Locating XML Documents in a Peer-to-Peer Network Using Distributed Hash Tables *

Praveen Rao *Bongki Moon*

{raopr@umkc.edu,bkmoon@cs.arizona.edu}

Technical Report UMKC-TR-DB-2008-01

Abstract

One of key challenges in a peer-to-peer (P2P) network is to efficiently locate relevant data sources across a large number of participating peers. With the increasing popularity of the extensible markup language (XML) as a standard for information interchange on the Internet, XML is commonly used as an underlying data model for P2P applications to deal with the heterogeneity of data and to enhance the expressiveness of queries. In this paper, we address the problem of efficiently locating relevant XML documents in a P2P network, where a user poses queries in a language such as XPath. We have developed a new system called *psi*X that runs on top of an existing distributed hashing framework. Under the *psi*X system, each XML document is mapped into an algebraic signature that captures the structural summary of the document. An XML query pattern is also mapped into a signature. The query's signature is used to locate relevant document signatures. Our signature scheme supports *holistic processing* of query patterns without breaking them into multiple path queries and processing them individually. The participating peers in the network collectively maintain a collection of distributed hierarchical indexes for the document signatures. Value indexes are built to handle numeric and textual values in XML documents. These indexes are used to process queries with value predicates. Our experimental study on PlanetLab demonstrates that *psi*X provides an efficient location service in a P2P network for a wide variety of XML documents.

Created: March 2008

Revised: January 2009

Department of Computer Science and Electrical Engineering

University of Missouri-Kansas City

Kansas City, MO 64110

* The authors assume all responsibility for the contents of the paper.

1 Introduction

The recent advances in peer-to-peer (P2P) computing have drawn much attention from researchers in various disciplines as well as users surfing the Internet. Although early P2P systems, such as Napster and Gnutella, made their debut as a music file sharing application and are still best known for similar file sharing services, P2P systems are gaining popularity rapidly in entrepreneurial sectors and scientific communities (*e.g.*, the SEED project [25] for sharing genome data among biologists). These systems provide an infrastructure for a wide variety of applications that increasingly rely on decentralized computing, distributed and redundant storage, and self-organizing overlay networks.

Today, the extensible markup language (XML) has become the lingua franca for information representation and exchange over the Internet. Coupled with the growing popularity of P2P systems, XML is commonly used as a data model for P2P applications to overcome the limited expressiveness of queries and to deal with the underlying heterogeneity of data [21]. This is a natural and synergistic combination of XML and P2P, because XML provides a means to represent, index, and query semistructured data, while P2P systems enable peers to share and locate data sources across the network.

Suppose users in a P2P network publish information such as bibliographies, items-for-sale, movie reviews, and personal profiles in XML format. A user can issue the following XQuery query to find the names of people who reside in `Kansas City`.

```
for $d in collection("P2P")
where $d/person[place="Kansas City"]
return <res>$d/person/name/text()</res>
```

In the P2P network, this query can be processed in two phases. In the first phase, all the qualifying peers are located who store any XML document that has a match for the XPath expression `/person[place="Kansas City"]`. In the second phase, the query can be shipped to those qualifying peers and be executed on the matched XML documents, or the user can download these documents and then execute the query locally.

The past few years have witnessed tremendous research endeavors aimed at developing efficient query processing strategies for XML data. In the second phase of the example above, processing the XQuery query can be done efficiently by applying one of the many indexing and pattern matching strategies reported in the literature (*e.g.*, XISS [22], TwigStack [6], PRIX [28, 29]). Therefore, we shift the focus of this paper to the first phase of query processing, and we address the problem of locating all relevant XML documents across an Internet-scale P2P network. Formally, the problem that we address in this paper is stated as follows.

> Suppose P denotes a set of peers, and D denotes the set of XML documents published by peers. Given an XML query pattern q, determine, by *best-effort*, the set of documents $D^{'} \subseteq D$ such that q has a match in each document in $D^{'}$, along with the peer that published it.

We adopt the notion of *best-effort* service, as is traditionally done in the Internet regarding the delivery of network traffic. In the problem statement, we mean by *best-effort* that a search is expected to find as many matching documents as possible. This is because some peers may not be reachable or may fail during query processing. In fact, DHT frameworks [32] rely on UDP (best-effort protocol) rather than TCP for key insert and lookup operations.

Our main challenge is to organize and index XML data in a distributed way such that the peers storing relevant XML data can be located, by contacting as few hosts in the network as possible. While our previous work on PRIX focussed on finding all occurrences of a query pattern in XML documents, *psi*X aims to find relevant documents *by testing the existence of a query pattern* in the documents. Thus, storing and indexing the document summaries, that can be graphs, is sufficient. In this work, we focus purely on *XML pattern matching* and do not consider the use of different vocabularies for generating XML tag names, which may be required for ontology-based searching. The main contributions of this work are as follows.

- We propose a novel signature scheme to summarize XML documents. Each XML document is mapped into an *algebraic signature* that captures the structural summary of the document. A query pattern is also mapped into an *algebraic signature*. The existence of a pattern can be tested by dividing a document signature with a query signature. Our system supports *holistic matching* of a query pattern, which can contain '//' and wildcard '*', without breaking it into multiple simple path queries and processing them by issuing separate lookups.

- To quickly find matching documents, *psi*X organizes the document signatures in a collection (or forest) of distributed hierarchical indexes called *H-index*, built using Distributed Hash Tables (DHT). The index is decentralized and maintains itself gracefully under concurrent writes from different peers. In each hierarchical index (*H-index*), the document signatures of structurally similar documents tend to be grouped together in the same index node. As a result, while locating documents, few peers tend to be contacted. Value indexes are also built to handle numeric and textual values in XML documents and queries.

- We have performed the experimental evaluation of our proposed signature scheme and indexing strategy, using a rich heterogeneous collection of documents and queries in a P2P setup on PlanetLab [26]. We have compared *H-index* with an inverted index approach called *I-index*, which is similar to the one proposed by Galanis *et al.* [14], but stores document signatures instead of document paths.

2 Background and Motivations

Several techniques have been proposed for indexing textual data, numeric data, and XML documents in P2P environments. For content-based full-text search, Tang *et al.* developed a P2P information retrieval system called pSearch [33], in which document semantics are computed by latent semantic indexing (LSI) [11] in a vector space. Although LSI is an effective method for keyword-based information retrieval, it is not well-suited to capture the hierarchical structure of XML documents.

Aberer *et al.* [1] proposed PGrid that builds a trie and clusters semantically similar data, thereby providing in-network indexing. BATON [19], a balanced tree overlay structure, was proposed to support exact match and range queries. Each node of the tree is stored on exactly one peer, and each node has links to its parent, children, adjacent nodes, and selected neighbors at the same level. P-Ring [9] proposes a P2P range index for efficiently supporting equality and range queries. It overcomes the probabilistic guarantees on searches and load balancing provided by PGrid, and the lack of load balancing guarantees in BATON. The above approaches are suitable for linearly ordered domains. However, an ordering of XML documents does not aid the process of twig pattern matching for locating relevant documents.

Viglas [36] proposed schemes to maintain distributed structures such as B^+-trees and heap files in a DHT. Index nodes and heap pages are assigned random ids picked from the DHT's logical address space. They are then stored as key-value pairs. An index or a heap file is accessed by performing DHT lookup operations. Further, for B^+-trees, a peer storing an index node is responsible for splitting it and can pick a random id for the newly created node. A local locking scheme has been suggested to support concurrent operations. A limitation of this approach is that the storing peer may not leave or fail when the splitting operation is being performed. Our *psi*X system also stores index nodes as key-value pairs in a DHT, but node ids are assigned deterministically. In fact, a storing peer can leave or fail during splitting.

P2P indexes have been proposed for multi-dimensional data [23, 15]. In these approaches, the entire multi-dimensional space is partitioned and merged as peers join and leave the P2P system. Liu *et al.* 's [23] approach relies on the existence of more powerful nodes called *super-peers* in the system. These indexes are not suitable for *psi*X, because XML document summaries are not mapped into a multi-dimensional space.

Next, we describe the techniques proposed for XML indexing and pattern matching in P2P enviroments. Sartiani *et al.* proposed a *super-peer* based XML database system called XPeer [30]. More powerful peers take up extended responsibilities for a group of peers. Peers export a summary of their XML data in the form of a tree-shaped DataGuide [17]. Locating data sources is performed in a hierarchical way by matching a query twig with summaries, by traversing the super-peers organized to form a tree until any interesting location is detected. A super-peer based approach is, however, not fully decentralized and requires special nodes.

Several approaches leverage a DHT that provides a fully decentralized overlay network. A distributed catalog framework was proposed for locating XML data sources in a P2P network [14]. Under this framework, element tags are chosen as keys. For each distinct tag, all unique element paths leading to the tag are stored in its data summary. All the key-summary pairs are stored in a distributed hash table (DHT), in a similar way that keys and instances are organized into an inverted list. For a given XPath path expression, the element tag at its leaf position is used as a lookup key in the DHT. Instead of element tags, paths can be used as keys. XP2P [5] also builds on a DHT framework and allows peers to store whole or fragments

of XML documents locally, whose path expressions are encoded by Rabin's fingerprinting method [27] and stored in the DHT. The concatenation property allows the fingerprints to be computed incrementally and facilitates fragment lookup.

Garces *et al.* [16] built a hierarchy of indexes using a DHT containing query-to-query mappings, such that a user can look up more specific queries for a given broader query, thereby refining his or her interests. Skobeltysn *et al.* [31] proposed a path based distributed index for XPath queries. Paths are hashed to obtain keys that are then stored in a structured overlay. Path queries with '//' and '*' are processed by a lookup on a query fragment that only contains '/' axis, and then issuing a broadcast to certain other peers to match the remaining portion of the query. Recently, Abiteboul *et al.* [2] proposed the KadoP system that supports holistic processing of XPath queries and is built atop a DHT. Their indexing scheme is a combination of an inverted index on XML tags and hierarchical indexes to store the positional representation of instances of the tag name. A holistic twig join algorithm [6] is executed after fetching the contents stored for each query node.

Koloniari *et al.* [20] proposed the use of multi-level bloom filters to summarize XML documents. The underlying overlay network is based on network promixity and content similarity among peers. Only linear path queries are handled by their system.

Providing transactional semantics in P2P systems has been a recent topic of interest [3], as existing distributed protocols cannot scale in large dynamic environments. We do not deal with transactional semantics in this work.

Key Motivations

Most of the previous work uses a kind of path summarization technique for indexing XML documents in a P2P environment. Essentially, a complex XPath expression is first broken into simple paths. Each simple path is separately looked up in the P2P network to find the candidate documents and peers. The intersection of the results for each simple path denotes the final result. Such an approach can potentially increase the number of node hops when locating documents. As a result, a summarization technique that allows a query to be processed *holistically* – even in the presence of '//' and '*' – is necessary.

Load balancing is a challenge when inverted lists are maintained. If a tag appears frequently in the data, then its list can become very large. A peer storing such a list would spend more resources to store it. Moreover, this peer will be overloaded if that tag is frequently accessed. Splitting-and-replication of data summaries [14] can avoid extra lookups but consumes more resources, and requires maintenance and extra effort to keep replicas consistent on updates. It is also unclear how this approach would scale when peers can join and leave at anytime.

Rather than storing the entire list of document summaries for an XML element tag on one peer, *psi*X builds a hierarchical index on these summaries (*i.e.*, signatures), and the index nodes are distributed across a set of peers. A similar approach has been used by KadoP [2]. However, in KadoP, '*' wildcards in queries are ignored during processing. Another shortcoming is that the number of DHT lookups depends on the number of nodes in the query. Although parallel lookups can reduce the response time for queries with many nodes, they still increase the network traffic.

Our *psi*X system *summarizes documents* to quickly test the existence of a query pattern in them. The algebraic signatures are indexed. A query is matched in a holistic way, and the number of DHT lookups *is independent* of the number of nodes in a query. Our hierarchical indexes accomodate concurrent write operations by different peers and maintain themselves gracefully.

3 Our Signature Scheme

The concept of document signatures or fingerprints has been well studied by the information retrieval community. In this paper, we present a novel signature scheme that is specifically designed for quickly testing the existence of a query pattern in an XML document. In our scheme, XML documents are summarized and mapped into algebraic signatures. A query pattern is also mapped into an algebraic signature. We also describe how a pattern match can be found using the signatures of the documents and queries.

3.1 Polynomials over Galois Field

Under the *psi*X system, signatures of both documents and queries are constructed based on irreducible polynomials over a finite field [4], so that algebraic operations can be performed on the signatures for indexing and query processing purposes. Similar to prime numbers in a set of integers, irreducible polynomials over a finite field are polynomials that cannot be represented by a product of two or more non-trivial polynomials of degree greater than zero. A finite field or Galois Field of modulo 2 (denoted by GF(2)) has two elements 0 and 1. In this field, the addition and multiplication operations are equivalent to bitwise XOR and AND operations, respectively. Namely, $0 + 0 = 1 + 1 = 0$, $1 + 0 = 0 + 1 = 1$, $0 \times 0 = 1 \times 0 = 0 \times 1 = 0$, and $1 \times 1 = 1$. An irreducible polynomial in GF(2) can be represented by a fixed-length bit string, each bit of which represents the coefficient of a polynomial term. Generally, a polynomial of degree n over a GF(2) is given by $a_0x^n + a_1x^{n-1} + ... + a_n$, where each a_i is either 0 or 1. This polynomial can be stored in $\lceil \frac{n+1}{8} \rceil$ bytes.

Polynomials in a finite field have been well studied in the number theory discipline and have properties similar to integers. In particular, such operations as addition, multiplication, least common multiple (LCM), and greatest common divisor (GCD) can be applied to these polynomials [4]. For example, addition of two polynomials in GF(2) is equivalent to performing a bitwise XOR of their corresponding bit string representations. For a given product of irreducible polynomials, the presence of any particular irreducible polynomial in the product can be tested by a polynomial division. This is because irreducible polynomials behave like prime factors. These convenient properties form the basis of our indexing and query processing strategy. For a given XML document and a query pattern, the occurrence of the query pattern in the document can be determined *by dividing the signature of the document by the signature of the query*.

3.2 Benefits of Irreducible Polynomials

In our scheme, signatures of documents are a product of irreducible polynomials. Rather than storing a signature as a list of irreducible polynomials each using fixed-length words, we represent it as a single polynomial (using product) to be space efficient. One may argue that prime numbers could be used instead, but there are some issues. To organize signatures in a hierarchical index, a way to measure similarity between signatures is necessary. The measure we propose is based on identifying common factors between signatures. However, there is no efficient algorithm for factorizing large numbers. On the contrary, factorization of polynomials in finite fields can be done efficiently using randomization [4]. (We show in Section 4 that this factorization step can be avoided by choosing irreducible polynomials of the same degree. This optimization would not be possible if primes were used.) Moreover, irreducible polynomials can be easily generated [13]. They can be represented and manipulated fairly easily by binary strings and binary operations.

3.3 Signatures for XML Documents

We first consider only the elements and attributes, and their structural relationship for signature generation. Subsequently, in Section 5.4, we describe how values are handled. To capture the structural summary of a document in a polynomial signature, we first define the *structural summary graph (SSG)* of a document.

Definition 1 (Structural Summary Graph) *Given an XML document T, its structural summary graph SSG(T) is a directed graph G = (V, E), where the vertex set V denotes a set of distinct tags in T, and a directed edge from v_1 to v_2 exists in G, if and only if, the tag corresponding to v_1 is a parent of the tag corresponding to v_2 in T. The vertex corresponding to the root of T has a dummy incoming edge.* □

Note that an SSG can be constructed for a collection of document trees that have the same root. A dummy incoming edge is added to deal with a special case when a document contains just a single node. For a document that has a schema (or DTD), its SSG can be constructed from the schema (or DTD). For a schema-less document, its SSG can be constructed by parsing the document itself. An SSG generated by parsing a document is a subgraph of the SSG that would be produced directly from the document's schema (or DTD). (Note that a document conforming to a DTD/schema may only have a subset of element names and attributes from the DTD.) An SSG can form a tree or a (cyclic or acyclic) graph depending

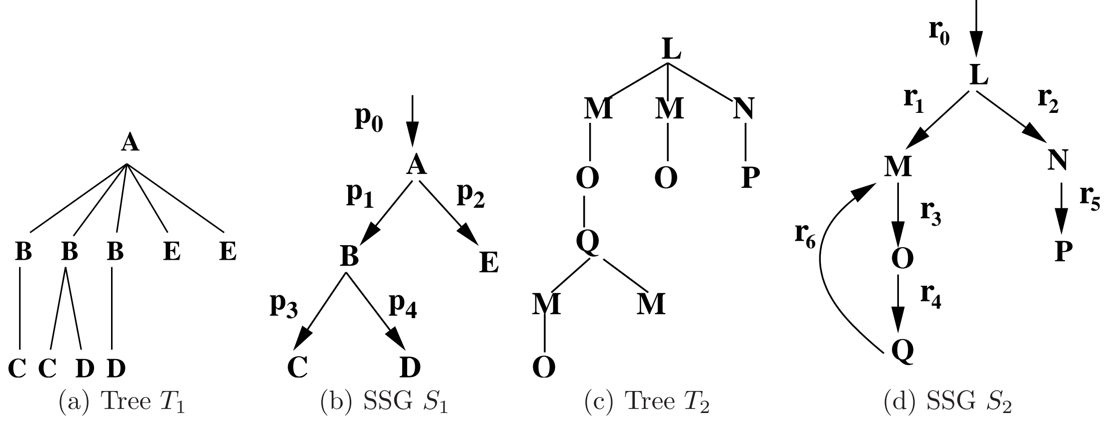


Figure 1: XML documents and their SSGs

on the characteristics of a document. For illustration, two SSGs S_1 and S_2 for document trees T_1 and T_2, respectively, are shown in Figure 1. The constructed SSG resembles a backwards simulation of the input document tree [24].

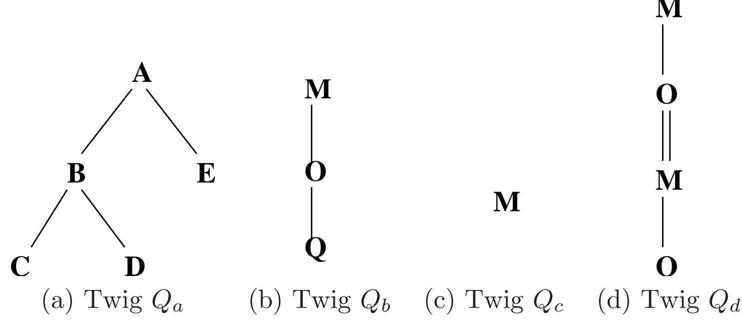
Next, we describe how irreducible polynomials are used to map an XML document into a *polynomial signature*, which is essentially a product of irreducible polynomials. To begin with, we assume that each edge in an SSG is assigned a distinct polynomial. (A method is proposed in Section 5.1.)

Given a document tree T and an SSG S, we use the notation $sig(T, S)$ to represent the signature of T with respect to S, or s for short whenever obvious. A *valid* signature is a non-zero bit string that represents a product of irreducible polynomials. If T does not conform to S, then $sig(T, S) = 0$, because the signature of T is invalid. Unless otherwise stated, all signatures that we refer to in subsequent discussions are valid.

The process of mapping an XML document into a polynomial signature is described in Algorithm 1. Assuming that an input document tree T conforms to an SSG S, T is traversed in preorder. A partial signature s is initialized to a polynomial assigned to the dummy incoming edge in SSG S (Line 1). If T has only one node (*i.e.*, $|T| = 1$), then the partial signature s becomes its ultimate signature. During the preorder traversal of T, each time an edge e_T (associated with a parent-child pair) appears at a certain level (set by the depth of the parent tag in T) for the first time, the partial signature s is updated by multiplying itself with the polynomial assigned to the directed edge e_S in S corresponding to e_T (Lines 3 through 7). (Note that the source (tag) of e_S is identical to the parent tag in e_T and the sink (tag) of e_S is identical to the child tag in e_T.) If e_T appears at k different levels in T, then the polynomial assigned to e_S is multipled with the partial signature k times. By using the level information, signatures of XML documents with recursive tag names are made more precise.

Algorithm 1: Polynomial Signature Generation

```
   procedure GenerateTreeSignature(T, S)
   /* T - XML document tree; S - SSG of T */
1: polynomial s ← p_dum; /* given to the dummy edge in S */
2: if T has one node then return s; /* signature */
   /* Traverse the document in preorder. */
3: for each edge e_T in T do
4:     let l denote the level at which the parent tag in e_T appears in T
5:     if edge e_T appears at level l for the first time then
6:         let p be the polynomial assigned to the directed edge e_S in S corresponding to e_T
7:         s ← s × p
       endif
   endfor
8: return s; /* signature of the document */
```



(a) Twig Q_a (b) Twig Q_b (c) Twig Q_c (d) Twig Q_d

Figure 2: Twig Patterns

Example 1 *We shall construct signatures for the two XML document trees shown in Figures 1(a) and 1(c). Edges of S_1 are assigned distinct polynomials p_0 to p_4, and edges of S_2 are assigned distinct polynomials r_0 to r_6. By applying Algorithm 1, we compute $sig(T_1, S_1) = p_0p_1p_2p_3p_4$, and $sig(T_2, S_2) = r_0r_1r_3^2r_4r_5r_6$. Identical edges appearing at different levels from the root are accounted for by our algorithm. Hence, r_3^2 appears in $sig(T_2, S_2)$. Note that $sig(T_1, S_2) = sig(T_2, S_1) = 0$ since T_1 does not conform to S_2, and T_2 does not conform to S_1.* □

Our signature scheme is flexible because the precision of document signatures can be improved by making the structural summaries more accurate. For instance, consider a document, say D_1, `<A> <B> <C></C> </B> <B> <D></D> </B> </A>`. Consider another document, say D_2, `<A> <B> <C></C> <D></D> </B> </A>`. Using Algorithm 1, $sig(D_1, S_1) = sig(D_2, S_1) = p_1p_3p_4$. However, if a polynomial p_5 is multiplied to the signature when C and D appear as siblings *i.e.*, children of B, then $sig(D_2, S_1)$ becomes more precise.

3.4 Signatures for Twig Queries

Recall that it is of our interest to find documents with structural matches that can be formulated by XPath expressions given in XQuery queries. Since an XPath expression can be mapped to one or more twig patterns, we directly deal with twig pattern queries. Essentially, a *twig pattern* has a defined structure on the nodes in the pattern, and each node has a label associated with it. A twig pattern can be mapped to an equivalent XPath expression.

3.4.1 Twig Queries with Parent-Child Edges

We shall first deal with twig patterns that have only parent-child edges and can occur anywhere in the document. Such a twig pattern can be represented as a labeled tree. Suppose we have a twig pattern Q that is a labeled tree with more than one node (*i.e.*, $|Q| > 1$). Assuming that Q conforms to an SSG S, we can generate $sig(Q, S)$ by applying Algorithm 1, but with Line 1 in the algorithm replaced by the statement $s \leftarrow 1$. On the other hand, if a twig pattern Q consists of only a single node (*i.e.*, $|Q| = 1$), this query needs to be handled differently, because there is no edge present in the query. The signature of a singleton query Q is a list of irreducible polynomials assigned to the incoming SSG edges for the node in Q. If Q does not conform with S, then $sig(Q, S) = 0$.

Example 2 *Let us compute the signatures for the twig patterns, shown in Figure 2(a) and 2(b), using the SSGs S_1 and S_2 in Figure 1. By applying Algorithm 1 to Q_a and Q_b, we obtain $sig(Q_a, S_1) = p_1p_2p_3p_4$, and $sig(Q_b, S_2) = r_3r_4$. Q_c has one node and its signature is a list* i.e., *$sig(Q_c, S_2) = \{r_1, r_6\}$.* □

3.4.2 Handling Twig Queries with '//' and '*'

When Q has '//' and '*', the query signature is a list of signatures. The basic idea to construct signatures for such a query is to resolve '//' and '*' by examining the SSG, but not the data. Suppose we need to compute the signature of `A/B//C/D`. The partial signature for `A/B` is computed. Then in the SSG, we determine each

incoming edge to C via which C is reachable from B.[1] The polynomial assigned to such a qualifying incoming edge is multiplied with the partial signature of A/B, thereby resulting in a list of signatures. Each signature in the list is then multiplied with the polynomial assigned to the edge from C to D in the SSG. To handle a query A/B/*/C/D, we only consider those qualifying incoming edges to C that belong to (directed) paths of length 2 from B to C. We show in Section 3.5 that considering only the incoming edges is enough, instead of considering entire paths when resolving '//' and '*'.

Example 3 *Consider twig Q_d in Figure 2(d). The partial signature of* M/O *is r_3. To resolve '//', the only incoming edge reachable from O to M is edge QM. Thus, we have a list of signatures $\{r_3 r_6\}$. Finally, we multiply each signature in this list with r_3. Thus, $sig(Q_d, S_2) = \{r_3^2 r_6\}$.* □

3.5 Finding a Twig Match

We begin with a theorem that states a *necessary condition* for a twig pattern to have a match in an XML document by examining their corresponding signatures. First, we consider the case when a twig pattern has more than one node, can appear anywhere in the document, and has only parent-child edges.

Theorem 1 *Given an XML document tree T and a twig query tree Q ($|Q| > 1$), and suppose T and Q both conform to an SSG S. If Q is a subgraph of T, then $sig(Q, S)$ divides $sig(T, S)$.*

Proof. It is given that Q has at least one edge. When an edge e appears at level l in Q, it matches only one directed edge in S (Definition 1). Let us call it e_S. The irreducible polynomial assigned to e_S is multiplied with the partial signature while computing $sig(Q, S)$, if it was not already seen in Q. Since Q is a subgraph of T, the edge e appears (at some level) in T and matches the same directed edge e_S in S. If it seen for the first time at that level, then the irreducible polynomial assigned to e_S is multipled with the partial signature while computing $sig(T, S)$. Therefore, every irreducible polynomial included for computing $sig(Q, S)$ is also included for computing $sig(T, S)$. Hence $sig(Q, S)$ divides $sig(T, S)$. □

As mentioned in Section 3.4, if Q has exactly one node or contains '//' or '*', then $sig(Q, S)$ is a list of signatures.

Theorem 2 *Given $sig(Q, S) = \{s_1, s_2, ..., s_n\}$, if Q has a match in an XML document tree T, then for $(1 \leq i \leq n)$, $\exists i$, s.t. s_i divides $sig(T, S)$.*

Proof. Suppose Q has a single node, then $sig(Q, S)$ is a list of signatures that denote the polynomials assigned to the incoming edges of the node in Q in S. If the node in Q appears in T that conforms with S, then at least one polynomial assigned to the incoming edges of the node in S, appears in $sig(T, S)$. Hence, at least one signature in $sig(Q, S)$ divides $sig(T, S)$.

Suppose Q has '//' axis (or wildcard '*'). Then $sig(Q, S)$ is a list of signatures. Suppose '//' is resolved with paths that are parent-child edges using the SSG S. Then Q's signature contains the signatures of all such resolved tree patterns. Let us call them $\{Q_{r_1}, Q_{r_2}, \ldots\}$. If Q has a match in T, then at least one resolved tree pattern Q_{r_i} is a subgraph of T. By Theorem 1, $sig(Q_r, S)$ divides $sig(T, S)$. However, we avoid this method of resolving '//' (or '*') to have a finite list of signatures. Based on our approach mentioned in Section **??**, while constructing $sig(Q, S)$, we consider only the polynomials assigned to the incoming edges of a sink node of '//' (or '*') via which it is reachable from the source node of '//' (or '*') in S. The intuition is that when a source and sink of '//' appear in the document as an ancestor-descendant pair, then at least one incoming edge to the sink from the source in S appears in the document. For every resolved tree pattern Q_{r_i} there exists a signature $s_i \in sig(Q, S)$ that divides $sig(Q_{r_i}, S)$ since the edges considered during the construction of s_i is a subset of the edges in Q_{r_i}. Since at least one resolved tree pattern Q_{r_i} is a subgraph of T, at least one signature $s_i \in sig(Q, S)$ that divides $sig(T, S)$. □

In order to compute a signature for a query with '//' (or '*') such as A//D, it is sufficient to consider only the incoming edges to D reachable from A in the SSG. This is because, if D is a descendant of A in a

[1]Depth-first search can be used to test reachability.

document, then the polynomial assigned to at least one of the incoming edges to `D` (reachable from `A`) (in the SSG) will be present in the document signature. Thus, we do not consider all the distinct (minimal) paths from `A` to reach `D`. Hence, testing for the existence of these polynomials (by division) is enough if `A//D` appears in the document.

Since Theorems 1 and 2 are themselves a necessary condition for a twig match, false matches may occur, but there are no *false dismissals*. Hence, *recall is always one.* We evaluated the precision of the polynomial signatures using SSGs, and observed that our scheme yielded very high precision in all cases. We also constructed signatures by purely considering the distinct tags in documents and queries, and measured their precision. These results are reported in Section 6.

4 Indexing Document Signatures

We propose a collection (or forest) of hierarchical indexes for storing document signatures. We call each index and the collection *H-index*. Our goal is to perform the divisibility tests (Theorems 1 and 2) on a subset of document signatures, by ensuring no false dismissals. We shall first describe a local index.

4.1 *H-index*

For each distinct XML element tag in a document, the document signature in stored in an *H-index* maintained for that tag. (This is similar to the inverted-index scheme [14] that stores, for each distinct XML element tag, the unique paths from the root to that element.) Each node in an *H-index* contains entries of the form (*sig*, *ptr*). In a non-leaf node, *ptr* is a pointer to a child node in the index, and *sig* is the LCM of all the signatures in the child node. In a leaf node, *ptr* denotes a docid, and *sig* is a document signature. What the *containment property* is to the R-tree index [18], is the *divisibility property* to the *H-index*, which is stated next.

Remark 1 *Let s_{parent} denote the signature in a node entry. Let s_{child} denote a signature in the child node pointed by the node entry. If a query signature s_q divides s_{child}, then s_q divides s_{parent}.* □

Remark 1 provides strong leverage to the pruning strategy for query processing, because if a query signature s_q does not divide a signature in a node entry, then this implies that no signatures in the child node are divisible by s_q.

4.1.1 Finding Relevant Documents

Given a query signature s_q, the goal is to find all document signatures in the index that are divisible by s_q. The *H-index* for the target node in the query is first determined. The search begins by first fetching the root node of this *H-index*. For each entry in the node, if its signature is divisible by a query signature s_q, then the child node of the entry is fetched, and the divisibility test is performed again for the signatures stored in this child. This process continues recursively until a leaf node is reached. For each (*docsig*, *docid*) pair stored in the leaf node, if s_q divides *docsig*, then *docid* is returned as a candidate. The search process may traverse along multiple paths in the index to find all the relevant document signatures. If the query signature is a list, then a child node is searched further if at least one signature in the list divides an node entry's signature.

4.1.2 Inserting a Document Signature

A document signature is inserted into an *H-index* by traversing the index from the root. At each non-leaf node, an entry, whose signature has the highest similarity with the input document signature, is chosen as the best child. Consequently, similar document signatures are placed close to each other in an *H-index*, and the corresponding documents are expected to contain common twig patterns. In a P2P environment, this clustering property plays a vital role, since similar document signatures end up being stored across a few peers, and thereby minimizing the time required to locate relevant XML documents.

4.1.3 Measuring Similarity between Signatures

Given a pair of document signatures, we can estimate the similarity between the two corresponding documents using the algebraic properties of the signatures. The intuition is that, XML documents with similar structure and tag names would share common edges from their SSG, and their signatures would share common irreducible polynomials assigned to the edges in the SSG. Therefore, counting the number of irreducible polynomials common to a pair of signatures provides a sense of similarity between their document structures. Below, we define a similarity measure called $PSim$.

Definition 2 (PSim) *Given polynomial signatures s_a and s_b, let g and l denote the number of irreducible polynomials in $GCD(s_a, s_b)$ and $LCM(s_a, s_b)$, respectively. $PSim(s_a, s_b)$ is defined as the ratio $\frac{g}{l}$.*

Example 4 *Let p_0, p_1, and p_2 be irreducible polynomials. Suppose $s_a = p_0 p_1^2 p_2^3$ and $s_b = p_1 p_2^2$. Then, $GCD(s_a, s_b)$ has 3 irreducible polynomials (*i.e.*, $p_1 p_2^2$), and $LCM(s_a, s_b)$ has 6 irreducible polynomials (*i.e.*, $p_0 p_1^2 p_2^3$). Thus, $PSim(s_a, s_b) = 0.5$.*

Based on Definition 2, a higher value of $PSim$ denotes a higher similarity between two signatures. Note that $PSim$ resembles the Jaccard index for measuring the similarity of sets. A signature can be mapped into a set by treating identical irreducible polynomials as different by numbering them with a subscript from 1 to k if an irreducible polynomial has a power k in the signature. Then the LCM between the two signatures is denoted by the product of the irreducible polynomials in the union of these two sets. And the GCD between the two signatures is denoted by the product of the irreducible polynomials in the intersection of these two sets.

Remark 2 *Given two signatures s_a and s_b, and if all the irreducible polynomials of s_a and s_b have the same degree, then $PSim(s_a, s_b)$ is computed by $\frac{deg(GCD(s_a,s_b))}{deg(LCM(s_a,s_b))}$.*

This remark provides a significant opportunity of optimization for determining document similarity. That is, if all the irreducible polynomials assigned to edges in an SSG are of the same degree, then $PSim$ can be computed by computing the polynomial degrees, without incurring a potentially costly factorization step.

Such an assignment of polynomials is feasible, because the number of irreducible polynomials of degree n, denoted as $C(n)$, is about $(\frac{1}{n})^{th}$ the number of all polynomials of degree n [4]. For example, $C(17) = 7710$, which is enough for an SSG with a modest number of distinct edges.

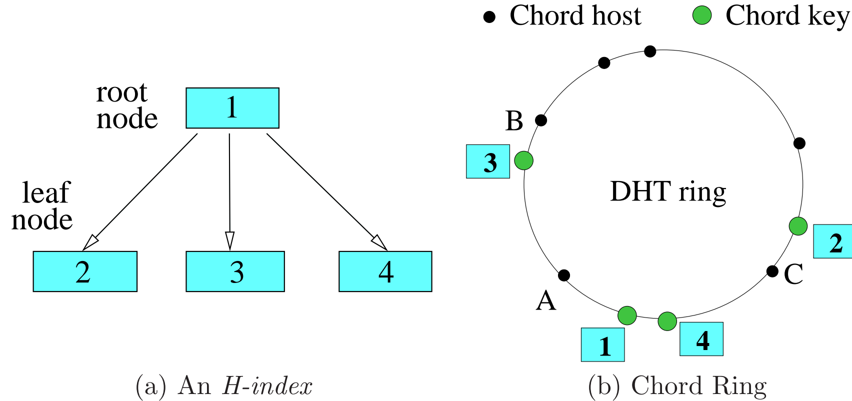
5 Coping with a P2P Environment

In this section, we present our main design principles to cope with a peer-to-peer environment, which include signature generation and indexing document signatures using *H-index*es. Since *psi*X is built on top of Chord [32], a popular distributed hash table (DHT) framework, it inherits the benefits of Chord (*e.g.*, scalability, load balancing, robustness).

5.1 Signature Generation using SSGs

An SSG can be constructed from a DTD, or by parsing the document once if it is schema-less. (Hereinafter, we shall only use the term *SSG*.) In order to assign irreducible polynomials to the edges in an SSG, each peer uses the same algorithm [2] to generate all the irreducible polynomials of degree n – the value n may be chosen by the bootstrap (first) peer or is publically known for that P2P environment. Then, given an SSG S, it is traversed in depth-first order. For each encountered edge e in S, a path from the *dummy* tag up to the sink tag of e is constructed. This path is hashed to an offset $i \in [1, C(n)]$ by using, for example, a combination of Rabin's fingerprinting method [27] and the universal hash functions [7]. Recall that $C(n)$ is the total number of irreducible polynomials of degree n in GF(2). The polynomial at offset i in the list of irreducible polynomials is then assigned to edge e. The signature is computed using Algorithm 1.

[2] In the current implementation of *psi*X, we use an irreducible polynomial generator obtained from the Combinatorial Object Server [13].



(a) An *H-index* (b) Chord Ring

Figure 3: Storing the nodes of an *H-index*

If an SSG is public (*e.g.*, dblp.dtd), then a peer can construct signatures for XML documents that conform to the SSG, or for twig queries on such XML documents. However, if the SSGs of the data (or DTDs), which are much smaller than the actual data, are not public, then they should be advertised (*e.g.*, via the Internet) so that other peers can issue queries to locate such data. The incentive to a publisher, for doing so, is that other peers can now search its data. Thus, given a query, a peer constructs signatures based on the SSGs that it is aware of and are relevant to the query. (The signature for the query can be a list.) For example, a query `//author/name` may be relevant to `dblp.dtd`, `books.dtd`, and an SSG of a schema-less document. Our system is particularly suited for an environment where peers are willing to share their DTDs using a website such as `http://xml.coverpages.org`.

5.2 Signature Generation without SSGs

If a publisher does not want to publish SSGs for its documents, then *less precise* document signatures and query signatures can be constructed, by assigning irreducible polynomials solely based on tag names. For a document, each distinct tag is hashed to an irreducible polynomial, and their product is the document signature. These signatures, thus, ignore the structural relationship between tag names. For example, a document with two nodes `A` and `B` has the same signature, whether `A` is a parent or child of `B`. Note that we consider individual tag names instead of tag pairs in order to support queries such as `A//B`. (Koloniari *et al.* [20] also use distinct XML tags to construct multi-level bloom filters on XML documents for linear path queries.)

A publisher can insert such signatures into the same *H-index*es. A peer can construct a query signature by considering the distinct tags in it and lookup this separate index. We measured the precision of signatures constructed without the knowledge of SSGs. We observed that for 16% of the DTDs (*e.g.*, Treebank), the precision was drastically reduced. (See Section 6.) This reduced precision is a disincentive for a user, and thus, encourages a publisher to advertise the SSG for such documents, and construct their signatures using the SSG. This criteria may not be suitable in an environment where peers may be selfish. Dealing with selfish and malicious peers is a future research challenge.

5.3 Distributed *H-index*

To avoid confusion, we shall use the term *node* to refer to an index node instead of a peer/host in a network. For each distinct XML tag name, an *H-index* is maintained by a collection of peers. Each *H-index* is stored in the Chord DHT. Each Chord peer and key is mapped to a 160-bit identifier. The identifier space forms a ring called the Chord identifier ring. Chord supports *lookup* operations to retrieve the value of a key and *insert* operations to store key-value pairs. Chord allows multiple inserts for the same key and appends the new value to the existing values. For an *H-index*, the *identifier of an index node* is treated as a key, and the *entire content of the index node* as a value. This key-value pair is stored in the DHT. An *H-index* is then maintained and traversed by *insert* and *lookup* operations.

Example 5 *An* H-index *is shown in Figure 3(a). Each index node is assigned a unique identifier. The root node has id "1," and its first, second and third child nodes have ids "2," "3," and "4", respectively. The ptr field of each entry in a non-leaf node is a node id of a child node. Each index node is then stored in Chord as a key-value pair, with its node id as the key and the entire node content as the value. Figure 3(b) shows the Chord identifier ring, where the dark dots denote the 160-bit identifiers of the peers, and the light dots denote the 160-bit identifiers of the signature index node ids. According to the Chord protocol, a key is stored on the first peer that has an identifier equal to or follows the identifier of the key in the Chord ring. In Figure 3(b), the nodes with id "1" and "4" are stored on peer A, the node with id "2" is stored on peer C, and the node with id "3" is stored on peer B. The index can be traversed by issuing a lookup operation for node id "1." The child node id is obtained from the ptr field, and a lookup is issued again.* □

5.3.1 Our Design Rationale

We aim to provide better flexibility to peers while splitting nodes – an improvement over the approach proposed by Viglas [36]. In our system, a peer storing an index node supports only the *lookup* and *insert* operations on that node. This peer can leave the network or fail at any time. It is the responsibility of a publishing peer to successfully complete the insertion and node splitting procedures on an *H-index*. Our system is flexible in the sense that a peer storing an index node can change during the publishing process. To achieve this flexibility, a few challenges should be addressed. Two peers, that attempt to independently split a node and create a new node, should assign the same id to the new node. The index should remain in a consistent state under concurrent operations from peers without using any local locking schemes. (The DHT framework does not provide guaranteed consistency for multiple-writers.) A publishing peer should be able to proceed with the insertion and splitting procedures, even if the peers that store index nodes change during these procedures. Next, we describe our techniques to address the above challenges.

5.3.2 Assigning Identifiers to Index Nodes

We propose a scheme to assign ids to the index nodes in a deterministic fashion. Each id is then hashed by Chord using the SHA-1 algorithm [12] (160 bits). As a result, two index nodes may be assigned the same Chord id, but with a very low probability. Instead, if the SHA-2 algorithm (512 bits) is used, then for all practical purposes, we can ignore the possibility of collisions. (Viglas [36] choose ids randomly from the Chord identifier space, and repick another id if it has already been assigned.)

The root node of an index is special, and its id is computed based on the XML element name. Consider an *H-index* maintained for XML tag `author`. Thus, its root node id is computed using Chord's hash function $h(\text{"}author\text{"})$. For other nodes in this *H-index*, we use simple Farey fractions [8]. Suppose a node is assigned an interval $(\frac{n_l}{d_l}, \frac{n_r}{d_r})$ such that $\frac{n_l}{d_l} < \frac{n_r}{d_r}$, where n_l and n_r are non-negative integers, and d_l and d_r are positive integers. Let l be its level. In our design, each index node has a header that stores its level and assigned interval. Then the node's id is computed by Chord's hash function $h(\text{"}author.l.n_l.d_l\text{"})$ that maps a key to a 160-bit identifier with a very low probability of collision. When a node at level l is split, its interval is split into two non-overlapping intervals, namely, $(\frac{n_l}{d_l}, \frac{n_r+n_l}{d_r+d_l})$ and $(\frac{n_r+n_l}{d_r+d_l}, \frac{n_r}{d_r})$ – *a nice property of Farey fractions.* The first interval is retained by the original node, and the second interval $(\frac{n_r+n_l}{d_r+d_l}, \frac{n_r}{d_r})$ is assigned to the newly created node whose id is computed by $h(\text{"}author.l.(n_r+n_l).(d_r+d_l)\text{"})$. At any level, all the node intervals are contained in the interval $(\frac{0}{1}, \frac{1}{1})$. By design, once a non-root node is created at level l, it remains at level l. Also, the left boundary of a node's interval never changes. In an *H-index*, the leaf node is at level 0, and the root node has the highest level. Each time the root splits, its level increases by one, and two child nodes at the previous level are created. The root's interval is fixed at $(\frac{0}{1}, \frac{1}{1})$, and its child nodes are assigned intervals $(\frac{0}{1}, \frac{1}{2})$ and $(\frac{1}{2}, \frac{1}{1})$, respectively, whose node ids are computed by $h(\cdot)$.

Once a node is created, its id never changes. Note that certain attributes in a node header change due to splitting, and hence, the entire header is not used to construct a node id.

5.3.3 Document Publishing

For every distinct element in an XML document, a peer inserts the document signature and its identifier into its corresponding *H-index*. Note that a document resides at the publishing peer. We assume that this

Table 1: Variants of *insert* and *lookup* operations

Operation	Purpose
$insert_U(k, ., op)$ $op = \{NONE, STRICT, REPLACE\}$	update an index node
$insert_S(k, \ldots)$	split an index node
$lookup_O(k, ., op)$ $op = \{NONE, CHILD, HDR, DIVIDE\}$	fetch specific contents in an index node

document identifier contains the IP address of the publishing peer. When a peer issues a query, it receives a set of document identifiers of relevant documents, and then contacts the publishers for the documents by using the IP addresses. We propose an approach that allows peers to synchronize among themselves during node splits, and this approach ensures that the nodes in the index do not go missing due to concurrent operations by peers. Therefore, successfully inserted document signatures are not lost. Note that, although our techniques allow concurrent operations on an *H-index* and to maintain it gracefully, they should not be regarded as a general-purpose transactional protocol for distributed indexes.

We make an assumption that a peer does not *explicitly delete* an index node. (Chord does not support explicit key deletes.) So an *H-index* never shrinks in terms of the index nodes – *once created, the nodes are never deleted, and they are never merged together* unlike in typical B^+-tree indexes. Document signatures can be deleted from an index node. Each index node is stored as a value list for a key, and space is not preallocated. When a signature is deleted, the updated value list is written back to the underlying storage by the DHT framework. Therefore, space is not wasted. If a leaf node becomes empty, then it is not deleted from the system. Thus, its node id is still accessible, although its content is empty. The parent signature entry is updated to 1. Future insertions choose this entry for breaking ties when selecting the best child node.

For the ease of exposition, we assume that the DHT operations, such as *lookup* and *insert* issued to Chord, complete successfully. (In our implementation, we retry on failures.) Peer failures are handled by Chord via replication of keys. (See Section 5.5.) We also assume that the peers are *non-malicious* and adhere to the protocols described below for insertion and node split. We consider a typical P2P environment where queries from users are more frequent than document publications. In subsequent discussions, we shall frequently refer to a node with id k as node k.

Insertion Protocol In *psi*X, we provide peers with three new operations called $insert_U$, $insert_S$, and $lookup_O$ on a node id. These are variants of the basic *insert* and *lookup* operations provided by the DHT. These operations are invoked by a peer as part of the insertion protocol. The new $lookup_O$ operation is also used during query processing. These operations are listed in Table 1. Using an $insert_U$ operation, a peer can *update* a signature in a node entry to preserve the *divisibility* property, as well as adding a new (sig, ptr) entry to a node. Using an $insert_S$ operation, a peer can *split* a node. The $insert_S$ operation allows multiple peers to synchronize among themselves when splitting the same full node. Using a $lookup_O$ operation, a peer can fetch only certain contents of an index node, rather than reading the entire node and then extracting the necessary content. Thus, a peer storing the index node examines it and returns only the necessary contents to a calling peer. Both $insert_U$ and $lookup_O$ use an *op* parameter that will be explained in the following discussions.

Algorithm 2 describes the case when there is enough room in the index to accommodate a document signature s with docid d. Thus, no node splits occur at any levels in the index. Similar to an R-tree, the best child is picked (Line 1), and when the leaf level is reached, the input $(sig, docid)$ is added (Line 3). (When k is a leaf node, $lookup_O$ returns a *null* child in Line 1.) Then the signature entries along the selected path, from the root to the parent of the leaf node, are updated to preserve the divisibility property (Line 5). (In case of ties when selecting the best child, the entry whose signature has the smallest degree is chosen.)

When an $insert_U$ operation is issued with the arguments k, (s, p), and $NONE$, then the peer storing node k will atomically replace the pair (r, p) in the corresponding value list with the pair $(LCM(r, s), p)$. If such a pair (r, p) does not exist in this value list, then (s, p) is appended to this list. Note that multiple peers

can issue $insert_U$ operations on the same node entry whose signature is r. Their requests will be serialized in the order that they arrive at the peer storing the node.

Algorithm 2: Basic insertion protocol without node splits

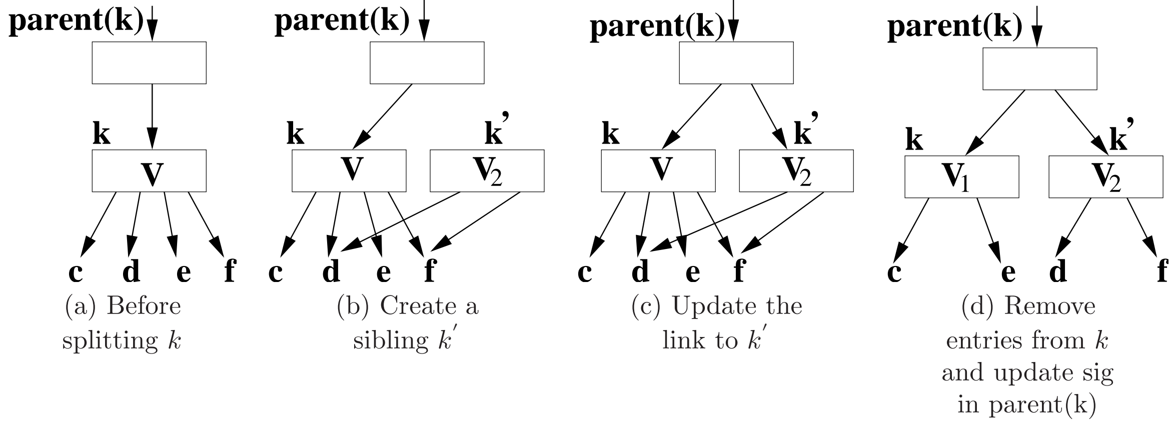
```
proc insertion(k, (sig, docid))
   /* k - index node id; (sig, docid) - signature and document id; */
1:  (nodetype, k_child) ← lookup_O(k, sig, CHILD)
2:  if nodetype = LEAF then
       /* Assuming node is not FULL */
3:     insert_U(k_child, (sig, docid), NONE)
    else
4:     insertion(k_child, (sig, docid)) /* Traverse the child */
       /* Assuming no node splits at lower levels */
5:     insert_U(k, (sig, k_child), NONE) /* Update entry */
    endif
endproc
```

Algorithm 3: Protocol for splitting a non-root node

```
/* This procedure is called by a peer to split a node */
proc splitNode(k, V_1, V_2)
   /* k - key or node id; V_1, V_2 - partitions of node contents; */
1:  (k', h_k', h_knew) ← f(h_k) /* new node id, and headers */
2:  let t and s denote the LCMs of signatures in V_1 and V_2 respectively
3:  status ← insert_S(k', ∅, h_k', V_2)
4:  if status = first then
5:     insert_U(parent(k), (s, k'), NONE) /* create link */
6:     insert_U(parent(k), (t, k), REPLACE) /* replace */
7:     insert_S(k, h_k, h_knew, V_2) /* update hdr */
    else
       /* wait for hdr to change using lookup_O(k, HDR)*/
8:     await(k's header != h_k)
    endif
endproc

/* This procedure is executed by the peer responsible for storing r */
proc insert_S(r, h_rold, h_rnew, V_r)
   /* r - key or node id; h_rold, h_rnew - old and new headers; V_r - list of entries */
9:  if r does not exist then
10:    create a node with id r, header h_rnew, and contents V_r return first; /* first peer */
12: else if r's header = h_rold then
13:    remove V_r from the value list of r, set h_rnew as r's header
14:    return first;
    endif
15: return ¬first; /* not the first peer */
endproc
```

Node Splitting Protocol A node split protocol is initiated by a publishing peer when a leaf node is full and cannot accommodate a new entry. (In our implementation, a caller of $lookup_O$ in Algorithm 2 is informed if a node is full through the return value.) In the subsequent discussions, we emphasize that an *H-index* remains *well-linked* under concurrent operations.



(a) Before splitting k (b) Create a sibling $k^{'}$ (c) Update the link to $k^{'}$ (d) Remove entries from k and update sig in parent(k)

Figure 4: Steps involved during a node split

Definition 3 (Well-linked) *An* H-index *is well-linked if the logical links to every index node of the* H-index*, including any newly created node, can be found by traversing the signature index from the root.* □

When a leaf node is full, the publishing peer determines the set of full nodes, from the leaf to a non-leaf node, that need to be split. The peer issues node splits (Algorithms 3 and 4) in a top-down fashion, starting from the top-most non-leaf node to the leaf, to prepare space before inserting the pair $(sig, docid)$. A reinsertion is done from the parent of the highest-level node that split. The splitting is done this way so that when multiple peers fetch the same full node, they consider the same set of child nodes while partitioning the node entries. If we proceed in the bottom-up fashion, then while splitting an internal node, two different peers may have different newly created child node entries to consider while partitioning the entries in the full node. Hence, the top-down approach simplifies the splitting algorithm.

A publishing peer will issue a $lookup_O$ operation with the $NONE$ option to fetch the entire full node, because Line 1 in Algorithm 2 does not return the entire node. Analogous to splitting a node in an R-tree, a peer (locally) creates a partition of the node entries of a full node using the $PSim$ measure to group similar signatures. Algorithm 3 is invoked for a non-root node. As described in Section 5.3.2, a new node id and new headers are computed (Line 1). The $insert_S$ operation (Line 3) enables peers, that concurrently attempt to split the same index node, to synchronize. The header information of a node and a Boolean handle, returned by a peer that stores the index node, are used.

When a peer issues $insert_S$ on a node k, then Lines 9-15 are executed by the peer responsible for storing k. It atomically does the following two steps: (a) If the node id does not exist, it creates the node. It initializes the node's header and content with the new header value and the (sig, ptr) entries passed by the calling peer, respectively. It returns $first$ as the handle. (b) Otherwise, it checks if the header of the node (that it has) is identical to the current header value passed by the calling peer. If so, it removes the (sig, ptr) entries from the node and updates the header with the new header provided by the calling peer. If the headers do not match, it returns $\overline{first}$ as the handle. (Note that Step (a) enables the creation of a new sibling node, and Step (b) enables the removal of entries from a node that split.) The calling peer, that receives $first$ as the handle, proceeds to complete the splitting protocol (Lines 5-7). The $REPLACE$ option for $insert_U$ (in Line 6) will replace the old LCM of node k with its new LCM t in $parent(k)$. This process is illustrated in Figure 4.

A publishing peer, that received a $\overline{first}$ handle, waits till the header of the node changes (Line 12). It uses the $lookup_O$ operation with the HDR option, which only fetches the header of a node. This lookup operation is performed using an exponential backoff strategy till the header changes. The splitting of a root node is special, and the root id never changes. Algorithm 4 describes the steps involved.

In our splitting protocol, only one peer performs the critical splitting operations on a node, and the remaining peers wait for completion of the splitting operations. By using a handle on a node, and awaiting the change of the node's header, peers synchronize among themselves to split a full node. It is straightforward to show that an *H-index* remains in a *well-linked* state under concurrent split operations by peers.

Theorem 3 *Given an* H-index *that is well-linked, suppose n peers independently and concurrently decide to split the same nodes starting from level l_1 to l_2. If each peer issues a node split and finishes successfully, then the index remains well-linked.*

Proof. We show that starting with a well-linked index, the index remains well-linked after a full node split at a level l by n peers. Then by using the top-most level split as the base case, we prove by induction that the index remains well-linked after splitting the same set of nodes from a level l_1 to l_2 by n peers. Given a well-linked index, we shall consider three cases while splitting a node at a level l in the signature index.

Case (1): All calling peers have a copy of the node k with h_k as the header and V as the node content.
When Algorithm 3 is applied, each peer creates the same partitions V_1 and V_2. Also, they compute the same new node id $k^{'}$, and headers for k and $k^{'}$. The first peer to receive a valid handle creates $k^{'}$ with V_2, updates the link from the parent of k, and then finally removes entries in V_2 from node k. It then changes the header of k. The remaining calling peers lookup k and finish once the header has changed. As a result, the index remains well-linked.

Case (2): All calling peers have a copy of the node k with h_k as the header but different node contents w.r.t. the signatures in the node entries.
When Algorithm 3 is applied, each peer computes the same new node id $k^{'}$, and headers to be created, but different partitions of V. However, the logical links in k are identical. Hence after partitioning V, all the logical links are still present. Since only one peer receives a valid handle, its partitioning of the node entries is adopted during the split, and it completes the critical splitting tasks that leaves the index well-linked. Once the header is changed, all the other calling peers finish successfully.

Case (3): All calling peers have a copy of the node k but with different headers.
When the headers are different, this implies that k has already split after some calling peers fetched k. When such peers try to invoke a split, then the new node $k^{'}$ that they attempt to create exists already and they do not receive a valid handle. When they check the header of k, it is already different from what they have and they finish successfully. One calling peer that has the latest header of k will perform the splitting tasks and finish successfully. Thus the index remains well-linked.

Now we shall apply induction to prove the theorem.

Let $P(t)$ denote the hypothesis "the signature index remains well-linked after top-down splits at t consecutive levels (by n peers) starting from a non-root node".

$P(1)$: the signature index remains well-linked after one split of a non-root node (by n peers).

Basis: $P(1)$ is true based on Cases (1), (2), and (3).

Inductive step: Assume $P(t)$ is true. To prove that $P(t+1)$ is true. Before the index splits the $(t+1)$'th time, by inductive hypothesis, it remains well-linked after the t'th split. Using Cases (1), (2), and (3), it remains well-linked after the $(t+1)$'th split. Hence $P(t+1)$ is true. By the principle of mathematical induction, the index remains well-linked after t (≥ 1) splits of the same set of full nodes starting from a non-leaf node to lower level nodes. □

Theorem 4 *An* H-index *remains well-linked under successful concurrent invocations of a root split by peers.*

Proof. Similar to the proof for Theorem 3, we have a few cases to consider.

Case (1): All calling peers have a copy of the root r with h_r as the header and V_R as the node content.
When Algorithm 4 is applied, each peer creates the same partitions V_{R_1} and V_{R_2}. Also, they compute the same child node ids k_1 and k_2, and headers h_{k_1} and h_{k_2} for the split. The first peer to receive a valid handle creates k_1 and k_2, removes entries from r, updates the links to k_1 and k_2 from r, and finally updates the header of r. The remaining calling peers lookup r and finish once the header has changed. As a result, the index remains well-linked.

Case (2): All calling peers have a copy of the root r with h_r as the header but different node contents w.r.t. the signatures in the node entries.
When Algorithm 4 is applied, each peer computes the same child node ids k_1 and k_2, and headers h_{k_1} and h_{k_2}, but different partitions of V_R. However, the logical links in r are identical. Hence after partitioning V_R, all the logical links are still present. Since only one peer receives a valid handle, its partitioning of the node entries is adopted during the split, and it completes the critical splitting tasks that leaves the index well-linked. Once the header is changed, all the other calling peers finish successfully.

Case (3): All calling peers have a copy of the root r but with different headers.
When the headers are different, this implies that r has already split after some calling peers fetched r. When such peers try to invoke a split, then the new nodes k_1 and k_2 that they attempt to create exists already and they do not receive a valid handle. When they check the header of r, it is already different from what they have and they finish successfully. One calling peer that has the latest header of r will perform the splitting tasks and finish successfully. Thus the index remains well-linked. □

Theorem 5 *An* H-index *remains well-linked under successful concurrent invocations of node split operations by peers.*

Remark 3 *Once a document signature is successfully inserted into a leaf node of an* H-index*, it remains in the index, because the index is well-linked under concurrent operations by peers.* □

Algorithm 4: Protocol for Splitting the Root of an *H-index*

proc $splitRoot(r, V_{r1}, V_{r2})$
  /* r - root id; V_{r1}, V_{r2} - partitions of root contents*/
1: $(c_1, c_2, h_{c1}, h_{c2}, h_{r_{new}}) \leftarrow f(h_r)$ /* new node ids and headers */
2: let l_1 and l_2 denote the LCMs of signatures in V_{r1} and V_{r2} respectively
3: $status \leftarrow insert_S(c_1, \emptyset, h_{c1}, V_{r1})$ /*try to create left child*/
4: **if** $status = first$ **then**
5:   $insert_S(c_2, \emptyset, h_{c2}, V_{r2})$ /* create right child */
6:   $insert_S(r, h_r, h_r, V_{r1} \cup V_{r2})$ /* remove root contents */
7:   $insert_U(r, (l_1, c_1), NONE)$ /* link to left child */
8:   $insert_U(r, (l_2, c_2), NONE)$ /* link to right child */
9:   $insert_S(r, h_r, h_{r_{new}}, \emptyset)$ /* update root's header */
  else
10:   $await(r's\ header\ != \ h_r)$ /* wait for a hdr change using $lookup_O$ */
  endif
endproc

5.3.4 Special Cases

An *H-index* always remains *well-linked*, but the divisibility property may not hold in the nodes that were involved in the splitting process. Suppose a peer splits a node and updates the parent's signature entry with the new LCM of the signatures in the split node. It is possible for another peer to have updated the parent's signature entry while inserting a document signature d, after this peer read the parent. Thus, the update could be lost and d may not be found by traversing the index from the root, although it is present in some leaf node of the index (Remark 3). Similarly, after a peer read a full node that it decided to split, another peer could have updated a signature in the node while inserting a signature. Such an update may be lost after the node splits. If a peer cannot find a document signature it successfully inserted by searching from the root, it can delete that signature from the leaf node it inserted into, and perform a reinsert from the root. Note that this check is not necessary every time a peer inserts a document signature. The likelihood of missing a signature will be high when node splits occur frequently. This could happen due to small node fanouts and a high rate of publishing.

In another scenario, suppose a peer inserts a signature into a leaf, and suppose its parent splits (due to other peers) before it updates the entry chosen for insertion using $insert_U$. Thus, the chosen entry in the parent could have been moved to another node after splitting. In this case, $insert_U$ (Line 5 in Algorithm 2) on the parent would result in two logical links to the leaf. To avoid this, a calling peer issues $insert_U$ using the $STRICT$ option. Thus when $insert_U$ is invoked on a non-leaf node with an argument (s, p), a peer, that is responsible for storing that node, atomically updates the signature only if some pair (r, p) already exists in the node. Otherwise, the node is unaffected.

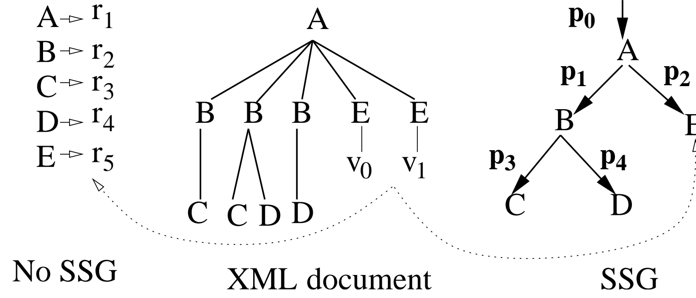


Figure 5: Summarizing values in an XML document with and without SSGs

Thus, in the current design of *H-index*, after node splits, an entry's signature may not be equal to the LCM of its child node signatures. This discrepancy is resolved when the child node splits, and its parent entry's signature is updated with the LCM. In our experiments, for a fanout of 500 and with 27% of peers simultaneously publishing, we observed that no signatures were missed for the tested queries in Section 6.

5.3.5 Locating Documents using an *H-index*

Algorithm 5: Locating XML Documents using an *H-index*

proc $location(k, sig_q)$
  /* k - index node id; sig_q - query signature; */
1: $(nodetype, list) \leftarrow lookup_O(k, sig_q, DIVIDE)$
2: **if** $nodetype = LEAF$ **then**
3:   Output docids in $list$
  else
4:   **foreach** *node id k in list* **do**
5:     $location(k, sig_q)$ /* Traverse each child */
    endfch
  endif
endproc

Given a query signature s_q, a peer traverses the corresponding *H-index* for the target node in the query, starting from its root node, using $lookup_O$ with the $DIVIDE$ option. The (sig, ptr) entries in an index node are tested for divisibility with s_q by the peer storing it. The ptr entries are returned and tested for further traversal. If a leaf level node is reached, then the matching docids are output. Otherwise, each candidate child node is traversed further. If a query has '//' or '*', then the process of locating documents is the same as above, except that the query signature is a list of signatures, and Theorem 2 is used for the divisibility test.

5.4 Indexing Values in XML Documents

Values in XML documents are summarized using histograms for numeric data, and polynomial signatures for textual data. (A text string is hashed to an irreducible polynomial.) For instance, the values v_0 and v_1 in Figure 5, assuming that they are of the same type, are summarized together (say as V), since their elements `E` map to the same node in the SSG (dotted arrow on the right). If v_0 and v_1 are numeric, then V denotes a set of intervals of histogram buckets that are not empty. If v_0 and v_1 are textual data, then V denotes the LCM of the hash values of v_0 and v_1.

The document signature and value summary (p_2, V) are considered for indexing. If the SSG is not available, then the irreducible polynomial assigned to tag E (during signature construction) is chosen, and the pair (r_5, V) is considered for indexing. (Although, the same can be done when SSGs are available, using

the edge polynomials can differentiate the case when E can have more than one parent in the SSG.) Since tags are already used to identify the *H-index*es, we use irreducible polynomials for value indexes.

Textual and numeric value summaries are indexed using hierarchical distributed indexes called *tH-index* and *nH-index*, respectively. A separate *tH-index* or *nH-index* is created based on the irreducible polynomial associated with the value summary. For example, the value summary (p_2, V) for the XML document in Figure 5 is indexed by a *nH-index* or a *tH-index* maintained for polynomial p_2. The *H-index* and the *tH-index* are almost similar, except that the index keys (*i.e.*, signatures) are constructed differently. The *nH-index* is also similar to the *H-index* except that it uses one-dimensional intervals as index keys, and its containment property is like that of an R-tree [18]. These value indexes store both the docid and document signature for each (p, V) that is indexed in its leaf nodes.

Given a query with a value predicate, this predicate is first examined. For example, to process a query `/A[E="XML"]`, if the SSG is available, $(p_2, hash(\text{“}XML\text{”}))$ is constructed. Otherwise, $(r_5, hash(\text{“}XML\text{”}))$ is constructed. The signature of `/A/E` is computed as before. (For numeric value predicates, the operator is also considered.) To process the above query, the *tH-index* for p_2 (or r_5) is searched. The signatures in the node entries are tested for divisibility with $hash(\text{“}XML\text{”})$. When a leaf entry that is divisible is found, the associated document signature is tested for divisibility with the signature for `/A/E`. For a query with a numeric value predicate, a *nH-index* is searched. The query interval is tested with the intervals in the node entries according to the operator in the value predicate.

Currently, *psi*X searches a value index if a value predicate is present in an XPath query. Otherwise, an *H-index* corresponding to the *target node* in the query is searched. Picking the best index is a future research challenge. We have evaluated queries with value predicates using value indexes and report the results in Section 6.5.4.

5.5 Failure, Joining and Leaving of Peers

The *psi*X system relies on the DHT for managing the failure of peers. For instance, Chord replicates a key (and its value) in some r immediate successors of the peer responsible for storing the key. If this peer fails, then a replica is fetched from one of its r successors. The effectiveness of replication has been well studied and evaluated [32, 10]. Joining and leaving of peers is handled using the Chord protocol. A peer could request to delete the documents, that it has published, before leaving the network.

6 Experimental Evaluation

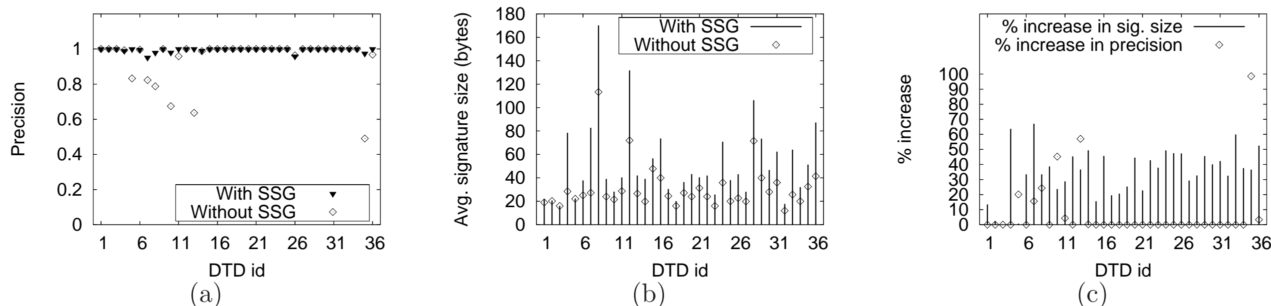


Figure 6: Evaluation of (a) signature precision with and without SSGs, (b) signature size with and without SSGs, and (c) the tradeoff between increase in signature size and precision improvement

The goal of our experimental evaluation is to test the effectiveness and performance of (a) our signature scheme, and (b) our proposed indexing strategy based on *H-index*. (Evaluation of the underlying DHT is not the focus of this work.) We built an inverted index based on XML element tags called *I-index*, which also stores signatures. *I-index* is similar to the inverted index scheme proposed by Galanis *et al.* [14], except

that document signatures are stored as summaries instead of document path summaries. Thus, twig queries can be processed without breaking them into root-to-leaf paths. This is an improvement over path-based summarization techniques. (The optimizations such as split-replicate and split-toss were not implemented.) We focus on a typical P2P scenario where queries are issued more frequently than document publications.

6.1 Implementation Details

We implemented both *I-index* and *H-index* in C++, and ran the experiments reported in Sections 6.2, 6.3, and 6.6 on a Linux machine with 2.66GHz Intel Core 2 Duo processor, 1GB RAM, and one 250GB SATA drive.

For the experiments reported in Section 6.4, we used PlanetLab [26], where peers can run on machines that are geographically distributed. Therefore, the network conditions are similar to that of the Internet. Each PlanetLab machine runs on Linux and can be simultaneously used by several users. Through virtualization technology, each user can run programs inside a separate virtual machine called *slice*. The load on the machines and network conditions vary with time, and hence, controlled experiments cannot be performed.

6.2 Evaluation of Signature Precision and Size

6.2.1 Datasets and Queries

We obtained 36 Document Type Definitions (DTD) (*e.g.*, Treebank, DBLP, Sigmod Record) published on the Internet [34, 35, 37]. For each DTD, a set of documents were generated using IBM's synthetic XML data generator. In all, our test data consisted of a rich set of heterogeneous XML documents with different characteristics such as depth, fanout, recursive structure, etc. The structural summary graphs were generated using these DTDs, and the edges were assigned irreducible polynomials of degree 17. A total of 6338 distinct irreducible polynomials were used.

We generated XPath queries for the collected DTDs using the XPath generator provided by the YFilter system [38]. The XPath queries contained nested path expressions, predicates, wildcard '*' , and '//' axis. We did not evaluate a value predicate in a twig, but we considered the attributes in the predicate during structure matching.

For each DTD, an average of 798 documents were generated. The total size of the document collection per DTD was an average of 7.3MB. An average of 176 queries were generated for each DTD. Signatures were generated for the documents and queries, with and without the knowledge of SSGs, as described in Sections 5.1 and 5.2. For each DTD, we computed the number of falsely matched documents (in the document collection) whose signatures were divisible by a query signature (in the query collection), but did not actually contain the query pattern.

6.2.2 Results

The precision values (averaged for each DTD), with and without the knowledge of SSGs, are shown in Figure 6(a). When SSGs were used for signature construction, for 29 out of the DTDs, the precision was 1.0, which indicated that our signature scheme precisely captured the structural summary of the documents. The lowest precision value among the DTDs was 0.95. When SSGs were not used, signatures did not capture the structural relationships between tags, and thus, the precision drastically reduced for 16% of the DTDs (*e.g.*, 0.49 for Treebank). In Treebank, the same element name can appear at different levels in the document, and so, ignoring the SSG was detrimental. In another case, precision for DBLP dropped to 0.675. A query `/dblp//mastersthesis[address]` matched a DBLP document that contained paths `/dblp/article/address` and `/dblp/mastersthesis`. For many of the DTDs, the precision was 1.0, even when SSGs were not used, because the queries were constructed according to the DTD.

For each DTD, the average document signature size, constructed with and without SSGs, is shown in Figure 6(b). Overall, the use of SSGs increased the size of the document signatures, but precisely captured structural relationships. The tradeoff between the percent increase in the average document signature size and percent increase in precision using SSGs, is shown in Figure 6(c). In some cases, the increase in signature size was, in fact, offset by significant improvement in precision. Thus, if a publisher has some knowledge of

the queries that can be issued by users, an analysis such as the above, can help in deciding whether to use (and advertise) an SSG or not.

6.3 Signature Construction Cost

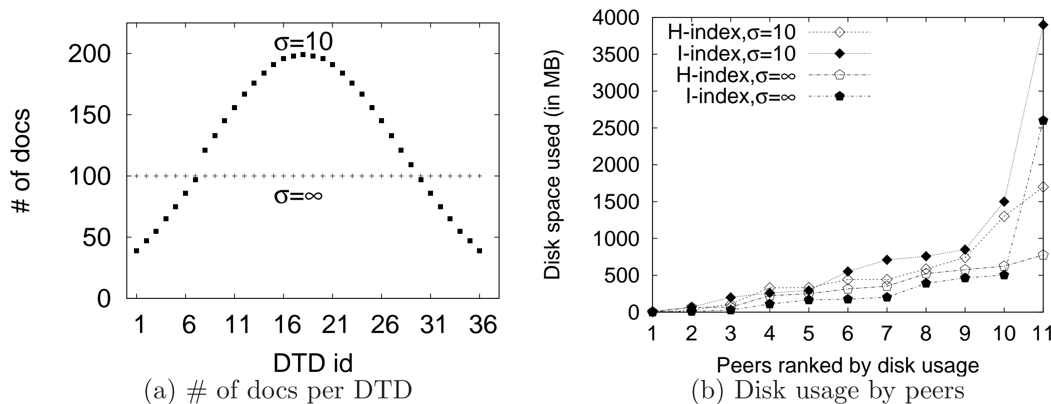
We measured the average cost of constructing polynomial signatures using Algorithm 1. For document signatures, the average cost was less than 2 ms. For query signatures, the average cost was less than 0.4 ms. Overall, this cost is small compared to the average network delays that can be more than 50ms.

6.4 Setup for Evaluation of *I-index* and *H-index*

For all the subsequent experiments, we used signatures constructed with the knowledge of SSGs. Irreducible polynomials of degree 22 were chosen to yield longer signatures.

Table 2: Insert Workload

Name	Avg. Doc. Size(bytes)	Max. Doc. Size(bytes)	Avg. Sig. Size(bytes)	Max. Sig. Size(bytes)
IW($\sigma = 10$)	920.58	14,726	46.17	464
IW($\sigma = \infty$)	924.87	14,726	46.97	464



(a) # of docs per DTD

(b) Disk usage by peers

Figure 7: (a) Insert workloads IW, and (b) disk usage on PlanetLab.

6.4.1 Datasets and Document Signatures

We experimented with two insert workloads (denoted as IW) by choosing a different number of documents for each DTD, based on a normal distribution. The DTDs were sorted lexicographically by their file names. (Thus `actors.dtd` was the first DTD, and `viml.dtd` was the last.) We chose a different standard deviation (σ) for each normal distribution, and denote them by $\sigma = 10$ and $\sigma = \infty$ (*i.e.*, uniform distribution). Table 2 shows the average document size, maximum document size, average signature size, and maximum signature size for the two IW workloads. Figure 7(a) shows the number of documents for each DTD in IW for the two σ values. The total number of documents in IW for $\sigma = 10$ and $\sigma = \infty$ was 4,565 and 3,600, respectively.

6.4.2 Queries

We tested seven different XPath queries that contained '*', '//' axis, and value predicates. (See Table 3.) Queries $Q_1 - Q_3$ demonstrate the case where the performance of *I-index* suffers due to long lists of document

Table 3: XPath queries

ID	XPath query	Indexes/tags used for documents	# of docs matched		# of signatures	total size (in bytes)
			$\sigma = 10$	$\sigma = \infty$		
Q_1	/*//member[email]//name	*H-index*,*I-index* for name	258	300	1	12
Q_2	//competency/level[evidence]/description	*H-index*,*I-index* for description	219	219	1	12
Q_3	//Actor[Name]/Filmography/Movie/Year	*H-index*,*I-index* for Year	87	240	1	16
Q_4	/FILE//ADVP[TO]/ADJ	*H-index*,*I-index* for ADJ	0	0	17	204
Q_5	//Quote[Last]//PE_Ratio	*H-index*,*I-index* for PE_Ratio	327	300	1	12
Q_6	//Date[Year="level 3"]	*tH-index* for Year	552	300	1	8
Q_7	/dblp/article[volume<1]	*nH-index* for volume	0	0	1	12

signatures. Queries $Q_4 - Q_5$ demonstrate the case where *H-index* and *I-index* process an identical set of signatures. (The selected *H-index* had just one index node.) Queries $Q_6 - Q_7$ contain value predicates.

6.4.3 P2P Setup

We chose eleven PlanetLab machines that were geographically distributed across the United States. A P2P network was setup by starting one Chord peer on each machine. (We could have started multiple independent Chord peers on each machine to setup a P2P network with a much larger number of peers. This would, however, increase the load on each machine. In addition, this would increase the number of hops per DHT insert/lookup operation, because the number of hops depends on the logarithm of the total number of peers.) The publishing step, using *I-index* and *H-index*, was performed at the same time on PlanetLab on two different slices. This was done to ensure that the slices witnessed the same load and network conditions, but did not directly interfere with each other. The case was similar when queries were executed.

Performance evaluation was carried out by first publishing all the document signatures for the documents in `IW`. Three peers published the same `IW` simultaneously. This tripled the total number of documents published for each `IW`, and provided more opportunities to observe index node splits in *H-index*. (The index fanout was fixed at 500.)

Once all the document signatures were published, each query was executed 100 times from a single peer. A query was considered complete, once the global identifiers of the relevant documents were retrieved.

6.5 Performance Evaluation on PlanetLab

Overall, we observed that *H-index* could significantly outperform *I-index* for processing XPath queries, to locate relevant XML documents. While publishing documents, *I-index* was twice faster and consumed less bandwidth than *H-index*. However, *I-index* yielded greater skewness in disk space usage among peers. As a result, a few peers spent considerably more resources than others. We conclude that *H-index* is a better choice in a typical P2P environment, where queries are issued more frequently than documents are published.

6.5.1 Disk Usage

Figure 7(b) shows eleven peers ranked by disk usage after publishing the insert workloads. The reason *I-index* yielded a greater skewness of disk space usage among participating peers than *H-index* is because some lists in *I-index* grew very large due to frequently occurring XML tags, even across different DTDs. Peers storing

such lists spent more disk space than others. *H-index* avoided this problem by splitting index nodes. For example, when $\sigma = \infty$, the maximum disk space consumed by a peer for *I-index* was 3.5 times more than when *H-index* was used.

When four peers published the `IW` workload simultaneously, *I-index* failed to complete successfully, because one peer ran out of disk space and crashed. This is a significant limitation of *I-index*.

6.5.2 Publishing Performance

Table 4: Publishing cost

Dataset	*I-index*	*H-index*			
	Total time (secs)	Total time (secs)	Total # splits	Avg. node split time (secs)	Avg. blocking time (secs)
$\sigma = 10$	23,208	49,157	155	1.76	1.45
$\sigma = \infty$	16,633	34,199	62	1.47	1.36

Table 5: Bandwidth consumed during publishing

Dataset	# of inserts into the indexes	*I-index*	*H-index*	
		Data write	Data read	Data write
$\sigma = 10$	146,265	13.5MB	20MB	23MB
$\sigma = \infty$	112,671	11MB	12.7MB	14MB

Table 4 shows the total publishing time for the two insert workloads. *H-index* required about twice the time of *I-index*, because, in *H-index*, both DHT *insert* and *lookup* operations were issued and node splits were performed. (*I-index* issued only DHT *insert* operations.) Note that the workloads $\sigma = 10$ and $\sigma = \infty$ were published at different times on PlanetLab, but the construction of *H-index*es and *I-index*es, for each σ, was started at the same time on two different PlanetLab slices.

The workload $\sigma = 10$ contained more documents than $\sigma = \infty$ and thus, took more time to publish. The number of times the insertion protocol was executed for $\sigma = 10$ and $\sigma = \infty$ was 146,265 and 112,671, respectively. The number of index node splits, the average time to split a node (lines 3-7 in Algorithm 3 and lines 3-9 in Algorithm 4), and the average time to block during splitting (line 8 in Algorithm 3 and line 10 in Algorithm 4) are also reported in Table 4. The bandwidth consumed during publishing is reported in Table 5. *I-index* consumed significantly less bandwidth, as it did not require DHT lookups and node splits.

In Chord, when a *lookup* is issued, the caller is returned the peer that stores the requested key. A subsequent insert operation on the same key can specify this peer as a starting point for routing. This reduces the number of hops required for a DHT *insert*. *H-index* used this optimization when a document signature was inserted into an *H-index* as it performed a *lookup* before updating the index nodes. (This optimization was not applied to an *I-index* as it directly issued an *insert*.)

Further, the process of picking the best child, when inserting into an *H-index*, was performed locally at the peer that stored an index node. When an index node was full, a publishing peer fetched the full node to perform the splitting protocol. This fetch could be avoided if the task of splitting a node is assigned to a peer that stores it [36]. This would reduce the bandwidth consumption, but would force the storing peer to remain in the P2P network until the splitting process has been completed successfully.

6.5.3 Locating XML Documents

Figure 8 shows the average time taken by *H-index* and *I-index* to locate relevant XML documents for queries $Q_1 - Q_5$. Table 6 shows the number of hops required per query. The XML element tag chosen to select an

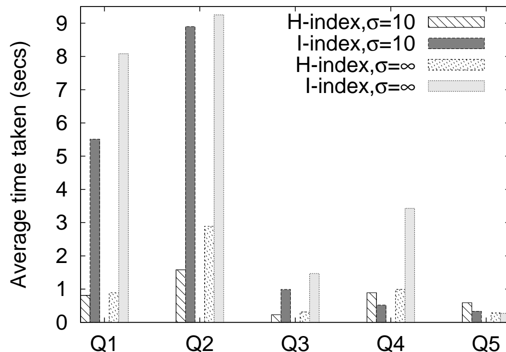


Figure 8: Average time taken for locating documents (over 100 runs)

Table 6: Total # of hops required (1 DHT lookup = 2 hops)

Type	Q_1	Q_2	Q_3	Q_4	Q_5
H-index, $\sigma = 10$	4	6	4	2	2
I-index, $\sigma = 10$	2	2	2	2	2
H-index, $\sigma = \infty$	4	6	4	2	2
I-index, $\sigma = \infty$	2	2	2	2	2

H-index and an *I-index* for processing each query is shown in Table 3. The same tag is used for a query to demonstrate that a hierarchical organization is a better choice for document signatures than a single long list. For $Q_1 - Q_3$, *I-index* processed long lists of signatures, among which only a fraction were part of the final result. (The list length processed by *I-index*, and the number of matched signatures are shown in Table 7.) For queries $Q_4 - Q_5$, *H-index* and *I-index* processed the same set of document signatures. (A selected *H-index* had only one index node and was essentially identical to its corresponding *I-index*.) Both *H-index* and *I-index* returned the same set of documents signatures, and thus, no document signatures were missed due to node splits in *H-index*. The recall was one for queries $Q_1 - Q_5$.

For $Q_1 - Q_3$, we observed that *H-index* was significantly faster than *I-index*. For example, *H-index* processed Q_1 about 7 times and 9 times faster than *I-index* for $\sigma = 10$ and $\sigma = \infty$, respectively. [3] The *H-index*es for the chosen XML tags had undergone node splits during publishing. Although, *H-index* required more hops for index traversal than *I-index* for queries $Q_1 - Q_3$, the processing of long lists slowed *I-index* substantially. For example, *I-index* required an average of 5.18 secs to process the list for Q_1 ($\sigma = 10$).

Table 7: *I-index*: length of list processed and # of matched signatures

# of sigs	Q_1		Q_2		Q_3	
	$\sigma = 10$	$\sigma = \infty$	$\sigma = 10$	$\sigma = \infty$	$\sigma = 10$	$\sigma = \infty$
total	1633	1477	1204	1171	1228	841
match	258	300	219	219	87	240

The bandwidth consumed by *H-index* was slightly higher than *I-index* due to accessing the root node of the index. For instance, *H-index* read 6,806 bytes as compared to 6,694 bytes by *I-index* for Q_1 ($\sigma = 10$). Both *H-index* and *I-index* tested for divisibility of signatures with a query signature locally at a peer storing an index node, rather than fetching the node over to a querying peer.

For queries $Q_4 - Q_5$, the structure of the chosen *H-index* and *I-index* were identical, as the *H-index* did

[3] The PlanetLab machines that we used had good connectivity and were heavily loaded.

not undergo any index node splits. The number of hops required were identical. Their query processing times are comparable, except for Q_4 ($\sigma = \infty$) where *I-index* required more time. This is attributed to the higher load on the peer that locally tested the signature list for *I-index*. This peer spent an average of 3.29 secs for processing this list.

6.5.4 Value Indexes

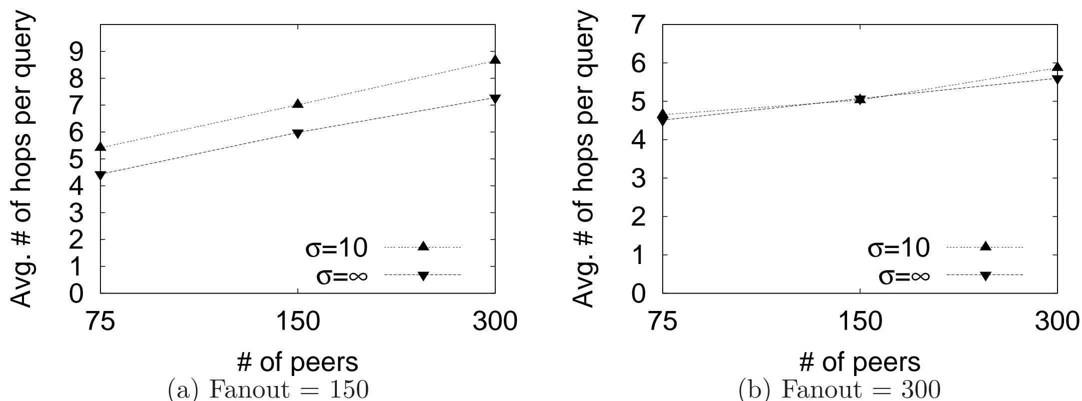
Table 8: Performance of value indexes (100 runs)

ID	Avg. time taken		# of hops	
	$\sigma = 10$	$\sigma = \infty$	$\sigma = 10$	$\sigma = \infty$
Q_6	0.69 secs	0.39 secs	6	2
Q_7	0.12 secs	0.118 secs	2	2

Table 8 shows the performance to process queries $Q_6 - Q_7$ using the value indexes built in *psi*X. The time taken for locating documents (averaged over 100 runs), and the required number of hops are reported. For Q_6, the *tH-index* for the irreducible polynomial assigned to the incoming edge of `Year` was used. For Q_7, the *nH-index* for the irreducible polynomial assigned to the incoming edge of `Volume` was used. Selecting the best among an *H-index*, *tH-index*, and an *nH-index* for a given XPath query is a future research challenge.

6.6 Scalability of *H-index*

To evaluate the scalability of *H-index* for locating documents, we setup a P2P network on one machine and ran up to 300 Chord peers. Only one instance of each insert workload was published. Also, a single *H-index* was built over all the document signatures. We built the *H-index* with a fanout of 150 and 300, respectively. We created a query workload (containing 3,314 queries) by randomly selecting queries from the query sets described in Section 6.2.1. We measured the average number of hops required per query. We observed that with an increase in the number of peers, the number of hops required increased slowly. This showed that *psi*X inherited the scalability properties of the underlying DHT.



(a) Fanout = 150 (b) Fanout = 300

Figure 9: Scalability of *H-index* by varying the # of peers

7 Conclusions and Future Work

We have presented a new system called *psi*X, built atop a DHT framework, for efficiently locating XML data in a peer-to-peer environment. Each XML document is mapped into an algebraic signature that captures

the structural summary of the document. A twig query is also mapped into a signature. By dividing a document signature with a query signature, the existence of a twig pattern can be tested quickly and holistically. This avoids breaking a twig pattern into simpler patterns and issuing separate lookups, which can be expensive given that the network delays are a dominating cost. The participating peers in the network collectively maintain a collection of distributed hierarchical indexes over the signatures. Document signatures of structurally similar documents can be effectively grouped together in each index. Our extensive experimental evaluation on PlanetLab shows that *psi*X provides an efficient location service for a variety of XML documents.

In future, we would like to investigate how caching can help improve the performance of *psi*X. We plan to evaluate the performance of *psi*X under different levels of *churn*. We have, so far, assumed that peers are neither malicious nor selfish. We plan to investigate schemes to protect *psi*X in the presence of malicious and selfish peers. Further, privacy issues can arise in *psi*X as peers are encouraged to share their DTDs. These issues will be investigated in future.

References

[1] K. Aberer, A. Datta, M. Hauswirth, and R. Schmidt. Indexing Data-oriented Overlay Networks. In *Proc. of the 31st VLDB Conference*, pages 685–696, Trondheim, Norway, 2005.

[2] S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun. XML Processing in DHT Networks. In *Proc. of the 24th IEEE Intl. Conference on Data Engineering*, Cancun, Mexico, Apr. 2008.

[3] S. Antony, D. Agrawal, and A. E. Abbadi. P2P Systems with Transactional Semantics. In *Proc. of the 11th Intl. Conference on Extending Database Technology*, Nantes, France, 2008.

[4] E. Bach and J. Shallit. *Algorithmic Number Theory (Volume 1: Efficient Algorithms)*. MIT Press, 1996.

[5] A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath Lookup Queries in P2P Networks. In *the 6th annual ACM Intl. Workshop on Web Information and Data Management (WIDM'04)*, pages 48–55, Washington, DC, Nov. 2004.

[6] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proc. of the 2002 ACM-SIGMOD Conference*, Wisconsin Madison, WI, 2002.

[7] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[8] J. H. Conway and R. K. Guy. *The Book of Numbers*. Springer-Verlag, 1996.

[9] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. P-Ring: An Efficient and Robust P2P Range Index Structure. In *Proc. of the 2007 ACM-SIGMOD Conference*, pages 223–234, Beijing, China, 2007.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area Cooperative Storage with CFS. In *Proc. of the 20th ACM Symposium on Operating Systems Principles*, pages 202–215, Banff, Oct. 2001.

[11] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[12] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). *IETF Request for Comments 3174*, 2001.

[13] F. Ruskey and K. Cattel. The Combinatorial Object Server. Available from http://www.theory.csc.uvic.ca/.

[14] L. Galanis, Y. Wang, S. R. Jeffery, and D. J. DeWitt. Locating Data Sources in Large Distributed Systems. In *Proc. of the 29th VLDB Conference*, Berlin, 2003.

[15] P. Ganesan, B. Yang, and H. Garcia-Molina. One Torus to Rule them All: Multi-dimensional Queries in P2P Systems. In *Seventh Intl. Workshop on the Web and Databases*, Paris, France, June 2004.

[16] L. Garces-Erice, P. A. Felber, E. W. Biersack, G. Urvoy-Keller, and K. W. Ross. Data Indexing in Peer-to-peer DHT Networks. In *Proc. of the 24th IEEE Intl. Conference on Distributed Computing Systems*, pages 200–208, Tokyo, Mar. 2004.

[17] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. of the 23rd VLDB Conference*, pages 436–445, Athens, Greece, Aug. 1997.

[18] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proc. of the 1984 ACM-SIGMOD Conference*, pages 47–57, Boston, MA, June 1984.

[19] H. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: A Balanced Tree Structure for Peer-to-Peer Networks. In *Proc. of the 31st VLDB Conference*, Trondheim, Norway, 2005.

[20] G. Koloniari and E. Pitoura. Content-Based Routing of Path Queries in Peer-to-Peer Systems. In *Proc. of the 9th Intl. Conference on Extending Database Technology*, pages 29–47, Crete, Greece, 2004.

[21] G. Koloniari and E. Pitoura. Peer-to-Peer Management of XML Data: Issues and Research Challenges. *SIGMOD Record*, 34(2):6–17, June 2005.

[22] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proc. of the 27th VLDB Conference*, pages 361–370, Rome, Italy, Sept. 2001.

[23] B. Liu, W.-C. Lee, and D. L. Lee. Supporting Complex Multi-Dimensional Queries in P2P Systems. In *Proc. of the 25th IEEE Intl. Conference on Distributed Computing Systems*, pages 155–164, Columbus, OH, June 2005.

[24] T. Milo and D. Suciu. Index structures for path expressions. In *Proc. of the 7th Intl. Conference on Database Theory*, pages 277–295, Jerusalem, Israel, Jan. 1999.

[25] R. Overbeek, T. Disz, and R. Stevens. The SEED: A Peer-to-Peer Environment for Genome Annotation. *Communications of the ACM*, 47(11):47–50, Nov. 2004.

[26] PlanetLab. http://www.planet-lab.org.

[27] M. O. Rabin. Fingerprinting by Random Polynomials. Technical Report TR 15-81, Harvard University, Cambridge, MA 02138, 1981.

[28] P. Rao and B. Moon. PRIX: Indexing And Querying XML Using Prüfer Sequences. In *Proc. of the 20th IEEE Intl. Conference on Data Engineering*, Boston, MA, March 2004.

[29] P. Rao and B. Moon. Sequencing XML Data and Query Twigs for Fast Pattern Matching. *ACM Transactions on Database Systems*, 31(1):299–345, Mar. 2006.

[30] C. Sartiani, P. Manghi, G. Ghelli, and G. Conforti. XPeer: A Self-Organizing XML P2P Database System. In *Intl. Workshop on Peer-to-Peer Computing and Databases*, Greece, 2004.

[31] G. Skobeltsyn, M. Hauswirth, and K. Aberer. Efficient Processing of XPath Queries with Structured Overlay Networks. In *The 4th Intl. Conference on Ontologies, DataBases, and Applications of Semantics*, Aiga Napa, Cyprus, Oct. 2005.

[32] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of the 2001 ACM-SIGCOMM Conference*, pages 149–160, San Diego, CA, Aug. 2001.

[33] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks. In *Proc. of the 2003 ACM-SIGCOMM Conference*, pages 175–186, Germany, Aug. 2003.

[34] The Niagara Project. http://www.cs.wisc.edu/niagara/.

[35] UW XML Repository. www.cs.washington.edu/research/xmldatasets.

[36] S. Viglas. Distributed File Structures in a Peer-to-Peer Environment. In *Proc. of the 23th IEEE Intl. Conference on Data Engineering*, pages 406–415, Cancun, Mexico, 2007.

[37] XML.org. Available from http://www.xml.org/xml.

[38] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang and Peter Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.

DTD Names
actors.dtd
apm_cim.dtd
assetselect.dtd
bib.dtd
chordml.dtd
club.dtd
competency10.dtd
content10.dtd
customer.dtd
dblp.dtd
department.dtd
ebay.dtd
Feedback-1_1.dtd
IML1.0.dtd
Invoice.dtd
lineitem.dtd
movies.dtd
nation.dtd
obligor.dtd
orders.dtd
originator.dtd
parts.dtd
partsupp.dtd
payment.dtd
paymentschedule.dtd
personal.dtd
PersonName-1_1.dtd
pets.dtd
profile.dtd
quote.dtd
reed.dtd
region.dtd
SigmodRecord.dtd
supplier.dtd
treebank.dtd
viml-2-0.dtd
viml.dtd

Table 9: DTDs

APPENDIX

A DTDs used for Experimental Evaluation

The DTDs used for our experiments are listed in Table 9.