XML processing in DHT networks

Serge Abiteboul[1]
serge.abiteboul@inria.fr

Ioana Manolescu[1]
ioana.manolescu@inria.fr

Neoklis Polyzotis[2]
alkis@cs.ucsc.edu

Nicoleta Preda[1]
nicoleta.preda@inria.fr

Chong Sun[2]
sunchong@soe.ucsc.edu

[1] INRIA Futurs & U. Paris,* Orsay, France [2] UC Santa Cruz,† USA

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 o ftechniques to lift them. First, we adapt the DHT platform to the needs of massive data processing. (This primarily consists of replacing the DHT store by an efficient native store and in streaming the communications with the DHT.) Second, we introduce a distributed hierarchical index and efficient algorithms taking advantage of this index 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 DHT-based XML processing system, used in a real-life software manufacturing application. We present experiments that demonstrate the benefits of the proposed techniques.

Keywords: *DHT, XML, P2P, bloom filters, distributed XML query processing, parallel query processing,*

1 Introduction

The current development of peer-to-peer (P2P) information sharing has opened the way for supporting high-level data management applications and in particular structured queries in a P2P setting. The vision is to enable a host of data management applications that can be realized today only through centralized and typically expensive infrastructures. One such example is the creation of ad-hoc content sharing communities, where users share and query information within some specific domain. Another one is the deployment of distributed database systems, where the principles and technology behind P2P enables an organic scaling to a high level of parallelism.

An issue of particular interest is the P2P management of distributed XML data. XML has emerged as a de-facto standard for data exchange and integration over the Internet, and is thus 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 [5] 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[1] and share the tasks of indexing the data and processing queries. Following common practice, KadoP indexes the XML data in the form of postings, where each posting encodes information on an element or a keyword. Given a query, the system combines the postings stored in the index to locate the peers that can contribute to the query, and subsequently forwards the query to these peers where the final results are computed.

A main issue with P2P indexing that could limit the

*Projet Gemo, Parc Club Orsay-University, 4 rue Jean Monod, 91893, Orsay Cedex, France

†Jack Baskin School of Engineering, Computer Science Departament, Engineering 2 Building, Santa Cruz, 95064, USA

[1]They publish also XML schemas, ontologies and Web services but, to simplify, these aspects will be mostly ignored here.

scalability of such system is the processing of large sets of postings [33]. This processing is first costly at the time of index construction, so (i) may delay the publication of documents. More critically, the transfer of large sets of postings may be damaging in terms of (ii) query response time and (iii) data transfer load. The purpose of this paper is to present techniques that we have developed to address these three issues. The techniques are fairly general as they can be applied in any XML system that follows the same general architecture as KadoP. Moreover, they are largely orthogonal and this allows for their easy composition.

To improve query processing 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 different peers based on range conditions. This scheme enables a highly parallel twig join algorithm that can reduce significantly the total processing time. Moreover, it allows the index to filter partitions that are irrelevant to the query, thereby saving on data transfers.

To limit data transfers, we also draw inspiration from distributed relational databases and 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 a technique, termed Bloom Reducer, that integrates these filters in the evaluation of index queries.

The third technique tackles the size of the index which is a serious concern for efficient query processing. When there are lots of *includes* (which often occurs in real applications), we want to avoid indexing the same information many times. Towards that goal, we introduce techniques for indexing what we call "intensional data" that have *includes* as a foremost example. Although the size of the index is reduced, the recall remains 100%, i.e., no answer is missed. We also briefly mention techniques that improve performance by reducing the size of the index at the cost of possibly missing some answers. This opens new avenues for performance improvements. However, developments in that direction are left for future research and the primary focus of this paper is on answering queries with full recall.

We present experimental results that demonstrate that the system scales gracefully to a large volume of data. We validate the effectiveness of the proposed techniques on synthetic and real-life data sets. The experimental study uses an implementation of the techniques in the KadoP system and is one of the first such studies of a real XML data management system over a P2P network, as opposed to experiments over simulated networks. (See [29, 21], for another full-fledge systems with an experimental study). It can therefore serve as a blueprint for future DHT-based XML management applications. Although there is an important overhead in running such experiments, we believe that there are essential to complement simulation-based studies. The development of the system proved invaluable for highlighting important limitations of the present DHT technology and served both as a motivation for the techniques presented here and as a test-bed for them.

We note that the modified system is currently being tested in cooperation with the Mandriva company (originally known as MandrakeSoftware) with a real application entitled Edos. In Edos [16], the data consists of the Mandriva Linux distribution, i.e., about 10 000 software packages and the associated metadata. The metadata for one Mandriva distribution is more than 100 megabytes of XML data. Of course, the system has to support simultaneously many (time-based) versions of the distribution. The peers (when the application will be deployed) will be Mandriva Linux developers, so potentially a population in the hundreds of peers.

The paper is organized as follows. The KadoP system [6] is presented in Section 2. Section 3 describes important techniques for making it scale. Section 4 introduces a technique based on partitioning and distributing index blocks, which greatly reduces query response time. Bloom-based optimization reducing network traffic are described in Section 5. Indexing of intensional data is discussed in Section 6. Related works are considered in Section 7. Finally, we conclude.

2 The KadoP system

We target applications with a possibly very large number (millions) of XML documents stored in a large number (tens of thousands) of peers, where the peer volatility is not very high. Issues such as popular terms and locality would have to be tackled for larger volumes. Also, the size of posting sets and the cost of copying them become serious concerns when peers are very volatile. We plan to address these issues as part of our future work. Observe however that, as it is, our system already meets the needs of a very wide range of applications.

For clarity, we focus on simple tree pattern queries that correspond to a subset of XPath. Our algorithms extend easily to more complex tree pattern queries, such as those that can be extracted from XQuery queries [12]. The focus of the paper is on efficient query processing, so we ignore several aspects of the KadoP system, such as XML schemas, ontologies, and Web services.

KadoP processes queries in two phases, that may be interleaved. First, an index query is computed to focus on a hopefully small set of documents that (potentially) match the query. Then the query is sent to the peers holding these documents and the answers are computed there. This shows the importance of indexing in our system. In what follows, we introduce the data model behind KadoP, and discuss the salient features of the indexing scheme and the query processing framework.

Data and Query Model We assume the existence of alphabets *uri*, *peer*, *doc*, *label*, *word*, *int*, that are all subsets of the alphabet *string*. Each peer and each document is identified on the Web by a uri. Each peer in the system is also internally identified by an integer and each document by a pair (p, d) of integers, where p is the identifier of the peer that checked it in, and d the document identifier within this peer. The database consists of the documents in the distributed collection and of the relations: $Peer(p, uri)$ (peer p has uri uri) and $Doc(p, d, uri)$ (doc (p, d) has uri uri). We note that KadoP uses other relations, e.g., for XML schema information, but we omit them to simplify exposition.

A *document* (p, d) is as 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 symbol from *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. Observe that (p, d, sid) provides an element identifier that is globally unique in the collection of documents. Structural ids allow deciding if element e_1 is an ancestor of element e_2 by verifying if $e_1.start < e_2.start < e_1.end$. We note that there exist other element-labeling schemes that support similar checks [8, 30]. The one we use is particularly suited for the techniques presented in Section 5.

We consider the subset of XPath consisting of tree pattern queries over single documents. A tree-pattern query is modeled as a tree where: (i) each node is labeled with a symbol from *label* $\cup\{*\}$, and (ii) each edge is labeled with "/" or "//" to denote the child or descendant axis respectively. A tree pattern node may carry a value condition of the form label=s or text contains s, where s is a string. Given a query q and an XML document d in the collection, q matches (p, d) if there exists a mapping from the nodes in q to elements in (p, d) that (i) preserves the parent/child and ancestor/descendent relationships, and (ii) satisfies the query predicates on labels and values. For a query q with *n* nodes, an *answer* is a tuple $(p, d, e_1, \dots e_n)$ identifying a document where a match was found, and the elements in this match. As an example, the answer for query //*[contains(.,'xml')]//title will consist of tuples (p, d, e_1, e_2), where e_2 is a title element and e_1 an ancestor in the same document that contains the keyword "xml".

Indexing and Query Processing We index element labels as well as words 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)

The $Term$ relation for each document can be easily constructed in one traversal of the document. We

henceforth use *posting* to refer to a tuple in $Term$. Accordingly, given a term a, we use "posting list" to refer to the set of postings for a and denote it as L_a.

The *Term* relation is distributed among the peers of the system using a *distributed hash table* (DHT for short) [14]. A DHT is a distributed data structure implementing a relation with a key. In brief, the interface of the DHT includes:

locate(k) returns the id of peer in charge of key k
put(k,α) enters a new posting for *k*
get(k) returns the postings for *k*
delete(k,α) delete a posting for *k*.

The DHT keeps the network peers connected, and distributes (quite evenly) the keys between the nodes of the network, typically using some hash function It also handles peers joining and leaving the network.

In KadoP, XML documents are stored at their publishing peer, whereas the $Term$ relation is stored in the DHT using terms as keys. Thus, $Term$ is split horizontally among peers, with peer p in charge of a portion $Term_p$ defined as follows: $Term_p = \{Term(p', d, sid, a) \mid \text{locate}(a) = p\}$. The posting lists in $Term_p$ are assumed to be 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.

Now, a query q submitted at peer p is processed as follows. For each term a in q (not a wildcard or a stop word), p asks the peer in charge of a for the posting list L_a, and subsequently performs a holistic twig join over all the received lists. For instance, consider again the query //*[contains(.,'xml')]//title. The index query uses the *Term* relation to find the *sid*s of all elements labeled title and all parents of text elements containing the word xml. Then, a join on these *sid* collections computes the *sid*s of the answers. The peers holding the corresponding documents are then asked to send their answers to the query peer.

In this example, the index query is *complete* in that no answer will be missed, and *precise* in the sense that, in the second phase, we only have to contact peers that contribute to the answer. For other queries, the index query may be incomplete or imprecise. For instance, for the query //a//*, the index query cannot rule out *sid*s of a elements with no descendent. This leads to an imprecise result, as the index returns a superset of the documents that contain answers to the query. For now, note that KadoP index queries are complete and precise in the absence of stop words and wildcards (*s).

Due to space constraints, several smaller indexing issues are omitted. For instance, (i) KadoP indexing distinguishes between labels and words; (ii) the *Term* relation also contains typing information (i.e, on the schemas of the documents); and (iii) other relations are indexed in KadoP, e.g. given a document's id (p, d), one can retrieve its actual uri and its type. The focus here is on (a portion of) the *Term* relation because this is the information that is the most critical from a query processing viewpoint and also because it permits describing the main techniques that we introduced in KadoP towards scaling.

Concerning index update, for the moment, a document modification is interpreted as deletion followed by insertion, since the structural indexing of a document is not very easily updated. We assume that peers are not too volatile, which is acceptable for instance in the case of the Edos distribution. The DHT replication protects the index entries against some peer failure. (For that, we had to re-engineer the DHT based on our BerkeleyDB store for acceptable performance).

3 Scaling KadoP

The KadoP system presented in [6] followed the general ideas that were presented in the previous section. It was implemented over PAST [35], a standard DHT overlay network. Our first experiments highlighted severe scaling limitations, primarily caused by the processing of long posting lists. Other DHT systems we are aware of, e.g., [3, 13, 22, 27], raise the same issues.

Motivated by the aforementioned issues, our goal is to develop techniques that enable the scalable distributed indexing of a large volume of XML data. In this direction, we identify three important metrics to assess the performance of a distributed index: (i) *indexing time*, (ii) *query response time*, (iii) *bandwidth consumption* (during index query processing since transferring the results to the query peer is unavoidable). Another interesting metric that we will mention is the *the time to the first answer*.

These metrics formed the criteria in the development of the techniques that we present in the following

sections. More concretely, Section 4 introduces a specialized index data structure and associated query processing algorithms aimed at reducing query response time and the time to the first answer. Section 5 describes a separate technique whose focus is on reducing the bandwidth consumption, and possibly also (ii). Finally, Section 6 presents an index organization scheme that reduces redundancy in the index, and can primarily improve (i) and (ii).

Before continuing with our presentation, we discuss some technical improvements we brought to the basic KadoP system that enhance system performance in terms of (i) and (ii). In the remainder of the paper, we will assume the modified system when we refer to KadoP.

Improving indexing time To index a document, the system constructs in one traversal the element postings (Section 2) and routes each posting, using the multi-hop routing algorithm of the DHT [34], 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 store system.

The original *insert* operation in a local DHT index is very inefficient. According to the standard DHT API [14], 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. This is quadratic in the number of index entries. To overcome this issue, we extend the DHT API described in Section 2 with a new operation, namely *append(key, entry)*, to obtain an indexing of linear cost.

Another measure to speedup indexing was the tuning of 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 concretely, we replace PAST's local index storage (based on XML *gzip*ed files) with a B+-tree storage provided by BerkeleyDB. Thus, $Term_p$ at peer p is organized as a clustered index, using term as the search key. Furthermore, the postings associated to a given term are lexicographically ordered by (p, d, sid).

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

Improving query response time For each term a in the query q to evaluate, the peer p in charge of the query asks the peer in charge of a for L_a. Observe that the only retrieval operation in the DHT API is *get*. This is 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 limits scalability. We therefore modified the DHT API (and the actual DHT system) by adding a *pipelined get* method, which transfers posting lists in pipeline. This simple modification brought important performance improvements.

KadoP implements a multi-threaded, block-based version of the holistic twig join from [10]. 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; in this case, 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.

4 DPP Algorithm

In practical applications, the distribution of posting list sizes is very skewed (a few terms are very frequent), leading to some very large posting lists. Managing and transmitting such lists is very costly, and essentially becomes the dominant cost factor in processing queries. (As previously explained, index queries are processed at the speed of the slowest posting list transfer.) 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.

4.1 Distributed Posting Partitioning

The distributed posting partitioning (DPP for short) is a a distributed hierarchical data structure for managing posting lists. The key idea is to split the posting list for a popular term horizontally based on range conditions, and to migrate portions to other peers. In this sense, DPP is similar in spirit to distributed B-trees [24].

Before we describe DPP in more detail, we introduce some necessary definitions. 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'.

We now discuss the precise DPP organization. Let L_a be a long posting list that is initially stored in peer p. In its simplest form, DPP partitions the list L_a in blocks $L_{a,1}, \ldots, L_{a,n}$ based on conditions $C_1, \ldots, C_n$ such that $C_i < C_{i+1}$. Each block $L_{a,i}$ is assigned the pseudo-key *overflow*:i:a and is routed to the peer in charge of that key in the DHT. Thus, the storage of L_a is distributed among several peers, and p stores only the conditions $C_1, \ldots, C_n$ and the corresponding pseudo-keys. (Observe that this information allows p to fully reconstruct L_a.) The key idea is that L_a can now be transferred in parallel to a query peer, and thus the total latency is reduced. Moreover, as we discuss later, the system can use the conditions to avoid joining blocks that will not generate matches.

This one-level partitioning can be generalized to multiple levels, which leads to the more general definition of DPP. In its most general form, DPP is a tree data structure consisting of internal blocks and leaf blocks. Each internal block comprises of conditions and corresponding "pointers" to other blocks, while a leaf block comprises 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:

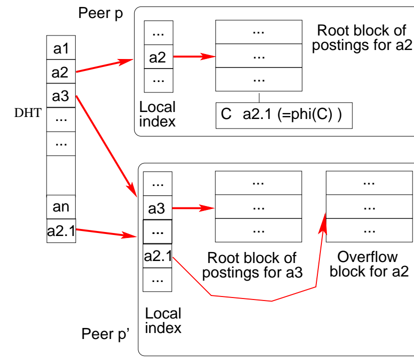

Figure 1. The organization of Term

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

A *DPP* consists of a *root* DPP block and of the whole tree obtained by following the φ pointers. The DPP allows distributing a posting list over several peers, thus the list can be fetched in parallel, reducing the query response time. Further, the DPP is a search structure, that is, the conditions attached to blocks allow focusing the search only on those blocks (thus peers) that may contain useful data. This may save in communication costs.

Let us now consider the implementation of DPP in KadoP. The implementation 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 tuples, 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. In practice, we found that the conditions are rather small. Thus, assuming reasonable block size (e.g., 1000 entries) for

both the data and condition blocks, posting lists of up to 10^6 entries require a single level of indirection. So, to simplify, we did not set a bound on the size of a condition block in KadoP, and implemented a two level DPP structure. If one block at the second level corresponding to some condition C overflows, it is split in two using two conditions, $C_1 \leq C_2$. One of the resulting blocks is moved to another peer. The root block replaces C_1 (and its pointer) with C_1, C_2 (and the corresponding pointers).

The DPP is an ordered search structure, thus the block splits respect data order. Alternatively, one could distribute a block's data randomly between subcontracting peers. This still allows for parallel transfers, but block conditions do no longer guide the search, and the lists have to be merged at the receiver. When tested, this approach brought performance improvements a few times smaller than the order-based DPP. Thus, we do not consider it further.

To conclude this section, we note that, in the actual system, all documents have an associated (user-specified or system-inferred) type, and terms appearing in a document are associated to its type. This type information is also stored in the conditions of the DPP blocks, which allows the system to filter posting blocks further based on the type constraints of the query. Thus, the system will avoid transferring posting lists of a term with a given type, if the other query terms do not match that type.

4.2 Query processing in DPP

We now describe a modification to the holistic twig join algorithm that can exploit the parallel transfer opportunities provided by the DPP.

We first explain the technique for an index query q' joining two long posting lists. The DPP splits the first list into blocks corresponding to $C_1, ..., C_n$ and similarly for the second list with conditions $C'_1, ..., C'_m$. For each pair (i, j), we have to perform the join of the blocks C_i and C'_j. The idea is to parallelize the join of blocks (C_i, C'_j) and that of some other (C_k, C'_l). In our system, the maximum degree of parallelism K is set in advance. When processing a query, the first K blocks for each posting are fetched in parallel, e.g., $C_1, ..., C_K$ and $C'_1, ..., C'_K$. The meaningful joins are computed in parallel and start producing answers. To shorten the time to the first answer, we relax the constraint that results be produced in lexicographical order, and return to the user the first results produced by each join. When an active block in some posting list completes, the next block in this list is activated and so on. Observe that if the conditions do not intersect, the answer of the join is empty.

Consider now an arbitrary query q' with n nodes. The join algorithm works similarly except that instead of joining pairs of conditions, we now have to join vectors $(C^1_{i_1}, ..., C^n_{i_n})$, one for each node of query q'. An issue is whether the algorithm is generating a too large number of joins. The answer is no, because the postings are lexicographically ordered. Indeed, suppose we are joining n postings, consisting respectively of $m_1, ..., m_n$ blocks. Then one can prove that we don't have to consider $m_1 \times ... \times m_n$ (as one might have expected in the worst case) but at most $m_1 + ... + m_n$ joins. In practice, it is often the case that even much fewer joins have to be considered.

Note that we only have to fetch blocks that have a chance to provide matches. For instance, if some block, say C^k_5 does not intersect with C^l_j, for any j, we don't need to fetch it. The filtering of non-matching blocks is performed as follows. Consider the i-th posting list with conditions $C^i_1 \leq ..., \leq C^i_{K(i)}$. We compute the minimum Id of a document that may satisfy C^i_1, say min_i and the maximum Id of a document that may satisfy $C^i_{K(i)}$, say max_i. We know that all answers will be (with some abuse of language) between min = $Max\{min_i \mid i \in [1.n]\}$ and max = $Min\{max_i \mid i \in [1.n]\}$. We do not have to fetch any block that does not intersect this interval. Furthermore, instead of transferring a block C^i_j that intersects, it suffices to transfer its intersection with $[min, max]$. Observe that by approximating $C^i_i \vee ... \vee C^i_{K(i)}$ by the document interval $[min_i, max_i]$, we may let through some blocks that have no possible match. Since we partition a posting list in blocks of equal size, the gaps between consecutive intervals are small on average and this does not happen often.

To conclude this section, we note that other well-known distributed query optimization techniques [31] could clearly apply. For instance, some structural joins could be pushed to the peer holding the longest posting list involved in the query, thus reducing data transfers. Replication is another source of potential optimiza-

tions, as the transfer of a posting list can be optimized by replicating it and transferring fragments from different copies. DPP blocks can also be replicated to several peers based on their popularity (i.e. how often they are accessed) and their position in the DPP (the root is always accessed, thus it should be replicated more). The DHT does replicate its index for reliability. However, this replication does not fit our needs, because the replication factor is *fixed* and has to be set prior to creating the network. In contrast, we would need to control the replication degree on a block-by-block basis.

4.3. 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 (9 sites in France) for wide-area distributed applications. 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). We have used the Aug. 2006 version of the 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. When multiple peers publish, the data set is split evenly among them.

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 3, since they brought significant improvements.

We first verified the belief that long postings are frequent and important in size in standard XML collections. 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.

Indexing time. Figure 2 reports the indexing time for several copies of DBLP. We varied the size of the KadoP network, and also varied the number of peers that index data, the *publishers*. The horizontal axis

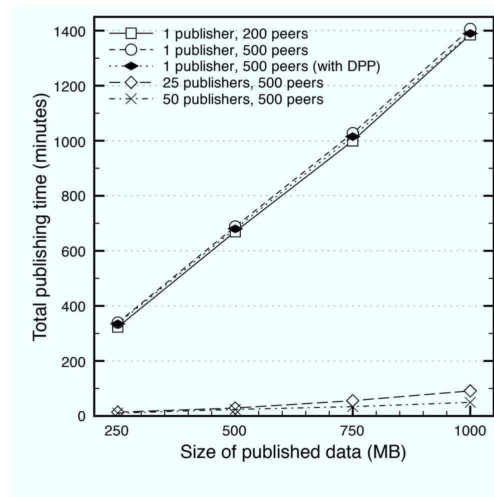

Figure 2. Indexing time.

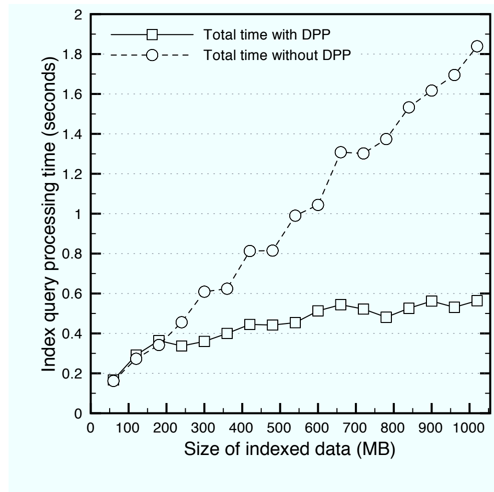

Figure 3. Query response time.

traces the *total* size of published data (over all peers). 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 study the processing of long postings, in this case author

(a stress test for our approach). 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. In another experiment, we studied traffic consumption. We run simultaneously many data intensive queries. More precisely, we run a workload of 50 queries, each of which involves at least one term with a long posting list. In an interval of 5 minutes, the 50 queries are randomly submitted to execution at 50 distinct nodes, generating a medium throughput of one query each 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 XML data indexed in DHT, is 32MB, 66MB, 95MB, and 127MB respectively. In conclusion, the traffic was not a problem for the 10GB network. However, since the traffic increases linearly with the size of the indexed data, it necessarily becomes a bottleneck, which motivated the work on Bloom Filter of 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.

5 Structural Bloom Filters

We introduce a mechanism, termed Structural Bloom Filters, for reducing the volume of transferred data in query evaluation. As the name suggests, 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.

To illustrate the proposed mechanism, assume that the simple query //a//b is initiated at peer p. Peer $locate(a)$ computes an *Ancestor Bloom Filter* (AB Filter for short) of the posting list L_a, denoted as $ABF(a)$, and sends it to Peer $locate(b)$. Peer $locate(b)$ filters the posting list of b based on $ABF(a)$, and computes a set B' of b postings that is a superset of $b[\backslash\backslash a]$, that is, the b postings having an a ancestor. At this point, Peer $locate(b)$ can send B' to the peer that submitted the query without compromising the recall of the result. Depending on the characteristics of the filter and the data, we expect the size of $ABF(a)$ to be much smaller than the size of the complete a postings and the size of B' to be much smaller than the size of the complete b postings. Thus, we expect important savings in data transfer at a moderate cost in terms of local processing time. As we will see, similarly, Peer $locate(b)$ may compute a *Descendant Bloom Filter* (DB Filter for short) $DBF(b)$ of b and send it to Peer $locate(a)$, so that the latter can select a superset of $a[//b]$, the a postings that have a b descendant.

[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

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

In what follows, we describe in detail AB and DB Filters and introduce strategies that integrate them in query processing. 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 each 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 [9] 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 $1 \le i \le k$. Similarly, a membership query for an element a 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 on an element not in S may return a *false positive* answer due to the existence of 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 precisely based on $|S|$ and the parameters n and k. For a given set S and a given false positive rate fp, 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 size is typically much smaller than the size of the set that it encodes, so its transmission costs much less than that of the original set. The trade-off, of course, is the introduction of false positive errors when the filter is used for membership queries.

5.1 Ancestor Bloom Filters

The AB Filter for a, denoted $ABF(a)$, is used to filter the posting list of b to obtain a sublist of $b[\backslash\backslash a]$, denoted $F(b, ABF(a))$. (Recall that $b[\backslash\backslash a]$ comprises the b postings having an a ancestor.) 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 $I \subseteq I'$. This suggests the following generalization to postings. The *cover* of a posting $e_a = (p_a, d_a, start_a : end_a : lev_a)$ is $\mathcal{D}(e_a) = \{(p_a, d_a, I) \mid I \in \mathcal{D}[start_a, end_a]\}$; 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 essence of the Ancestor Bloom Filter technique comes from the following theorem:

Theorem 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 $\mathcal{D}(L_a)$.*

The AB Filter encodes the set $\mathcal{D}(L_a)$ using a Basic Bloom Filter. The parameters of the Basic Bloom Filter can be determined based on $|\mathcal{D}(L_a)|$, which can be maintained incrementally at peer $locate(a)$ as new postings are inserted. The AB Filter also records an integer $dclev$ that denotes the highest level $j \le dclev$ such that an interval I_{ij} appears in $\mathcal{D}(L_a)$. As we discuss next, this is used to implement efficiently the probing mechanism.

Let $e_b = (p_b, d_b, start_b : end_b : lev_b)$ be in L_b. To check whether it belongs to $b[\backslash\backslash a]$, we compute the cover $\mathcal{B} = \mathcal{D}(e_b)$ and then process each $(p_b : d_b : I)$ in $\mathcal{B}$ as follows. We compute the dyadic container $\mathcal{D}_c(I)$, and test whether $(p_b : d_b : I')$ is in the Bloom Filter for each I' in $\mathcal{D}_c(I)$. 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 Theorem 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.

It is interesting to note that we can realize the AB Filter using a simpler containment condition, one that employs only the start value $start_b$. More concretely, note that the condition $start_a < start_b < end_a$ is sufficient to ensure that e_b is a descendant of e_a, since posting intervals cannot be partially contained. We can thus determine if e_b is in $b[\backslash\backslash a]$ by checking whether $(p_b : d_b : [start_b, start_b])$ is covered by an interval in $\mathcal{D}(//a)$. Clearly, this simpler approach is equivalent to Theorem 1 when $|\mathcal{D}(e_b)| = 1$. As we show later, however, the conjunction of containment conditions in Theorem 1 leads to an error probability that is exponential in $|\mathcal{D}(e_b)|$. Hence, while the two approaches have the same expected performance when $|\mathcal{D}(e_b)| = 1$, Theorem 1 leads to lower error probability in the general case.

Space overhead In a worst case scenario, every cover $\mathcal{D}(e_a)$ contains $2l$ intervals and the number of in-

Data set	Element count	$\lvert\mathcal{D}(e)\rvert$	$2l$
IMDB	100K	1.37	32
XMark	200K	1.50	34
Swiss Prot	3.2M	1.29	42
NASA	500K	1.55	38
DBLP	1.5M	1.23	40

Table 1. Average size of the dyadic cover.

sertions in the Bloom Filter thus grows asymptotically with $2l$. In practice, however, we expect a much lower number of elements. A main reason for that is that the number of dyadic intervals in the cover of $[x, y]$ essentially depends on the width $y - x + 1$. Given that XML documents are typically small and bushy, this implies that the average width of element is very short, so on average, an a posting is likely to be covered by a small number of dyadic intervals.

To verify this conjecture, we have performed a simple experiment on real-life and synthetic data sets. For each data set, we generated the start/end encoding of its elements and then measured the size of the dyadic cover for each element. The results are shown in Table 1. These results validate our assumption for practical data sets and demonstrate that the space overhead of $ABF(a)$ is likely to be much lower than the worst-case bound of $2l$.

Tracing Wide Intervals Observe that the AB Filter returns a false positive for a posting e_b if each interval $(p_b, d_b, I) \in \mathcal{D}(e_b)$ is covered by some $(p_a, d_a, I') \in \mathcal{D}(L_a)$. Now suppose that there is a collision in the Bloom Filter between some e_a and some (p_b, d_b, K). The larger the size of K, the more false positive this collision may generate. At the limit, a collision with $(p_b, d_b, [1, 2^l])$ will generate a false positive with any b-element in document (p_b, d_b). This connection between the size of an interval and its potential damaging effect on the false positive rate suggests the idea of using more "traces" at higher levels to boost the accuracy of the filter.

More precisely, we assume the existence of a function $\psi : [0, l] \rightarrow [1, \infty)$ that assigns a positive number to each level. In essence, $\psi(j)$ specifies the number of replicas (or, traces) that are inserted per interval I_{ij} at the same level j. Accordingly, it specifies that a look-up of I_{ij} is implemented as the conjunction of $\psi(j)$ look-ups, one for each trace. Thus, an increased $\psi(j)$ means that probes at level j are less likely to return a false positive. We notice that this mechanism encompasses two counter-forces: while an increased $\psi(j)$ implies better accuracy at level j, it also implies a larger number of insertions which can hurt the false positive rate. We revisit this issue later, and describe our choice of ψ, when we analyze the false positive probability of an AB Filter.

Analysis of error probability 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. Let e_b be a L_b posting not in $b[\backslash\backslash a]$ and let k be the number of elements in $\mathcal{D}(e_b)$ that are not covered by some element in $\mathcal{D}(L_a)$. The posting is falsely selected by the AB Filter if each of the k not covered elements is covered by a false positive answer of the Bloom Filter. Consider the containment check for a single not covered interval (p_b, d_b, I). In the worst case, the set $\mathcal{D}_c(p_b, d_b, I)$ will contain $l + 1$ elements (if $s_b = e_b$) and there will be $l + 1$ probes to the Bloom Filter. Note also that a probe at level j will translate to $\psi(j)$ probes, one for each replica. Thus, the probability that (p_b, d_b, I) is correctly not covered by any interval is not smaller than $\prod_{0 \leq j \leq l}(1 - fp[\psi])^{\psi(j)}$. In turn, this implies that the probability that (p_b, d_b, I) is wrongly covered is bounded by $1 - \prod_{0 \leq j \leq l}(1 - fp[\psi])^{\psi(j)}$. Since the Bloom Filter returns a positive answer if *every* uncovered interval (p_b, d_b, I) is covered, the probability that the filter falsely selects e_b is bounded by $(1 - \prod_{0 \leq j \leq l}(1 - fp[\psi])^{\psi(j)})^k$. The worst case occurs when $k = 1$ and we can thus bound the overall false negative rate as follows:

$$fp^A \leq 1 - \prod_{0 \leq j \leq l} (1 - fp[\psi])^{\psi(j)}$$

Observe that function ψ encodes an interesting trade-off: as we increase $\psi(j)$ at a level we increase the number of insertions and thus $fp[\psi]$, but we also increase the number of probes and thus decrease $fp[\psi]^{\psi(j)}$. Given that the upper levels are not likely to

generate a lot of insertions and that (as already mentioned) they are potentially very damaging in terms of false positive, it is desirable to set $\psi(j)$ high for high levels.

In our work, we consider the function $\psi[j] = \lceil 1 + j/c \rceil$ (for some integer $c \geq 1$) that essentially adds one extra trace every c levels. This choice is driven by the following heuristic. Consider an interval I_{ij} that is not present in $\mathcal{D}(L_a)$. If I_{ij} is a false positive in the Basic Bloom Filter, then the effect on accuracy is intuitively captured by the width 2^j. We can thus use $2^j fp[\psi]^{\psi(j)}$ as a measure of the expected effect, where $fp[\psi]^{\psi(j)}$ is precisely the probability that I_{ij} is a false positive. We can show that the particular ψ function ensures the following "balancing" property if $fp[\psi] < 1/2^c$: every interval I_{ij} has the same expected effect and it is bounded by $1/2^c$. In our experiments, we set $c = 4$ as we expect the basic false positive rate to be less than $1/16$. We will see in Section 5.4 that this scheme yields good performance in practice.

5.2 Descendant Bloom Filters

A *Descendant Bloom Filter* for b, denoted $DBF(b)$, is used to filter the postings in L_a to obtain a sublist of $//a[//b]$, denoted $F(a, DBF(b))$. The key idea remains essentially the same as $ABF(a)$ but reversed: we send in a Bloom Filter traces of b postings (using again dyadic intervals), and then perform some tests for the a postings. The crux of the DB Filter is given by the following theorem:

Theorem 2 *For each $e_a \in L_a$, $e_a \in a[//b]$ iff $\mathcal{D}(e_a) \cap \mathcal{D}_c(L_b) \neq \emptyset$.*

The filter is created by inserting in the Bloom Filter, for each posting e_b in L_b, each element in $\mathcal{D}_c(e_b)$. For a posting e_a, it suffices to perform a look-up in the filter for each element in $\mathcal{D}(e_a)$. Observe that each b posting typically entails many more insertions in the Bloom than with the AB, i.e., $\mathcal{D}_c(e_b)$ vs. $\mathcal{D}(e_a)$. 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 revisit this point in Section 5.4 when we present a comparison between the two techniques.

Due to space constraints and since the technique is very similar to that of the AB Filter, we will not discuss

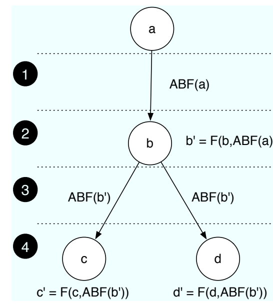

Figure 5. AB Reducer

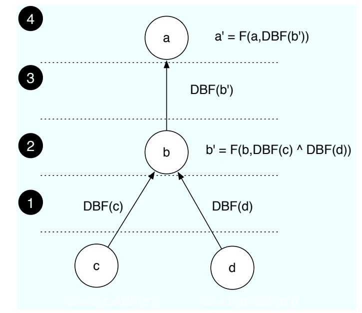

Figure 6. DB Reducer

any further the DB Filter. Note that, in particular, an analysis along the lines of that presented in Section 5.1 can also be obtained.

5.3 Query Evaluation with Bloom Filters

We introduce three query processing strategies based on Structural Bloom Filters: *Ancestor Bloom Reducer*, *Descendant Bloom Reducer*, and *Bloom Reducer* that can be seen as a hybrid of the previous two. In the interest of space, we present the strategies by example using the twig query //a//b[//c][//d]. For ease of exposition, we assume that each tag is handled by a distinct peer and use the name of the tag to refer to the corresponding peer.

All strategies proceed in two phases: in the first phase, the peers exchange 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. 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.

The intuition behind each strategy is to perform some Bloom Filter-based pre-processing to focus the query to a (hopefully) small set of documents and peers. The hope is that the savings in reduced posting list transfers offset the relatively small overhead of transmitting compact Bloom Filters. Of course, this trade-off depends heavily on the data and query characteristics, and the use of Structural Bloom Filters may not always decrease the total network traffic. We examine this point in more detail in Section 5.4 where we evaluate experimentally the performance of these strategies.

5.4. 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 Section 4. 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 ofarticle 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. AB Reducer is in fact costlier than the baseline no-filter strategy, as it also transfers the unfiltered article list.

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

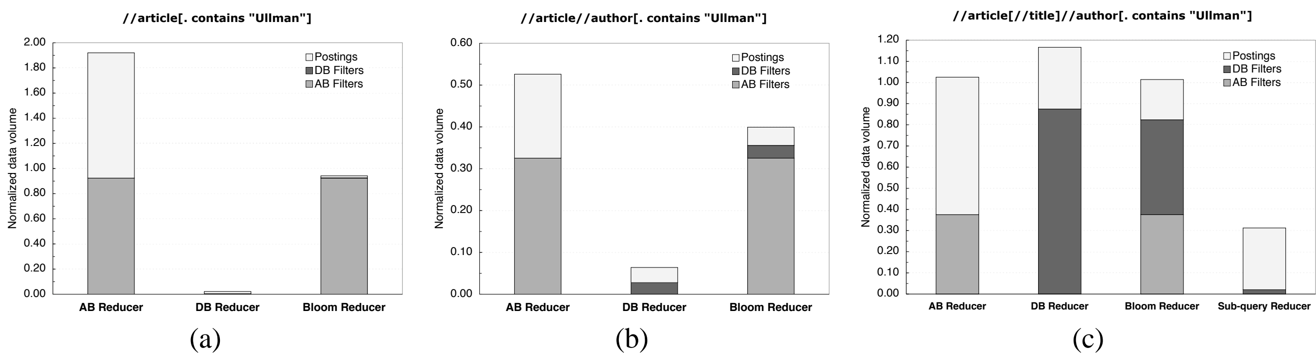

Figure 7. Performance of Bloom-based strategies for different queries

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.

```
⟨?xml version="1.0" encoding="UTF-8"?⟩
⟨!DOCTYPE document [
⟨!ENTITY thisabstract SYSTEM "2445abstract.xml"⟩
⟨!ENTITY paper SYSTEM "2445paper.xml"⟩
⟨!ENTITY md SYSTEM "MartinDoe.xml"⟩
⟨!ENTITY dj SYSTEM "DanJones.xml"⟩ ]⟩
⟨article⟩
  ⟨author name="Jones"⟩ &dj; ⟨/author⟩
  ⟨author name="Doe"⟩ &md; ⟨/author⟩
  ⟨title⟩More on XML⟨/title⟩
  ⟨abstract⟩ &thisabstract ⟨/abstract⟩ &paper
⟨/article⟩
```

Figure 8. Intensional data using includes

(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.

6 The Fundex

We first present the motivation for intensional data. We then consider a technique that we call *Fundex* to index and query intensional data. We then shortly describe some experiments.

Motivation The XML standard proposes two important features for managing intensional information, namely includes (using the entity keyword) and references (using id, idref). Figure 8 shows an example of includes with a bibliography document. Observe

for instance that the *abstract* is kept in a separate file. This classical idea of storing data in separate files is typically very convenient for maintenance purposes. There are differences between includes and references, e.g., in the way queries are stated, but from a functional viewpoint (e.g., in a user interface) and from an indexing viewpoint, this comes to the general idea of having portions of the data being *intensional* instead of extensional.

Consider again the document d in Figure 8 and the query: Retrieve the bibliography references containing the word graph in the abstract. We have two alternatives: (*naive*) do not return d because its abstract element does not contain the word graph; or (*brutal*) return d and similarly return any document including intensional data since this intensional data possibly contains the desired pattern. The *naive* alternative corresponds to simply indexing the documents as they are, and is very incomplete. The *brutal* one may be easily implemented and is very imprecise. It may result in contacting (almost) all peers for each query. We will show that it is possible to be complete at a lower cost using an indexing technique called the *Fundex* (for functional indexing).

For simplicity, assume that intensional data is obtained by function calls (this also covers includes and referencing). The indexing is modified as follows:

- The element identifier specifies in addition whether the subtree rooted at that node is purely extensional or whether it contains some intensional data, i.e., it is an *intensional-node*.

- Let $w = f(u)$ be the string corresponding to a function call occurring in some document. The peer p in charge of this function call is the peer in charge of the key *fun:w* (so anyone can find who it is). The identifier of the result of this call is $h'(w)$ for some hash function h'. The *functional id* (in short, *fid*) $(p, h'(w))$ plays the role of the pair (p, d) for regular documents. When first asked to index $f(u)$, Peer p materializes $f(u)$ and indexes it using $(p, h'(w))$ in place of the standard docid. Once indexed, the result $f(u)$ does not have to be kept. Suppose another peer encounters another occurrence of the same function-call, and requests from p the indexing of $f(u)$. Then p has nothing to do.

- We treat an occurrence of a call to $f(u)$ as a pointer to the virtual document $(p, h'(w))$. We use a relation Rev to capture the *reverse* pointers that provides all occurrences of a particular function call. This relation is also supported by the DHT and given a pair $(p, h'(w))$, we can use the DHT to obtain the structural identifiers of all occurrence of the corresponding function call.

We now discuss the integration of these features in query processing. To simplify exposition, we will illustrate the techniques on the previous example query //article//abstract[.contains "graph"]. Query processing utilizes the same twig join algorithm as before, except that a match may be marked as incomplete if the corresponding element is intensional. Returning to the example, this will occur when the abstract of an article includes a reference to another document (see also Figure 8). In this case, the algorithm will generate an answer tuple $t = (e_1, e_2?)$, where e_1 is the id of the article element, e_2 is the id of the abstract element, and the "?" denotes an incomplete match since the abstract is intensional. We refer to t as a potential answer and use R to refer to the set of such tuples. We also say that abstract is an incomplete query variable in R. In parallel to the main twig join algorithm, the system evaluates the predicate contains "graph" on the functional documents (i.e., documents appearing in references), and computes the set S_a comprising the ids of documents that satisfy the predicate. Using Rev, S_a is mapped in turn to a set R_a of element ids that contain references to the matching functional ids. Having computed R and R_a, the incomplete answers in R can be completed by performing a θ-join between R and R_a.

It is straightforward to generalize this approach to the case where a potential answer contains several incomplete variables. The intuition is that the system has to compute several relations R_i, one for each incomplete variable, and then perform a multi-way join between R and relations R_i to complete the answers. We note that it is possible to optimize the final join by grouping R on incomplete variables and then joining each group with the corresponding subset of the relations R_i.

Clearly, the new processing algorithm results in improving completeness if compared to *naive*; or preci-

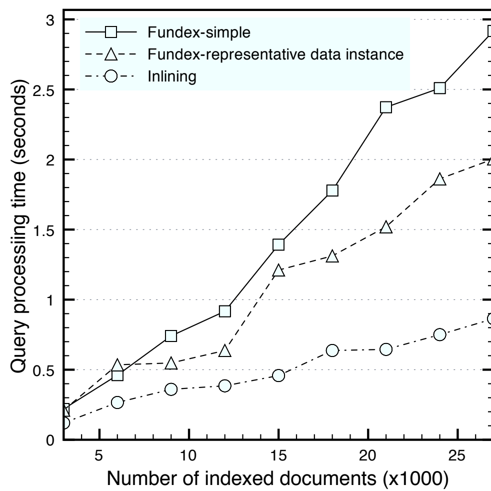

Figure 9. Query processing times with the Fundex.

sion if compared to *brutal*. We have tested the Fundex algorithm with the INEX [38] HCO collection, consisting of 28000 documents. Each document contains a description of a publication, and a reference to an abstract kept in a separate file. This makes a total of 56000 small XML documents, each roughly 1KB. We illustrate these tests with the query: //article[contains(.//title,'system') and contains (.//abstract, 'interface') The posting lists for title, article and abstract have at least 28000 entries. The words system and interface are reasonably frequent, but there are very few actual query matches, precisely 10. Figure 9 shows query evaluation time on a collection of increasing size. The figure also shows the impact of two other optimizations, namely in-lining and representative data indexing. We briefly discuss them next.

In-lining In-lining consists in actually inserting some intensional data in the document *from the indexing viewpoint*. So for instance, for the example of Figure 8, the includes are replaced by the actual files before indexing the document. As Figure 9 indicates, in-lining results in important savings. This also reflects measures the cost of following backward pointers. When references are used to split (and possibly distribute) an XML tree as in [10, 11], in-lining does not generate any extra cost and works nicely. When the data resembles a graph, i.e. the same portion of a document is referenced several times, in-lining potentially generates a lot of extra indexing cost.

Representative-data-indexing The full indexing of intensional data may be quite expensive, suggesting some less precise or less complete indexing. To illustrate, we briefly outline a technique of particular interest, based on lossy indexing of intensional data. For instance, consider the intensional paper in Figure 8. We may have some information about its type as provided, e.g., by an XML schema or a DTD. In the spirit of the representative objects of [28], we can make the index aware of such typing information using the notion of *representative-data-indexing*. Think of the document d' obtained by replacing in d intensional data by a skeleton of the data they provide, i.e., by a representative instance. Instead of indexing d, we index d'. Now, omitting details, we can answer queries more precisely than *naive* if not as completely as with *in-lining*. For instance, consider the query: //article//contains(.//section-title,'Graph') and the document d in Figure 8. The tag article is matched by d, the tag section-title is matched by a node e of the representive-data-indexing for d. And because e is only representative, the conditions underneath are ignored.

In experiments, we found that the representative-data-indexing technique brings important savings. It does that by pruning the search space: a number of backward pointer chasing is avoided because the "type" does not match.

7 Related Work

Many works have studied P2P keyword search, e.g., [25, 33]. The transfer of long posting lists is highlighted as an important problem in DHT networks in [25].

We share motivations with a number of works on P2P data-sharing platforms, [20]. Unlike our system, [7, 19, 29] are based on unstructured networks. A large emphasis on scaling is also considered in [29], but in the context of the relational model. They often rely on multicasting queries to all the peers, which we try to avoid in KadoP. Like us, [17] deals with XML but their indexing is based on paths (queries including text keywords cannot be processed based on the index). Iris-

Net [18] supports distributed hierarchical queries over the Internet. Like KadoP, IrisNet uses XML (trees) as a data model. However, it is very different from KadoP in that it builds on a hierarchical overlay network (a DNS) to route queries and data.

The idea to use a streaming algebra in such a context is very natural and can be found, e.g., in [2, 20]. The DPP structure is based on the idea of distributing a tree over a network. This idea has been used to support range queries in a DHT in BATON and in PTrees [23, 26]. Using BATON instead of PAST would enable KadoP to process index queries with range conditions, too.

The twig join algorithm that we use is an extension of [10]. The idea of distributing portions of documents in other documents (an aspect of what we call intensional data) and the problem of evaluation queries over such distributed documents is not new, see, e.g., [4, 32, 36]. The most obvious difference is that KadoP stresses indexing over a structured P2P network to focus on the relevant data query, while the mentioned works execute queries using the local information available at one site (peer) and route subqueries to the neighbors sites, following the link between documents. [11] studies the same problem for a single XML tree distributed horizontally and vertically over sites. However, they use a bottom-up algorithm while ours is top-down in the spirit of [10].

The proposed structural bloom filters are inspired by the use of basic bloom filters for processing distributed relational joins [31]. Clearly, the details of our mechanism are more involved, as we are dealing with structural joins over hierarchical data. We note that the underlying framework of dyadic interval decomposition has been used in several applications, such as, the estimation of quantiles over streams [1], approximating the selectivity of spatial joins [15], or evaluating stabbing queries [37]. 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.

8 Conclusions

The work presented here is motivated by the scaling of XML indexing and query processing in a P2P context, and in particular by the Edos application. The system has been extensively tested and measured on a large number of peers. The Edos platform based on KadoP is now almost complete and we are planning its deployment.

We are currently working on improvements of the KadoP system. We have started building a query optimizer able to explore other strategies, such as Bloom-based strategies, and standard distributed database optimization techniques, such as load balancing. Also, the index construction for a large collection of documents (such as a new Linux release in Edos) places a lot of stress on the system. We are exploring techniques to improve index construction time. Finally, we are working to improve the system's existing support for peers joining and leaving the network.

To reduce the index size, one could also consider indexing at a coarser level of details, e.g., record only the document and peer ids in the index. This strongly reduces the index, but renders index queries imprecise. An interesting middle ground is, instead of indexing the document, to index a representative instance [28] in the spirit of [17]. We may also consider indexing words (not tags) selectively, e.g. index the words in abstracts but not in the bodies of papers. We believe that carefully trading precision and completeness for performance is a fascinating avenue of research.

Acknowledgments We thank Nitin Gupta and Gabriel Vasile for participating to implementing KadoP. We thank Sophie Cluet and Karl Schnaitter for discussions on P2P XML indexing. We thank Grid5000, the wide distributed infrastructure, for the support provided in testing KadoP.

References

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

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2), 2003.

[3] K. Aberer, A. Datta, M. Hauswirth, and R. Schmidt. Indexing data-oriented overlay networks. In *VLDB*, 2005.

[4] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *SIGMOD*, pages 527–538, 2003.

[5] S. Abiteboul, I. Manolescu, and N. Preda. Constructing and querying peer-to-peer warehouses of XML resources. In *SWDB*, 2004.
[6] S. Abiteboul, I. Manolescu, and N. Preda. Constructing and querying P2P warehouses of XML resources. In *ICDE (demo)*, 2005.
[7] P. Adjiman, P. Chatalic, F. Goasdoué, M. Rousset, and L. Simon. Distributed reasoning in a peer-to-peer setting. In *ECAI*, 2004.
[8] S. Al-Khalifa, H. Jagadish, J. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.
[9] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.
[10] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, 2002.
[11] P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis. Using partial evaluation in distributed query evaluation. In *VLDB*, 2006.
[12] Z. Chen, H. Jagadish, L. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of XQuery. In *VLDB*, 2003.
[13] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. An indexing framework for peer-to-peer systems. In *SIGMOD Conference*, pages 939–940, 2004.
[14] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured P2P overlays. In *Proc. of IPTPS*, 2003.
[15] A. Das, J. Gehrke, and M. Riedewald. Approximation techniques for spatial data. In *SIGMOD*, 2004.
[16] Edos project: Environment for the development and distribution of open source software. http://www.edos-project.org.
[17] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt. Locating data sources in large distributed systems. In *VLDB*, 2003.
[18] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. An architecture for a world-wide sensor web. *IEEE Pervasive Computing*, 2(4), October-December 2003.
[19] 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.*, 16(7), 2004.
[20] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based peer-to-peer networks. In *Proc. of IPTPS*, 2002.
[21] 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.
[22] H. Jagadish, B. Ooi, K. Tan, Q. H. Vu, and R. Zhang. Speeding up search in peer-to-peer networks with a multi-way tree structure. In *SIGMOD*, 2006.
[23] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: a balanced tree structure for peer-to-peer networks. In *VLDB*. VLDB Endowment, 2005.
[24] P. Krishna and T. Johnson. Index replication in a distributed B-tree. In *COMAD*, 1994.
[25] 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.
[26] P. Linga, A. Crainiceanu, J. Gehrke, and J. Shanmugasundaram. Guaranteeing correctness and availability in p2p range indices. In *SIGMOD Conference*, pages 323–334, 2005.
[27] B. Loo, J. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica. Enhancing P2P file-sharing with an internet-scale query processor. In *VLDB*, 2004.
[28] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative objects: Concise representations of semistructured, hierarchical data. In *ICDE*, 1997.
[29] W. S. Ng, B. C. Ooi, K. Tan, and A. Zhou. Peerdb: A P2P-based system for distributed data sharing. In *ICDE*, 2003.
[30] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-friendly XML node labels. In *SIGMOD*, 2004.
[31] M. T. Ozsu and P. Valduriez. *Principles of distributed database systems*. Prentice-Hall, 1991.
[32] V. Papadimos and D. Maier. Distributed queries without distributed state. In *WebDB*, 2002.
[33] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Int.'l Middleware Conf.*, 2003.
[34] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Int.'l Middleware Conf.*, 2001.
[35] A. I. T. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage. In *SOSP*, 2001.
[36] D. Suciu. Distributed query evaluation on semistructured data. *ACM TODS.*, 27(1), 2002.
[37] K.-L. Wu, S.-K. Chen, and P. S. Yu. Interval query indexing for efficient stream processing. In *ACM CIKM*, 2004.
[38] INitiative for the Evaluation of XML Retrieval. http://inex.is.informatik.uni-duisburg.de/.