XML processing in DHT networks

Serge Abiteboul #1, Ioana Manolescu #1, Neoklis Polyzotis *2, Nicoleta Preda #1, Chong Sun *3

INRIA Futurs & University of Paris XI
Gemo Team, 4 rue Jacques Monod, Orsay Cedex, 91893, France
1 firstname.lastname@inria.fr

* *Computer Science Departament, University of California, Santa Cruz*
1156 High St, Santa Cruz, CA 95064, United States
2 alkis@cs.ucsc.edu
3 sunchong@soe.ucsc.edu

***Abstract*— We study the scalable management of XML data in P2P networks based on distributed hash tables (DHTs). We identify performance limitations in this context, and propose an array of techniques to lift them. First, we adapt the DHT platform's index store and communication primitives to the needs of massive data processing. Second, we introduce a distributed hierarchical index and associated efficient algorithms to speed up query processing. Third, we present an innovative, XML-specific flavor of Bloom filters, to reduce data transfers entailed by query processing. Our approach is fully implemented in the KadoP system, used in a real-life software manufacturing application. Our experiments demonstrate the benefits of the proposed techniques.**

I. Introduction

The current development of peer-to-peer (P2P) information sharing has opened the way for supporting rich data management applications in a distributed environment. Of particular interest is the support of structured queries over distributed XML data, as XML is well suited to represent the variety of data that may be shared within a P2P system. In this direction, a recent work has proposed the KadoP platform [1] that relies on the well known technology of Distributed Hash Tables (DHT) in order to support complex queries over the shared XML data. In KadoP, the peers publish XML documents and share the tasks of indexing the data and processing queries. KadoP indexes the XML data in the form of postings, where each posting encodes information on an element or a keyword. The use of a DHT allows KadoP to implement a "responsibility" mechanism, by which (typically) a single peer stores all the postings for a given term, and this peer can be efficiently identified by all other peers in the system. Given a query, the system combines the postings stored in the index to locate the peers that can contribute to the query, and forwards the query to these peers where the final results are computed.

The potential problem faced by this efficient approach is that posting lists for very popular terms grow very large and limit the system's scalability. More specifically, index construction (equivalently, document publishing) becomes slower, and, more critically, the transfer of large sets of postings may be damaging in terms of query response time and data transfer load. This paper presents techniques that we have developed to address these issues. More concretely, the technical contributions of our work can be summarized as follows:

– To improve query response time, we introduce a novel optimization technique that speeds-up the exchange of large sets of postings, a main cause of poor query performance. We employ a horizontal partitioning scheme, in which a large set of postings is distributed among peers based on range conditions. This scheme enables a highly parallel twig join algorithm that can reduce significantly the total processing time. It also allows the index to filter partitions irrelevant to the query, thereby saving on data transfers.

– To limit data transfers, we introduce Structural Bloom Filters for distributed structural XML joins. A Structural Bloom Filter provides a compact representation of a set of postings that is suitable for filtering the postings of another list. In brief, given a Structural Bloom Filter on the postings of some term a, it is possible to check (with a configurable error probability) whether another posting has a descendant or an ancestor element in the set of postings of a. We detail the mechanism behind Structural Bloom Filters, and propose techniques that integrate these filters in the evaluation of index queries.

The proposed techniques are fairly general, as they can be applied to any XML system that follows the same general architecture as KadoP. Moreover, they are largely orthogonal and can thus be easily composed.

We have implemented the proposed techniques in KadoP and present experimental results, on synthetic and real-life data sets, demonstrating that the system scales gracefully to large data volumes. Our study is among the first performed on an actual XML data management platform over a P2P network, as opposed to simulations. (Such studies for relational data sharing appear in [2], [3]). Implementing the actual system allowed detecting a set of real performance bottlenecks (see Section III), which simulations alone do not reveal. We view our development and the solutions we brought to these problems as one of our contributions, that can benefit to any DHT-based data management tool.

The modified system is currently being tested in cooperation with the Mandriva company (originally known as Mandrake Software) with a real application entitled Edos. In Edos [4], the peers are Mandriva Linux developers, so potentially a population in the hundreds of peers, and the indexed data comprises the XML descriptions of the software packages in the Mandriva Linux distribution. (A distribution is composed of about 10,000 software packages, which translates to more than 100 megabytes of XML metadata.) One sample query supported by the system is "find the packages that provide a 'terminal-emulator' and that are compatible with version '4.2' of the 'readline' library". Of course, the system has to support simultaneously many (time-based) versions of the distribution.

The paper is organized as follows. We review the existing KadoP system [5] in Section II, and in Section III we describe important techniques that address some of its scalability limitations. Section IV introduces a technique based on partitioning and distributing index blocks, that greatly reduces query response time. Bloom-based optimization techniques reducing network traffic are described in Section V. We review related works in Section VI, and conclude the paper in Section VII.

II. The KadoP system

Our work considers P2P applications with a possibly very large number (millions) of XML documents stored in a large number (up to the thousands) of peers. We assume that peer volatility is moderate, i.e. a peer's typical online span is of the order of a few hours (as opposed to a few minutes). We note that this model encompasses a wide range of real-life applications, such as scientific data sharing platforms, shared bookmarks, or virtual libraries (e.g., the French federated online scientific library *HAL* [1]).

Data and Query Model. Each document in the system is identified by a pair (p, d), where p is the numerical identifier of the peer that checked it in, and d the document identifier within this peer. A document (p, d) is a labeled unranked tree, comprising *element* and *text* nodes. (For simplicity, we do not distinguish between elements and attributes.) Each element is labeled with a string symbol from some alphabet $Label$ and is uniquely identified by a *structural identifier* (*sid* for short) $sid = (start, end, lev)$. Here, $start$ (resp. end) is the number assigned to the opening (resp. closing) tag of the element, when reading the document and numbering its tags in the order they appear in the document. The third value lev denotes the element's level in the tree. Structural ids allow deciding if element e_1 is an ancestor of element e_2 by verifying if $e_1.start<e_2.start<e_1.end$. As we discuss later, this property forms the basis of the techniques that we present in Section V.

In our work, we focus on the class of tree-pattern queries over single documents. A tree pattern query q intuitively corresponds to the FOR/WHERE clause of a FLWR expression and it can be modeled as a tree where: each node is labeled with a symbol from $Label$; and each edge is labeled with "/" or "//" to denote the child or descendant axis respectively. Moreover, a query node can specify whether the label is to be matched as an element label, or as a word occurring in a text node. We say that q matches a document (p, d) if there exists a mapping from the nodes in q to nodes in (p, d) that (i) preserves the parent/child and ancestor/descendent relationships, and (ii) satisfies the query predicates on labels and values. We refer to such a mapping as a *binding tuple* for q.

Consider a FLWR query Q that is built around a corresponding tree pattern query q. Essentially, Q employs q as its FOR clause, but it may add additional (non-textual) predicates in the WHERE clause, or the computation of complex functions in the RETURN clause. The queries Q and q thus satisfy the following important property: if Q has a non-empty result on a document (p, d), then q matches (p, d). As we describe shortly, this property forms the crux of the query evaluation mechanism in *KadoP*.

Indexing and Query Processing. KadoP indexes element labels as well as words[2] in documents. We henceforth use *term* to refer to either of the two. The indexing scheme of KadoP is based on the *Term* relation, defined as follows:

Term(p,d,sid,l) l is the label of element (p, d, sid)
Term(p,d,sid,w) w is a word under element (p, d, sid)

We refer to a tuple in *Term* as a *posting*. Given a term a, we refer to the set of its postings as the *posting list* for a and denote it as L_a.

In KadoP, XML documents are stored at their publishing peer, whereas the *Term* relation is distributed among the peers of the system using a *distributed hash table* (DHT for short) [6]. At a high level, the DHT exposes the following interface: $locate(k)$ returns the id of peer in charge of key k; $put(k, \alpha)$ enters a new value for k; $get(k)$ returns the value for k; and, $delete(k)$ deletes the key k. Under the covers, the DHT assigns the keys automatically among the peers (typically through some hash function), and handles the redistribution of keys when peers join or leave the network. In KadoP, the keys of the relation are the terms and the values the corresponding posting lists. Thus, *Term* is split horizontally among peers, with peer p in charge of a portion $Term_p$ defined as follows: $Term_p$={*Term(p',d,sid,a)* | *locate(a)=p*}. The posting lists in $Term_p$ are clustered based on the term value, and the postings of a term are ordered in the lexicographic order dictated by the (p, d, sid) attributes.

The distributed *Term* relation essentially implements a data catalog that enables KadoP to locate documents matching a

[1]hal.archives-ouvertes.fr

[2]For efficiency, stop words are excluded from the index.

query Q. More specifically, assume that Q is submitted at peer p and let q be the corresponding tree pattern query. For each term a in q, p asks the peer in charge of a for the posting list L_a, and performs a holistic twig join [7] over all the received lists. For instance, assume that $q \equiv$ //abstract[contains(., "xml")]. The query peer p uses the *Term* relation to find the *sid*s of all elements labeled abstract and all parents of text nodes containing the word xml. A structural join on these *sid* collections computes the ids of documents that match q, and in effect, a superset of the documents that match Q. At this point, p can forward Q to the owning peers and collect the answers.

As the previous discussion demonstrates, the distributed index is key for efficient query evaluation in a P2P system. An important property of the KadoP index is that it identifies precisely the documents that contain results for q, which in turn can limit considerably the set of peers to which the complex query Q is forwarded. In contrast, other DHT-based XML indexing systems [8], [9] become imprecise in the presence of text predicates and may thus forward the query to a larger set of peers. We find it crucial to efficiently support index queries with text predicates, since XML blurs the distinction between structure (tags) and content (text), and such queries are thus frequent in practice.

Due to space constraints, we do not discuss here certain aspects of the KadoP system, such as, handling $*$-labeled queries, or index updates. The complete details can be found in the full version of this paper [10].

III. SCALING KADOP

As discussed earlier, the distributed index is an inherent component of query evaluation and its efficiency is thus crucial for good system performance. However, our experiments with an initial implementation of KadoP have demonstrated that the index has severe scaling limitations, caused primarily by the processing of long posting lists. (Similar points have been raised by other works on DHT-based systems [11], [12], [13].) Thus, for the remainder of the paper, we focus our attention on the evaluation of index queries that involve long posting lists since they represent the true challenge for a DHT-based approach.

We identify three important metrics that assess the performance of a distributed index (and essentially form the guiding criteria behind our techniques): (i) *indexing time*, (ii) *query response time*, and (iii) *bandwidth consumption*. (The latter refers again to index query processing, since transferring the results to the query peer is unavoidable.) In the remainder of this section, we discuss technical improvements we brought to the original KadoP system (that was described in [5]) that already greatly enhance the performance of the system in terms of (i) and (ii). The following sections discuss more involved techniques that aim to reduce query response time and bandwidth consumption.

Improving indexing time. To index a document, the system constructs in one traversal the element postings (Section II) and routes each posting, via the DHT, to the peer in charge of the corresponding term. Postings of the same term are buffered and sent in batches, which reduces slightly the index latency (the time it takes to index a document) compared to the naive method of routing each posting separately. More important gains are obtained by (i) extending the DHT API; and (ii) replacing its data store. We discuss these two points next.

The original *insert* operation in a local DHT index is very inefficient. According to the standard DHT API [6], when a peer in charge of a key k receives a *put* request, it (1) reads the old value for *k*, (2) applies a DHT-specific reconciliation of the old value and the new entry, and (3) inserts the result in the repository. Performing this operation for n successive entries associated to key k leads to a total I/O complexity of n^2, since i entries must be read in order to append the (i+1)th entry. To overcome this issue, we extend the DHT API described in Section II with a new operation, namely *append(key, entry)*, to obtain an indexing of linear cost.

To speed up indexing, we tuned the DHT's communication buffers to cope with many small messages generated by small posting lists. Another important improvement is the tuning of the index storage. More precisely, $Term_p$ at peer p is organized as a B+-Tree, using term as the search key, and the postings associated with a given term are lexicographically ordered by (p, d, sid). (*KadoP* is built on PAST [14] that by default uses gzipped XML files.)

In our experiments, enhancing the API, buffer tuning, and replacing the index storage has sped up publishing by two to three orders of magnitude. As a side effect, replacing the index storage has also reduced index query processing time by one order of magnitude.

Improving query response time. As discussed earlier, the peer p in charge of a query q performs a holistic twig join on the posting lists received from other peers. Observe that the only retrieval operation in the DHT API is *get*, which is defined as a blocking operation, i.e., it returns only when the content of the posting list has been fully retrieved. Therefore, the holistic twig join must wait until at least two lists have been entirely received before it can start processing. This poses serious performance problems when evaluating index queries and obviously limits system scalability. We therefore modified the DHT API (and the actual DHT system) by adding a *pipelined get* method, which transfers posting lists asynchronously. This simple modification brought important performance improvements. More concretely, KadoP implements a multi-threaded, block-based version of the holistic twig join from [7]. For each term a of query q, the peer p in charge of L_a runs a *producer*, whereas the query peer p runs a *consumer*, which is the holistic join. (We detect faulty peers with time-outs; when this happens, the answer is incomplete.) We assume that the consumer and the producers can synchronize via network pipes. The consumer runs on in-memory data-structures and is likely to run faster than producers, which have to read (potentially large) posting lists from disk and send them over the network. Since the join in itself is pipelined, the index query is processed at the speed of the slowest producer or at that of the slowest transfer.

IV. DPP Algorithm

The distribution of posting list sizes in real-life applications is very skewed, since it follows the typically skewed frequency distribution of terms. Moreover, the query distribution tends to follow the data distribution, implying that the large posting lists are accessed more frequently in index queries. These phenomena lead to a situation where a small number of very large posting lists become the dominant cost factor in processing index queries. In this section, we describe a novel data structure for managing long posting lists and a modified holistic twig join algorithm taking advantage of this structure. Together, they allow *parallelizing* posting list transfers, thereby reducing query response time. As we shall see, this will also result in reduced data transfers.

A. Distributed Posting Partitioning

The distributed posting partitioning (DPP for short) is a distributed hierarchical data structure for managing posting lists. The idea is to split the posting list for a popular term horizontally based on range conditions, and to migrate portions to other peers, in the style of distributed B-trees [15].

Consider a posting list L_a and in particular the lexicographical ordering dictated by its (p', d, sid) attributes. A *condition* C is an interval $[\alpha, \beta]$, with α the least tuple and β the largest. In a slight abuse of notation, we use C to refer also to the block of postings that satisfy the corresponding condition. For each C, C', we define $C \subseteq C'$ if each tuple satisfying C also satisfies C', $C \cap C'$ if there exists a tuple that satisfies both, and $C < C'$ if each tuple satisfying C is lexicographically less that all tuples satisfying C'.

A DPP is used to split a posting list for a given term over several peers. It comprises internal blocks and leaf blocks. Each internal block consists of conditions and corresponding "pointers" to other blocks, while a leaf block consists of a set of postings. Similar to B-trees, the entries in one block satisfy all the conditions from the root of the DPP-tree to that block. Formally, a DPP block is denoted as $(C_1...C_n, \varphi)$, where:

1) $C_1...C_n$ is a sequence of conditions such that $C_i < C_{i+1}$ for each i;
2) φ is a (pointer) function that assigns to each non-leaf C_i a pseudo-key leading to the corresponding DPP-block;
3) if $\varphi(C_i) = (C'_1...C'_m, \varphi')$, then $C'_j \subseteq C_i$ for each j.

It is important to stress that the pointer function ϕ implements pointers as pseudo-keys, which implies that DPP relies on the DHT in order to reach children blocks. In turn, this makes DPP oblivious to the placement of blocks at different peers, and thus resilient to changes in the composition of the P2P network. Moreover, as we discuss later, the DPP enables a more efficient twig join operator, since a posting list can be transferred in parallel, and, by using the range conditions, the join can focus only on those blocks that may contain join results.

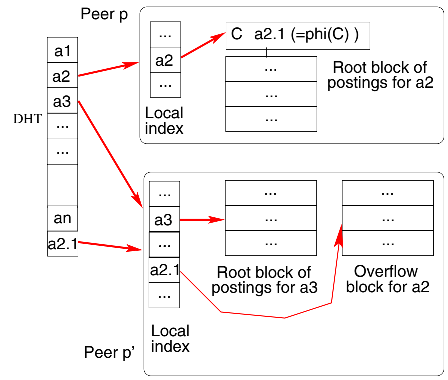

Fig. 1. The organization of Term

The implementation of DPP in KadoP uses the general organization for relation *Term* that is pictured in Figure 1. Originally, the entries of one posting list are all in one data block. The system sets a bound on the number of entries in a data block and a bound on the number of conditions in a condition block. When inserting entries, a block may overflow and be split. The smallest and largest elements of each new block determine the condition of the block. In principle, the DPP may need re-balancing, just like a B-tree. For reasonable block sizes (e.g., 1000 entries) of both the data and the condition blocks, the posting lists of up to 10^6 entries require a single level of indirection. Based on this observation, our implementation does not set a bound on the size of a condition block in KadoP and employs a two level DPP structure.

B. Query processing in DPP

We now describe a modification to the holistic twig join algorithm that exploits the parallel transfer opportunities provided by the DPP. Consider a query q with n nodes and assume that the postings for each query node, say i, are split into several blocks $(C^i_1, ..., C^i_{m_i})$. The idea is to perform the join on a per-block basis, allowing up to K parallel-running joins (where K is a parameter set in advance). The processing starts with an initialization step where only the conditions describing the DPP blocks are fetched, and a description for each parallel join is generated. During the join, the first K blocks for each posting are fetched in parallel, e.g., $C^1_1, \ldots, C^1_K$, $C^2_1, \ldots, C^2_K$, and so on until $C^n_1, \ldots, C^n_K$. The meaningful (pipelined) joins are computed in parallel. To increase the throughput, we do not require that results be produced in lexicographical order, and return to the user the first results produced by each join. When an active block in a posting list completes, the next block in this list is activated etc.

An issue is whether the algorithm generates too many joins. The answer is no, because the postings are lexicographically ordered. Indeed, suppose we are joining n postings, consisting respectively of $m_1, \ldots, m_n$ blocks. Then one can prove that we do not have to consider $m_1 \times \ldots \times m_n$, but at most $m_1 + \ldots + m_n$ joins, and in practice, much less.

Another important point of our approach is that the join operator has to fetch a block only if the latter has a chance to provide matches, i.e., the corresponding condition has a non-empty intersection with the conditions of other blocks. Moreover, the join operator can compute the portion of a block which has a chance to find matches in the other blocks, and only fetch that useful portion.

To conclude this section, we note that it is possible to employ other well-known distributed query optimization techniques [16]. For instance, some structural joins can be pushed to the peer holding the longest posting list in the query, thus reducing data transfers. Since such optimizations are standard, we do not consider them here. Also, the transfer of posting lists can be optimized by replicating them and transferring fragments from different copies in parallel. Similarly, DPP blocks can also be replicated based on their access frequency. Although the DHT already does replicate its index for reliability, this replication does not fit our needs. Its main drawback is that the replication factor is *fixed* at the time of the initialization of the DHT. In contrast, we need here a finer control of the replication degree, on a block-by-block basis. We plan to investigate this particular point as part of our future work.

C. Experiments

In this section, we present an experimental study on the effectiveness of the DPP scheme.

The study uses a deployment of actual KadoP peers on the Grid5000 platform (www.grid5000.fr), a testbed for wide-area distributed applications that spans 9 cites in France. Each Grid5000 node has 2 CPUs, and the nodes are connected in a 10GB network. As we could not reserve a large number of nodes, we deployed 10 KadoP peers per Grid5000 node. We report on experiments on up to 500 peers (so 50 Grid5000 nodes). Our implementation of DPP employs a maximum block size of 4MB before it performs a split. In all our experiments, we apply the optimizations described in Section III, since they brought significant improvements.

We have used the Aug. 2006 version of the real-life DBLP bibliographic data (340 MB, available at dblp.uni-trier.de). To experiment with larger data volumes, we cut the DBLP corpus in small XML documents of 20 KB each, and publish several copies of the same documents when larger volumes are needed. In all experiments, the data set is split evenly among all the publishing peers. It is interesting to note that several tags in the DBLP data set generate large posting lists. Even for a 200 MB fragment of DBLP data, there are posting lists larger than 200K entries for inproceedings, 1M entries for author, and 500K entries for title, to name a few. Observe that these frequent tags are typically queried often.

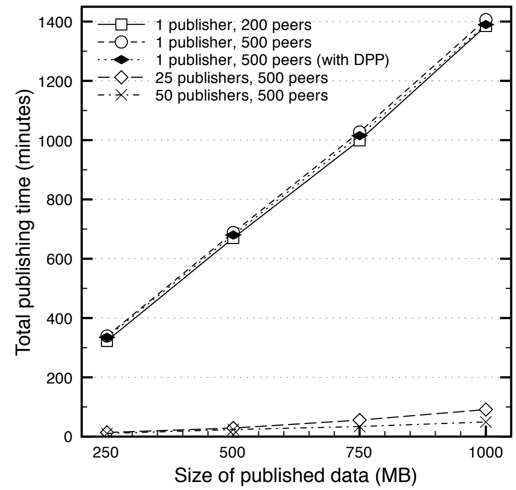

Fig. 2. Indexing time.

Indexing time. Figure 2 reports the time to build the distributed index as we vary the total size of published data (over all peers). We test several configurations for the size of the KadoP network and for the number of peers that publish data. Thanks to our robust replacement of the DHT's index store, publication now scales linearly in all settings. When 1 peer publishes, the network size increase from 200 to 500 peers brings a negligible overhead, demonstrating that *locate()* costs incurred by the DHT are small. Also with 1 publisher, the usage of a DPP brings a negligible overhead when compared with the default KadoP index. This demonstrates that DPP block splitting has a moderate cost. Most importantly, Figure 2 shows that many publishers drastically cut indexing time, as they work in parallel.

Query response time. Figure 3 reports index-query evaluation times for the query: //article//author//Ullman. The query was chosen to stress test our approach as it involves the processing of the long posting list for author. The results clearly demonstrate the benefits of the DPP: query processing is cut by a factor of three, and its growth is really slow as the data volumes grow. With or without the DPP, query processing is network-bound. When the DPP is used, the largest posting list fragment stored on a peer is of moderate size, thus transfer time does not grow much as the size of the indexed data grows.

Traffic consumption. The next experiment measures the volume of network traffic during query execution. In particular,

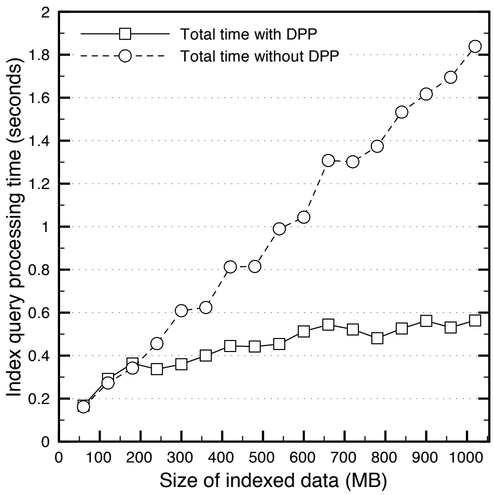

Fig. 3. Query response time.

we examine a scenario where several data-intensive queries are executed concurrently, as follows: we create a workload of 50 queries, each of which involves at least one term with a long posting list, and submit them for execution at 50 distinct nodes in intervals of 5 minutes. (This results in a medium throughput of one query every 6 seconds.) We repeat the test for several values of the total indexed XML data (DBLP documents).

The total traffic registered for 200MB, 400MB, 600MB, and 800MB of published XML data is 32MB, 66MB, 95MB, and 127MB respectively. Clearly, these volumes are not a problem for the 10GB network of our experimental platform. On the other hand, they can be prohibitively high for other network deployments. (This is precisely what motivated the work on Structural Bloom Filters that we present in the next section.) We mention that these values are registered using a simple query execution plan, where all postings are transferred at the peer that executes the query. We are currently developing a cost model and an optimizer to select the best execution plan that minimizes query response time or traffic consumption, depending on the setting.

V. Structural Bloom Filters

We introduce Structural Bloom Filters, a mechanism for reducing the volume of transferred data during the evaluation of index queries. Our techniques are inspired by the use of simple Bloom Filters in the evaluation of distributed relational joins. The details are more involved, however, as we deal with structural joins over tree-structured data.

The proposed mechanism includes two kinds of filters, namely, the *Ancestor Bloom Filter* and the *Descendant Bloom Filter* (AB Filter and DB Filter for short respectively). In the sequel, we describe in detail AB and DB Filters and introduce strategies that integrate them in query processing.

[1							8]	j=3
[1			4]	[5			8]	j=2
[1	2]	[3	4]	[5	6]	[7	8]	j=1
[1,1]	[2,2]	[3,3]	[4,4]	[5,5]	[6,6]	[7,7]	[8,8]	j=0

Fig. 4. Dyadic decompositions for $[1, 2^3]$.

Before continuing with our presentation, we discuss briefly two key mechanisms on which we develop our framework: a canonical base for representing arbitrary intervals (a *dyadic* base), and conventional Bloom Filters.

Dyadic Intervals. Let l be a positive integer and consider the interval $[1, 2^l]$. The dyadic decomposition of $[1, 2^l]$ at level j, $0 \leq j \leq l$, is its partition in 2^{l-j} disjoint intervals of length 2^j, termed *dyadic* intervals. Figure 4 shows an example of this decomposition for $l = 3$. We use I_{ij} to refer to the i-th dyadic interval at level j and $\mathcal{I} = \{I_{ij}\}$ to refer to the complete set of dyadic intervals. It is easy to see that an arbitrary interval $[x, y] \subseteq [1, 2^l]$ can be represented as the union of at most $2 \cdot l$ disjoint intervals from $\mathcal{I}$. Moreover, there is a unique representation that contains the least number of intervals, termed the *dyadic cover* of $[x, y]$ and denoted as $\mathcal{D}[x, y]$. Returning to the example of Figure 4, $\mathcal{D}[1, 7]$ is $\{[1, 4], [5, 6], [7, 7]\}$. A dyadic interval containing an interval $[x, y]$ is called a *dyadic container* of $[x, y]$. The set of dyadic containers of $[x, y]$ is denoted $\mathcal{D}_c[x, y]$. For instance, $\mathcal{D}_c[3, 4] = \{[3, 4], [1, 4], [1, 8]\}$.

Bloom Filter. A Bloom Filter [17] provides a concise representation of a set S in a form that is suitable for membership queries. The filter consists of a vector $\mathcal{F}$ of n bits (initialized to zero) and a set of hash functions $H_1, \ldots, H_k$. An element $e \in S$ is inserted in the Bloom Filter by setting bit $\mathcal{F}[H_i(e)]$ to 1, for every $1 \leq i \leq k$. Similarly, a membership query for an element e is answered positively if all bits $\mathcal{F}[H_i(e)]$ are equal to 1. We refer to these operations as an insert and a look-up respectively.

Clearly, the Bloom Filter always returns true for the look-up of an inserted element. On the other hand, a look-up of an element not in S may return a *false positive* answer due to collisions in the hash functions. The probability of obtaining a false positive, denoted as fp, is termed the *false positive rate* of the filter and it can be computed based on $|S|$ and the parameters n and k. For a given set and a given false positive rate, it is possible to choose k so that n is minimal, i.e., communications are minimized. An essential aspect of a Bloom Filter is that the vector is typically much smaller than the set that it encodes, so its transmission costs much less. The trade-off is the introduction of false positive errors when the filter is used for membership queries.

A. Filter Definition

We now present the details of the proposed bloom filter mechanism. We begin with the AB Filter, and then sketch the ideas behind the DB Filter.

AB Filter. Consider two tags a and b and the respective posting lists L_a and L_b. The AB Filter for L_a, denoted as $ABF(a)$, enables the filtering of L_b and the computation of a sub-list $F(b, ABF(a))$ that contains a superset of $b[\backslash\backslash a]$, i.e., the set of L_b postings that have an ancestor in L_a. The main idea is that $F(b, ABF(a))$ can be used in lieu of L_b in the structural join $a//b$, without compromising the accuracy of the result.

We now describe the theoretical underpinnings of the AB Filter mechanism. Consider a posting $e_a = (p_a, d_a, start_a{:}end_a{:}lev_a) \in L_a$ and a posting $e_b = (p_b, d_b, start_b{:}end_b{:}lev_b) \in L_b$. Clearly, e_b is a descendant of e_a iff $p_a = p_b$, $d_a = d_b$ and $[start_b, end_b] \subseteq [start_a, end_a]$. The key observation is that we can express the previous containment condition in terms of the dyadic covers of the two intervals. More concretely, one can show that $[start_b, end_b] \subseteq [start_a, end_a]$ iff for each interval $I \in \mathcal{D}[start_b, end_b]$ there exists an interval $I' \in \mathcal{D}[start_a, end_a]$ that contains it, i.e., such that $I \subseteq I'$. Alternatively, this can be expressed as $\mathcal{D}_c(I) \cap \mathcal{D}[start_a, end_a] \neq \emptyset$ and can thus be realized by probing $\mathcal{D}[start_a, end_a]$ with every interval in $\mathcal{D}_c(I)$. This suggests the following generalization to postings. We define the *cover* of a posting $e_a = (p_a, d_a, start_a{:}end_a{:}lev_a)$ as $\mathcal{D}(e_a) = \{(p_a, d_a, I) \mid I \in \mathcal{D}[start_a, end_a]\}$, and $\mathcal{D}(L_a) = \cup_{e_a \in L_a} \mathcal{D}(e_a)$. The containers $\mathcal{D}_c(e_a)$ and $\mathcal{D}_c(L_a)$ are defined similarly. The following theorem formalizes a condition so that $e_b \in b[\backslash\backslash a]$ and provides the foundation for the AB Filter:

Proposition 1: For each $e_b \in L_b$, $e_b \in b[\backslash\backslash a]$ iff for each (p_b, d_b, I) in $\mathcal{D}(e_b)$, there exists I' in $\mathcal{D}_c(I)$ such that (p_b, d_b, I') in $\mathcal{D}(L_a)$.

We are now ready to define the AB Filter mechanism. The AB Filter comprises: (a) a *tracing function* $\psi : [0, l] \rightarrow [1, \infty]$ (whose role we will describe shortly), (b) a Basic Bloom Filter that encodes the set $\mathcal{D}(L_a)$, and (c) an integer $DCLev$ that records the highest level $j \leq DCLev$ such that an interval I_{ij} appears in $\mathcal{D}(L_a)$.

To insert a posting e_a in the AB Filter, we compute its cover $\mathcal{D}(e_a)$ and insert its elements in the Basic Bloom Filter. More precisely, given an element $(p_a, d_a, I) \in \mathcal{D}(e_a)$ such that I is at level j of the dyadic decomposition, we insert $\psi(j)$ replicas (or, traces) of it in the Basic Bloom Filter. The intuition is that an increased $\psi(j)$ makes it less likely to observe a false positive interval at level j and thus controls the accuracy of the filtering mechanism. To check a posting e_b against the AB Filter, we proceed as follows. For each interval $(p_b{:}d_b{:}I)$ in the cover $\mathcal{D}(e_b)$, we test whether $(p_b{:}d_b{:}I')$ is in the Bloom Filter for some I' in the container $\mathcal{D}_c(I')$. Note that this implies $\psi(j)$ probes to the Basic Bloom Filter if I' is at level j. Moreover, it suffices to test container intervals up to level $DCLev$, since the AB Filter does not contain intervals past this recorded level. If we cannot find any such I', then I is not covered by any interval in $\mathcal{D}(L_a)$ and we can decide by Proposition 1 that e_b is not in $b[\backslash\backslash a]$. If every I in $\mathcal{D}(e_B)$ is covered then we conclude that e_b is in $b[\backslash\backslash a]$, and this is correct up to collisions in the Bloom Filter.

As noted earlier, the tracing function $\psi(j)$ controls the accuracy of the filter at level j of the dyadic decomposition. Intuitively, a false positive interval at a high level provides coverage for more intervals in L_b and thus increases the error rate of the filter. At the limit, a collision with $(p_b, d_b, [1, 2^l])$ will cause the AB Filter to falsely select every b-element in document (p_b, d_b). This connection suggests using more "traces" at higher levels to boost the accuracy of the filter. We describe our choice of ψ at a later point, after we analyze the false positive probability of an AB Filter.

We now examine the error probability of an AB Filter. We define the *Ancestor false positive rate* (for a and b), denoted $fp^A(a, b)$ (or simply fp^A when a, b are understood), as the probability that an AB Filter falsely identifies a b posting as a member of $b[\backslash\backslash a]$. Let $fp[\psi]$ be the false positive rate, i.e. the probability that the underlying Bloom Filter returns a false positive answer. Note the dependency of the probability to ψ, since the latter affects the number of insertions in the Bloom Filter. The following theorem captures the relationship between fp and fp^a:

Proposition 2: The Ancestor false positive rate is bounded as follows: $fp^A \leq 1 - \prod_{0 \leq j \leq l} (1 - fp[\psi])^{\psi(j)}$.

It becomes clear that function ψ encodes a trade-off: as we increase $\psi(j)$, we increase the number of insertions and thus $fp[\psi]$, but we also increase the number of probes and thus decrease $fp[\psi]^{\psi(j)}$. One interesting observation is that the AB filter is not likely to encode many dyadic intervals at high levels, since the structural ids for real-life data sets tend to have narrow intervals. (This happens because XML data tends to be shallow and bushy.) As a result, an increased $\psi(j)$ for high levels is not likely to affect significantly the total number of insertions in the filter. At the same time, decreasing $fp[\psi]^{\psi(j)}$ is crucial as j increases, due to the increased coverage of the corresponding dyadic intervals (see also our earlier discussion). We adopt these ideas in our work and we employ the function $\psi[j] = \lceil 1+j/c \rceil$ (for some integer $c \geq 1$) that essentially adds one extra trace every c levels. As we show in the full version of the paper, this function ensures that every interval I_{ij} has the same expected effect[3] on the error rate provided that $fp[\psi] < 1/2^c$. In our experiments, we set $c = 4$ as we expect the basic false positive rate to be less than $1/16$.

DB Filter. The DB Filter for L_b, denoted as $DBF(b)$, can filter the postings in L_a to obtain a sublist $F(a, DBF(b))$ that contains a superset of $a[//b]$, that is, the postings in L_a that have at least one descendant in L_b. The key idea remains

[3]We measure the effect of I_{ij} as the expected number of covered descendant intervals if I_{ij} is false positive.

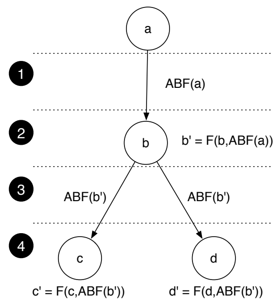

Fig. 5. AB Reducer

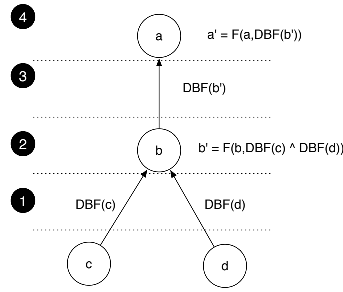

Fig. 6. DB Reducer

essentially the same: we send a Bloom Filter that traces of the dyadic intervals of b postings, and then perform membership tests for the a postings. The filtering mechanism is formalized by the following theorem:

Proposition 3: For each $e_a \in L_a$, $e_a \in a[//b]$ iff $\mathcal{D}(e_a) \cap \mathcal{D}_c(L_b) \neq \emptyset$.

Essentially, the DB Filter consists of a Basic Bloom Filter. To insert a posting e_b in L_b, we insert in the Basic Bloom Filter every interval in $\mathcal{D}_c(e_b)$. To check membership for a posting e_a in L_a, it suffices to perform a look-up in the filter for each element in $\mathcal{D}(e_a)$. We observe that a posting typically entails more insertions in the DB Filter than in the AB Filter, since the dyadic cut tends to contain more intervals compared to the dyadic cover. So, intuitively we should expect a DB Filter to have a higher space overhead compared to the AB Filter or, equivalently, less accuracy for the same storage space.

We do not discuss the technique further as it is very similar to that of the AB Filter. The complete details can be found in the full version of this paper.

B. Query Evaluation with Bloom Filters

We introduce three query processing strategies based on Structural Bloom Filters, namely, *Ancestor Bloom Reducer*, *Descendant Bloom Reducer*, and *Bloom Reducer* that can be seen as a hybrid of the previous two. The proposed strategies proceed in two phases: in the first phase, the peers exchange structural Bloom Filters and reduce their posting lists; in the second phase, the reduced lists are sent to the query peer for the final join. The strategies essentially differ in the realization of the first filtering phase. Figure 5 depicts the filtering phase of Ancestor Bloom Reducer (AB Reducer, for short) on the example query //a//b[//c][//d]. In a nutshell, each peer receives an AB Filter from its parent, filters its postings, and forwards an AB Filter of the reduced postings to its children peers. Thus, peers (except the root) filter their postings according to the corresponding incoming path from the root query variable. Descendant Bloom Reducer (DB Reducer, for short) follows an inverse process, forwarding DB Filters along the leaf-to-root paths and essentially filtering based on outgoing paths. (This is shown in Figure 6.) Finally, Bloom Reducer performs a combination of the two previous strategies: it initially forwards AB Filters top-down, and then DB Filters bottom-up.

Clearly, the filter-based strategies are efficient if the savings in the transfer of the reduced list can offset the cost of building, transferring, and processing the Bloom Filters. As with relational bloom filters, we expect these factors to depend heavily on the data and query characteristics. We examine this point in more detail in the next section.

C. Experiments

In this section, we present the results of an experimental study to evaluate the performance of AB and DB Filters.

Filter Sensitivity Analysis. The first set of experiments performs a sensitivity analysis of the structural filters. We use the simple query $a//b$ and consider two scenarios: filtering b with $ABF(a)$, and filtering a with $DBF(b)$. We measure filter performance as the fraction of false positive answers. We term this metric the *empirical false positive rate of the filter*.

Due to space constraints, we only present a brief overview of our findings. (The complete experiments can be found in the full version of this paper.) Our experiments have indicated that the AB Filter achieves a lower error probability compared to the DB Filter when they both use an equally accurate Basic Bloom Filter. For instance, the error rate of the AB filter remains below 10% even when $fp[\psi]$ reaches 20%, whereas the error rate of the DB Filter remains below 10% only when $fp[\psi] < 5\%$ and rises to over 50% as $fp[\psi]$ increases. The difference is due mainly to the tighter probing mechanism of the AB Filter. Recall that the answer of the AB Filter is generated through a conjunction of containment predicates, which in turn reduces exponentially the probability of committing an error. The DB Filter, on the other hand, relies on a disjunction of probes that proves detrimental for the overall error rate. Our results have also demonstrated the benefits of the proposed ψ function for the AB Filter. For a filter of the same size, the proposed function achieved a lower error rate compared to the default function that uses a single trace per level.

Performance of Filter-based Query Strategies. In the next set of experiments, we examine the performance of the three query evaluation strategies that we have introduced earlier, namely, AB Reducer, DB Reducer, and Bloom Reducer. We use the total volume of transferred data as the performance metric for each strategy, since this is the major cost factor in distributed query evaluation over wide-area networks. For each strategy, we report its total data volume normalized by the amount of data shipped by the conventional query processing strategy. Thus, a normalized data volume of 0.4 implies that the strategy transfers 60% less data overall. We base our evaluation on the real-life DBLP data set described in

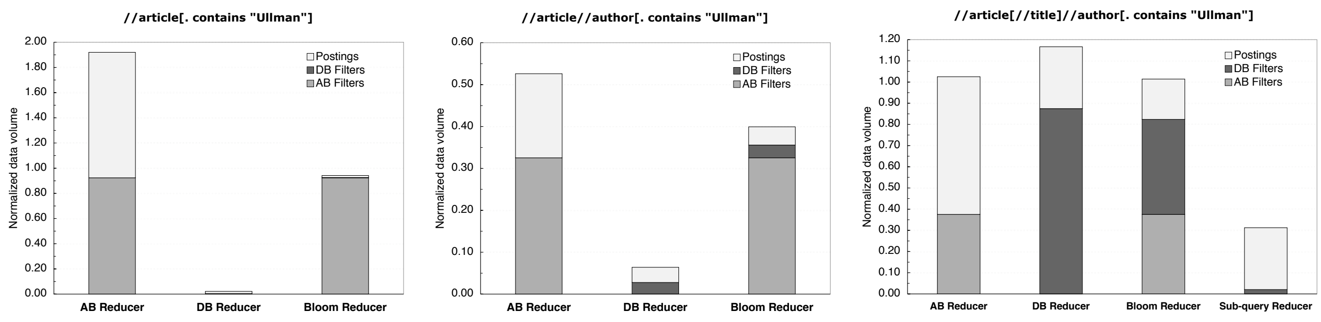

Fig. 7. Performance of Bloom-based strategies for different queries

Section IV. In all cases, AB and DB Filters are initialized with a basic false positive rate of 20% and 1% respectively. The idea is to allocate fewer bits to AB filters since our previous experiments have shown their resilience to errors in the Basic Bloom Filter.

Figure 7(a) shows the performance of the three strategies on the simple query //article[.contains "Ullman"]. (The graph breaks down the normalized data volume in terms of the size of AB and DB filters, and the size of the filtered posting lists.) We observe that DB Reducer is very effective in filtering postings that are irrelevant to the query, leading to a reduction of more than 90% in transfer load. Essentially, the keyword predicate is very selective as there are relatively few Ullman postings (compared to the number of article postings) , and this leads to a DB Filter that can select very effectively the matching postings of article. In contrast, Bloom Reducer and AB Reducer are less effective as they transfer a large AB filter on article, without getting any significant benefits from filtering the small list of Ullman.

Figure 7(b) depicts the performance of the three strategies on the slightly more involved query //article//author[.contains "Ullman"]. The injection of author is interesting, as it represents one of the largest posting lists in this data set. We observe that AB- and Bloom Reducer become more efficient than in the previous experiment, since the overhead of the AB filter on article is now offset by the savings of reducing author, the dominant list in this query. DB Reducer remains the dominant strategy, as the DB filter on Ullman is still the most cost-effective filter for this query.

The final experiment, shown in Figure 7(c), evaluates the performance of the three strategies on the branching query //article[//title]//author[.contains "Ullman"]. (The Figure also depicts a fourth strategy that we will discuss later.) Clearly, the proposed strategies do not enable any savings for this particular query. This is due to the existence of the title branch, which has a detrimental effect on the performance of each strategy. For DB Reducer, the branch leads to the creation of a large DB Filter that is not useful in filtering article elements. (Essentially, all articles have a title.) For AB Reducer, the AB filter on article is not sufficient to filter the title postings, and this leads to a high number of unfiltered postings. Finally, Bloom Reducer suffers from a combination of the previous two factors as it is a hybrid strategy.

Overall, Structural Bloom Filters can enable a significant reduction in the volume of transferred data. Our results indicate that there is no dominant strategy, as the performance depends heavily on the characteristics of the query and the data. In our current work, we employ the following simple heuristic in order to select the filtering strategy: we identify the subset of the query that has a guaranteed low selectivity factor, by examining the sizes of the stored posting lists, and we apply Structural Bloom Filters on the specific subset. Of course, this implies that only lists that correspond to the selected sub-query will be filtered, but this can still yield significant savings if the lists are large. To verify this, we have applied the DB Reducer strategy on the subset //article//author[.contains "Ullman"] of the previous query and have thus excluded title from filtering. (Thus, title is sent to the query peer in its entirety.) The performance of this approach is plotted in Figure 7(c) as the fourth strategy. As shown, the modified strategy offers close to 70% of savings in total transfer load. As part of our future work, we plan to investigate more principled optimization techniques that select the optimal strategy based on a formal cost model.

VI. Related Work

Earlier works [18], [3] have studied P2P data sharing over unstructured networks, where queries may be multi-cast to all peers, significantly increasing network traffic. IrisNet [19] uses a hierarchical peer network. In contrast, the structured network architecture of KadoP allows the system to maintain a data index and thus route the query only to a subset of relevant peers. Other studies of P2P keyword search [20], [21] have observed the difficulty of handling long posting lists on DHTs. Our work addresses the same problem, albeit in the context of tree queries over XML data.

XML data management based on DHT networks is considered in [8], [9], [22]. Indexing in [8] is based on XML linear paths, which makes the index imprecise (i.e., leading to false positives) for tree queries. Essentially, the index query returns an answer for a document if the latter contains independent matches for the paths in the tree query. In contrast, KadoP indexes the complete structure and thus achieves a more precise routing of queries to peers. Indexing in [9] and [22] does not consider words occurring in text, thus the index cannot help for such queries. Moreover, query processing in these systems is affected by query syntax, e.g., "//a//b" can be more expensive to process than "/a/b" even if the results are the same. This is not the case in KadoP.

Recent works have proposed DHT structures which, unlike Pastry used in KadoP, also support range searches [12]. Integrating such a DHT in KadoP would allow it to process index queries with range conditions.

The proposed structural Bloom filters are inspired by the use of basic Bloom filters for distributed relational joins. Our mechanism is more involved, as we are dealing with structural joins over hierarchical data. The dyadic decomposition framework has been used in several applications, such as, the estimation of quantiles over streams [23], or approximating the selectivity of spatial joins [24]. None of the previous works, however, has considered the application of this mechanism and its analysis in the context of XML filtering with one-sided errors. Finally, earlier studies have considered the use of Bloom Filters as XML document summaries that can estimate the selectivity of linear path expressions [25], evaluate the similarity between documents [26], or identify documents that do not match a given path query [26]. Our proposed filtering mechanism, on the other hand, tackles a very different problem, namely, building a concise summary of a posting list that can be used to filter any other posting list with respect to the “//” axis.

VII. Conclusions

Our work is motivated by the scalable indexing of distributed XML data. The techniques that we presented are implemented in the KadoP system, which is currently being deployed in the context of the Edos application. As part of our ongoing work, we are investigating several improvements on the KadoP system. We have started designing a query optimizer able to support standard distributed database optimization techniques as well as the Bloom-based strategies presented in this paper. Answering queries using a cache of queries similar to [27], is a future challenging problem in the distributed setting of KadoP . We are also exploring techniques to improve index construction time, as it remains significant when a large collection of documents is published. Finally, we are working to improve the system’s existing support for peers joining and leaving the network.

Acknowledgments. We thank Sophie Cluet, Karl Schnaitter, and Nitin Gupta for discussions on P2P XML indexing. We thank Gabriel Vasile for participating to implementing KadoP. We thank Grid5000, the wide distributed infrastructure, for the support provided in testing KadoP, and Simon Laurent for helping us with the first tests of KadoP on the university’s cluster.

References

[1] S. Abiteboul, I. Manolescu, and N. Preda, “Constructing and querying peer-to-peer warehouses of XML resources.” in *SWDB*, 2004.

[2] R. Huebsch, B. Chun, J. Hellerstein, B. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. Yumerefendi, “The architecture of PIER: an internet-scale query processor,” in *CIDR*, 2005. [Online]. Available: citeseer.ist.psu.edu/article/huebsch05architecture.html

[3] W. S. Ng, B. C. Ooi, K. Tan, and A. Zhou, “Peerdb: A P2P-based system for distributed data sharing.” in *ICDE*, 2003.

[4] “Edos project: Environment for the development and distribution of open source software,” http://www.edos-project.org.

[5] S. Abiteboul, I. Manolescu, and N. Preda, “Constructing and querying P2P warehouses of XML resources,” in *ICDE (demo)*, 2005.

[6] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, “Towards a common API for structured P2P overlays.” in *Proc. of IPTPS*, 2003.

[7] N. Bruno, N. Koudas, and D. Srivastava, “Holistic twig joins: optimal XML pattern matching.” in *SIGMOD*, 2002.

[8] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt, “Locating data sources in large distributed systems.” in *VLDB*, 2003.

[9] A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain, “XPath lookup queries in P2P networks.” in *WIDM*, 2004.

[10] S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun, “Xml processing in dht networks,” http://gemo.futurs.inria.fr/projects/KadoP/perf.pdf.

[11] K. Aberer, A. Datta, M. Hauswirth, and R. Schmidt, “Indexing data-oriented overlay networks.” in *VLDB*, 2005.

[12] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram, “P-Ring: An efficient and robust P2P range index structure,” in *SIGMOD*, 2007.

[13] B. Loo, J. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica, “Enhancing P2P file-sharing with an internet-scale query processor,” in *VLDB*, 2004.

[14] A. I. T. Rowstron and P. Druschel, “Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage,” in *SOSP*, 2001.

[15] P. Krishna and T. Johnson, “Index replication in a distributed B-tree.” in *COMAD*, 1994.

[16] M. T. Ozsu and P. Valduriez, *Principles of distributed database systems*. Prentice-Hall, 1991.

[17] B. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, 1970.

[18] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov, “The Piazza peer data management system.” *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 7, 2004.

[19] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan, “An architecture for a world-wide sensor web.” *IEEE Pervasive Computing*, vol. 2, no. 4, October-December 2003.

[20] J. Li, B. Loo, J. Hellerstein, M. F. Kaashoek, D. Karger, and R. Morris, “On the feasibility of peer-to-peer web indexing and search.” in *Proc. of IPTPS*, 2003.

[21] P. Reynolds and A. Vahdat, “Efficient peer-to-peer keyword searching.” in *Int.’l Middleware Conf.*, 2003.

[22] G. Skobeltsyn, M. Hauswirth, and K. Aberer, “Efficient processing of XPath queries with structured overlay networks.” in *OTM Conferences (2)*, 2005.

[23] S. M. A. C. Gilbert, Y. Kotidis and M. J. Strauss., “How to summarize the universe: Dynamic maintenance of quantiles,” in *VLDB*, 2002.

[24] A. Das, J. Gehrke, and M. Riedewald, “Approximation techniques for spatial data,” in *SIGMOD*, 2004.

[25] W. Wang, H. Jiang, H. Lu, and J. X. Yu, “Bloom histogram: Path selectivity estimation for xml data with updates.” in *VLDB*, 2004, pp. 240–251.

[26] G. Koloniari and E. Pitoura, “Content-based routing of path queries in peer-to-peer systems.” in *EDBT*, 2004.

[27] B. Mandhani and D. Suciu, “Query caching and view selection for xml databases,” in *VLDB*, 2005.