) and their predicates. When a new query arrives, all its prefixes are looked up in the database, starting from the longest, until a matching fragment that also satisfies the predicate condition is found. To further increase cache performance, in terms of hit ratio, a warm-up phase is performed. It involves creating a large set S of queries by generalizing the predicates of the queries in the workload so far. Next, the queries in S that have a higher utility (i.e. the cardinality of the subset of S a query can answer) are proactively cached. Additionally to XPath the semantic cache described can also be used for XQuery, by analyzing them in a set of XPath queries. A problem in [32] is that no discussion is made about avoiding redundancy caused by overlapping cached views for achieving a better cache space utilization.

6.4 P2P Cooperative Caching

The goal of cooperative P2P caching is the usage of results previously computed by other peers so as to avoid costly search, computation or server overloading for recomputing the result and improve response time by bringing the results closer to the requestor. Most of the work done in this area involves cooperative caching of results of complicated-costly queries or cooperative web caches.

In [41] peers cooperatively cache results of range selection queries. The results come from a remote central database with multiple relations whose schema is globally known to all the peers. Hence, the goal of the cooperative caching is to avoid overloading the database and improve response time. A containment relationship is defined between range queries, e.g. the answer of query <25, 30> is contained and can be answer by the result of query <20, 35>. Thus, when a query is issued the cooperative cache is looked up for the exact result or a result that contains the query. To facilitate lookup some of the peers are organized in the CAN overlay. These peers are called *active* and the rest (not patricipating in CAN) *passive* nodes. Each *active* node owns a CAN zone and each *passive* node connects to an active one. Each range query is assigned to a zone (target zone) by hashing the query range. The result of the query is stored at the peer that posed the query and indexed at the peer (target peer) owning the corresponding target zone. If the target peer has space it also stores the query result. The lookup for exact match is performed as in CAN until the target zone of the query is located. It is shown that the result of cached queries containing a posed query is indexed in zones which are at the left or above the query target zone. Thus the lookup for results containing the query, forwards the query in a number of zones in upper and left directions from the target zone.

[5] consideres data items described by an ordered set of attribute-value pairs and caching results (materialized views) of conjuctive/disjunctive queries on them. The queries are of the form $(a \wedge b) \vee (b \wedge c)$ where a, b, c are indexes on the corresponding attributes. If the indexes are located on distinct peers in the network, then computing a query $a \wedge b \wedge c$ directly from the indexes is much more expensive than intersecting the result of a prior $a \wedge b$ operation together with c. Thus, the materialized views of such operations are cached and indexed based on a data structure called *viewtree*. The view tree is implemented as a trie and maintains a distributed snapshot of the set of the currently materialized views. Each level l of the tree corresponds to a materilized view of l attributes. To answer a query the peer storing the corresponding materialized view must be located. This is done by descending from the root of the view tree, in a standard trie manner comparing the label of each node to the query. If a node having the query as a label is met during the traversal then there exists an exact match. Otherwise, the search proceeds from the last located node by visiting only tree nodes that index a view that contains one attribute from the query that does not occur in the views located so far. This way the maximum available result is retreived for the query. To improve the performance of the view tree an effort is made to keep the trie balanced.

In [43] a cooperative cache is maintained for multi-term queries in a P2P environment. The goal is to avoid indexing the whole content of the participating peers and the associated index maintenance cost. Thus, initially the lookup for documents is performed by broadcasting queries. Each peer locally caches query results and an index is built over a DHT, namely P-Grid, by indexing each peer's local cache content through randomly selected terms of the cached queries. Next lookups are performed in cache based on the distributed index and if there is a miss a broadcast is performed. To lookup a query in cache, a DHT lookup is performed for each query term. The corresponding DHT nodes respond with a list of cached result set locations for exact match queries and queries that subsume the posed query. For that purpose a subsuming relation between multi-term queries is defined. The caching and replacement decissions are made based on a $profit$ value which is calculated by the frequency that a query is queried as exact match or subsumption. The needed statistics are collected from cache lookups as well as by listening to broadcasts for queries. A peer caches a query only if there is available space in its cache or the query's profit is higher than at least one of the already cached queries. In the latter case the query with the lower profit is replaced. The approach of [43] resembles our IndexCache, since each peer locally maintains its cached queries and indexes them by the DHT. However, in [43] a simple index is required and subsumption for multi-term queries is easily determined. In IndexCache XPath queries require a more complex indexing technique to achieve efficient lookup and subsumption determination. Additionally, indexing a query by each term incurs considerable overhead for the index.

CoopNet [36] investigates cooperative web caching to avoid the $flash\ crowd$ problem. A flash crowd refers to a rapid and dramatic surge in the volume of requests arriving at a web server, resulting in the server being overwhelmed and response times shooting up. [36] concludes that the constrained resource at the server is not the CPU or the disk, but the network bandwidth at the server. Thus, the goal is to alleviate the bandwidth bottleneck at the server during flash crowds. To achieve this, each peer caches the results of its HTTP requests to the server. These results are indexed so that during a flash crowd other peers can use them, relieving the server. [36] proposes two approaches of indexing the cached results. In the first approach the server itself maintains the index information, by recording for each file the peers that cache it. Thus, when there is a flash crowd the server sends a redirection message to the requesting peers which contains the addressed of some of the peers caching the requested file. The message has a small size, thus it saves the server a significant portion of bandwidth helping it tide over the flash crowd problem. The second approach avoids the server-based redirection which may not alleviate the server in cases of extreme flash crowds. In this approach each peer searches for content among a small number of peers, i.e. its *peer group*. For each URL, there is a designated *root* peer which keeps track of all copies of the file within the group. The assignment of URLs to roots is done using a hash function. Hence, to locate the desired file, the appropriate root is found by hashing the URL and then the peer that caches the file is found based on the root's index.

In terms of building a cooperative cache, perhaps the work most related to ours is Squirrel [22]. In Squirrel, peers in a corporate LAN environment are organized in a DHT (Pastry) to form a cooperative web cache. The goal of the cooperative cache is to reduce the latency observed by web browsers and avoid the cost of a dedicated proxy cache at the corporate boundaries, by exploiting the local caches maintained by web browsers at each peer. Thus, when a peer issues a request for a web object the peers local cache is checked. If a fresh copy of the object is not found in the peers cache, then Squirrel tries to locate a copy to some other peer. It starts by mapping the URL of each document to a peer in the DHT, which is called the *home node* for the object. Two approaches are proposed for what the home-node stores: the actual object or directory information about a set of peers that store the object. In the *home-store* approach, the cached object is stored both at the local cache of the peer and at the home node. After a local miss, the peer sends a request to the home node. If the home node does not have a fresh copy of the object or it incurs a cache miss, the home node contacts the server and receives a fresh copy of the document. This copy is cached at the home node and returned to the requesting peer. Since all requests to the web servers are routed through the home node, the home node always maintains the most up-to-date copy of the objects it caches. In the *directory* approach, peers locally cache the results of their HTTP requests and the home node only stores a directory of pointers to these peers. If a request by a peer incurs a cache miss at the home node, i.e. the home node has no entry for in its directory for the requested object, then the peer itself contacts the web server. After downloading the required object the peer is added at the home node's directory. If the directory of the home node is populated with pointers to peers having stale copies of the object, then, according to the request type, the peer itself or one of the peers in the directory download the fresh copy. In the first case the peer is added at the directory properly updating it, and in the second case the peer receives the fresh copy and it is added at the directory. Finally, if the home node's directory points to peers with fresh copies of the object, the peer's request is redirected to one of the peers in the directory. The peer gets the fresh copy and is added at the directory. The directory approach 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. Additionally, we cache hierachical/semistructured data requiring more sofisticated indexing techniques. These lead to the need of building an index (eg, a prefix trie) that allows checking for subsumption and providing more involved cache operations. Furthermore, there is overlap between cached results.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, we consider a number of peers in close network proximity that pose XPath queries on XML documents located at remote data sources. To avoid the high cost of locating the data and transferring the results to the peers, we propose a framework for building a cooperative cache of XML documents. In this framework the peers cache the results of their queries and share the content of their cache. To facilitate sharing, a prefix-based distributed index is maintained of the cache content. Being prefix-based the index facilitates lookups and eases the determination of subsumption relations between queries. The index consists of a trie which is distributed among peers based on a DHT. The trie is dynamic as it expands and shrinks adapting to the peers' query workload.

We study 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 directory. In the DataCache approach, each peer is assigned and stores a specific part of the data space. We present efficient algorithms involving cache operations (i.e. cache lookup, insert, delete and maintenance) for both approaches and prove their correctness. Initially we consider simple root-to-path queries containing only the child axis and next we extend the cooperative cache operations to support more sophisticated queries, i.e. queries containing the descendant axis or wildcards, and non root-to-leaf queries. We have found that the cooperative cache and its perations are resilient to peer dynamicity by replicating a small amount of critical data, based on the replication algorithms of the underlying DHT.

We have also addressed the problem of optimizing cache space utilization by avoiding redundancy of cached results. Specifically, our cache management guarantees that overlapping results will be cached only once when such overlap is located. Furthermore, in DataCache we guarantee that result overlaps will always be detected whenever they exist, thus avoiding redundancy of cached results throughout the cooperative cache.

We have adopted a fine-grained replacement policy to get the most out of the subsumption relation among queries. Additionally we have proposed a proactive replacement technique which replaces cache entries before storage space is required. Proactive replacement decreases the cost of cache maintenance and leads to a fast adaption of cache content in cases of rapid change in the peers' quey workload.

We present experimental results where we determine the conditions under which the cooperative cache is benefitial versus individual caching. Additionally, we evaluate DataCache and IndexCache and identify the parameters that govern them and the conditions under which one surpasses the other. Next, we show experimentally that replacement decisions in the cooperative cache do not significantly deteriorate cache performance compared to a global cache, under reasonable conditions. Furthermore, proactive replacement decreases the maintenance cost in DataCache when there is a change of popularity in the query workload. Finally, we evaluate experimentally our decisions in cache construction and operations and show that they are the most appropriate in our setting.

7.2 Future Work

There is a number of issues that present topics for future research. One of them involves handling updates so that the cache content is always fresh. A practical approach for handling freshness is associating a time-to-live with each cache entry. After this time expires for a cache entry, the entry will be evicted from cache. The challenge is how to determine an appropriate time to live based on the type of the query and the sensitivity of the cached data. Another issue for future research includes using the proposed framework to build a distributed native XML database. Thus, instead of caches, peers share the XML content of their databases. Database content is expected to be less dynamic than cache content, thus the cost of the distributed index construction and operations is expected to be smaller. However, our index is a partial one, it only indexes and processes queries on content that was previously returned as a result. For indexing and querying full content, as in the case of a distributed database, a number of modifications are required on the index and its operations. Another interesting topic for future work is using our infrastructure for P2P content sharing in a hybrid P2P infrastructure similar to [31]. In the hybrid approach only rare and thus difficult to find items are indexed, while for popular items flooding is employed. This approach aims at combining the efficiency of flooding for locating highly replicated items with the efficiency of DHTs for locating rare items. Thus, our distributed index would be ideal for indexing a small amount of rare XML data, since this would demand a small maintenance cost.

Bibliography

[1] Aberer, K. *P-Grid: A Self-Organizing Access Structure for P2P Information Systems*. In CoopIS 2001, 179 - 194.

[2] Aboulnaga A., Naughton J.F., and Zhang C. *Generating synthetic complex-structured XML data*. In WebDB 2001, 79-85.

[3] Balmin, A., Ozcan, F., Beyer, K., Cochrane, R. and Pirahesh, H. *A Framework for Using Materialized XPath Views in XML Query Processing*. In VLDB 2004, 60-71.

[4] Berglund, A., Boag, S., Chamberlin, D., Fernandez, M.F., Kay, M., Robie, J.,and Simon, J. *XML Path Language (XPath) 2.0*. Technical Report W3C Working Draft, Version 2.0, World Wide Web Consortium, December 2001. http ://www. w3. org/TR/xpath20/.

[5] Bhattacharjee, B., Chawathe, S., Gopalakrishnan, V., Keleher, P., and Silaghi, B. *Efficient Peer-To-Peer Searches Using Result-Caching*. In IPTPS, 2003.

[6] Boag, S., Chamberlin, D., Florescu, D., Robie, J., Simeon, J., and Stefanescu, M. *XQuery 1.0: An XML Query Language*. Technical report, W3C, 2001.

[7] Bonifati, A., Cuzzocrea, A., Matrangolo, U., and Jain,M. *XPath Lookup Queries in P2P Networks*. In ACM WIDM Int. Work., 2004, 48-55.

[8] Chawathe, Y., Ramabhadran, S., Ratnasamy, S., LaMarca, A., Hellerstein, J., and Shenker, S. *A Case Study in Building Layered DHT Applications*. In SIGCOMM 2005, 97-108.

[9] Chen, L., Wang, S., Cash, E., Ryder, B., and Rundensteiner, E. A. *A Fine-Grained Replacement Strategy for XML Query Cache*. In ACM WIDM 2002, 76-83.

[10] Chen, L., and Rundensteiner, E. A. *ACE-XQ: A CachE-aware XQuery Answering System*. In WebDB, 2002, 31 - 36.

[11] Chen, L., and Rundensteiner, E. A. *Xquery containment in presence of variable binding dependencies*. In WWW ”05, 2005, 288297.

[12] Chung, C.-W., Min, J.-K., and Shim, K. *APEX: An Adaptive Path Index for XML Data*. In ACM SIGMOD, 2002, 121 - 132.

[13] Cooper, B., Sample, N., Franklin, M. J., Hjaltason G. R., and Shadmon M. *A Fast Index for Semistructured Data*. In VLDB, 2001, 341 - 350.

[14] Dar, S., Franklin, M. J., and Jonsson, B. *Semantic Data Caching and Replacement*. In VLDB 1996, 330-341.

[15] Dragan, F., Gardarin, G. and Yeh, L. *MediaPeer: a safe, scalable P2P architecture for XML query processing*. In Sixteenth International Workshop on Database and Expert Systems Applications, 2005, 368 - 373.

[16] Dragan, F., Gardarin, G. and Yeh, L. *Routing XQuery in a P2P Network using Adaptable Trie-indexes*. In IADIS, 2005.

[17] FastTrack Peer-to-Peer technology company. *FastTrack*. In http://www.fasttrack.nu/, 2001.

[18] Fredkin, E. *Trie Memory*. Communications of the ACM, 3(9):490-499, Sept. 1960.

[19] Galanis, L., Wang, Y., Jeffery, S. R., and DeWitt, D. J. *Locating Data Sources in Large Distributed Systems*. In VLDB 2003, 874-885.

[20] Goldman, R., and Widom, J. *Dataguides: Enabling query formulation and optimization in semistructured databases*. In VLDB, 1997, 436 - 445.

[21] Hose, K., Karnstedt, M., Sattler, K.-U., and Stehr, E.-A. *Adaptive routing filters for robust query processing in schema-based P2P systems*. In IDEAS, 2005, 223 - 228.

[22] Iyer, S., Rowstron, A., and Druschel, P. *Squirrel: A decentralized peer-to-peer web cache*. In ACM SIGACT-SIGOPS PODC 2002, 213-222.

[23] Karnstedt, M., Hose, K., and Sattler, K. *Distributed Query Processing in P2P Systems with incomplete schema information*. In DIWeb, 2004, 34-45.

[24] Karnstedt, M., Hose, K., and Sattler, K. *Query Routing and Processing in Schema-Based P2P Systems*. In DEXA, 2004, 544-548.

[25] KaZaA file sharing network. *KaZaA*. In http://www.kazaa.com/, 2002.

[26] Knuth, D. *The Art of Computer Programming, Vol. III, Sorting and Searching, Third Edition*. Addison Wesley, Reading, MA, 1998.

[27] Koloniari, G., and Pitoura, E. *Content-Based Routing of Path Queries in Peer-to-Peer Systems*. In EDBT 2004, 29-47.

[28] Koudas, N., Rabinovich, M., Srivastava, D., and Yu, T. *Routing XML Queries*. In ICDE, 2004, 844. Routing XML Queries. In ICDE, 2004.

[29] Laoutaris, N., Smaragdakis, G., Bestavros, A., and Stavrakakis, I. *Mistreatment in Distributed Caching Groups: Causes and Implications*. In IEEE INFOCOM 2006, 1-13.

[30] Li, Q., and Moon, B. *Indexing and Querying XML Data for Regular Path Expressions*. In VLDB 2001, 361-370.

[31] Loo, B.T., Huebsch, R., Stoica, I., and Hellerstein, J.M. *The Case for a Hybrid P2P Search Infrastructure*, In IPTPS, 2004, 141-150.

[32] Mandhani, B., and Suciu, D. *Query Caching and View Selection for XML Databases*. In VLDB 2005, 469-480.

[33] Miklau, G., and Suciu, D. *Containment and equivalence for a fragment of XPath*. In J. ACM 51(1), 2004, 2-45.

[34] Milo T., and Suciu D. *Index structures for path expression*. In ICDT, 1999, 277 - 295.

[35] Open Source Community. *Gnutella*. In http://gnutella.wego.com/, 2001.

[36] Padmanabhan, V.N., and Sripanidkulchai, K. *The Case for Cooperative Networking*. In IPTPS, 2002, 178 - 190.

[37] Rabin, M. *Fingerprinting by Random Polynomials*. Technical report, CRCT TR-15-81, Harvard University, 1981.

[38] Ramabhadran, S., Ratnasamy, S., Hellerstein, J. M., and Shenker, S. *Brief Announcement: Prefix Hash Tree*. In ACM PODC, July 2004.

[39] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. *A scalable content-addressable network*. in SIGCOMM 2001, 161-172.

[40] Rowstron, A., and Druschel, P. *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems*. In IFIP/ACM International Conference on Distributed Systems Platforms, 2001, 329-350.

[41] Sahin, O.D., Gupta, A., Agrawal, D., El Abbadi, A. *A Peer-to-Peer Framework for Caching Range Queries*. In ICDE, 2004, 165-176.

[42] Skobeltsyn, G., Hauswirth, M., and Aberer, K. *Efficient processing of XPath queries with structured overlay networks*. In ODBASE'05, vol. Springer Verlag 3761/2005 (2005), 1243-1260.

[43] Skobeltsyn, G. and Aberer, K. *Distributed Cache Table: Efficient Query-Driven Processing of Multi-Term Queries in P2P Networks*. In P2PIR 2006, 33-40.

[44] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. In ACM SIGCOMM 2001, 149-160.

[45] Wang, Q., and Ozsu, M.T. *A Data Locating Mechanism for Distributed XML Data over P2P Networks*. Technical Report CS-2004-45, University of Waterloo, October 2004.

[46] World wide web consortium. *Extensible markup language (XML) 1.0 (second edition)*. http://www.w3.org/TR/REC-xml.

[47] Yang, H. L., Lee, M. L. and Hsu, W. *Efficient Mining of XML Query Patterns for Caching*. In VLDB, 2003, 69-80.

Publications

- Lillis K., and Pitoura E. *Λανθάνουσα Αποθήκευση XPath Ερωτήσεων σε Συστήματα Ομότιμων Κόμβων*. Presented at the 6th Hellenic Data Management Symposium. Jun 5-6, 2007, Athens.
- Koloniari G., Kremmidas N., Lillis K., Skyvalidas P., and Pitoura E. *Overlay Networks and Query Processing: A Survey*. Computer Science Department, University of Ioannina, Technical Report, TR 2006-08, October 2006.
- Papapetrou E., Pitoura E., and Lillis K. *Speeding-up Cache Lookups in Wireless Ad-Hoc Routing using Bloom Filters*. The 16th Annual International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC 2005), Sept 12-14, 2005, Berlin, Germany, 2005, 1419-1423.

Brief CV

Kostas Lillis was born in Gjirokaster in 1975. He is a MSc student in the Department of Computer Science at the University of Ioannina since november 2005. He received his BSc degree in 2005 from the same department with graduation degree 8.40. During the three first years of his studies in the Graduate Program of Computer science he recieved awards for his performance. He obtained a diploma in Tourism Management in 1998 from the TEI of Heraclion receiving full scholarship. From 2005 until now, he is a member of the Distributed Management of Data Laboratory. His research interests include data management in peer-to-peer systems and distributed management of XML data.