FliX: A Flexible Framework for Indexing Complex XML Document Collections

Ralf Schenkel

Max-Planck-Institut für Informatik
Saarbrücken, Germany
`http://www.mpi-sb.mpg.de/units/ag5/`
`schenkel@mpi-sb.mpg.de`

Abstract. While there are many proposals for path indexes on XML documents, none of them is perfectly suited for indexing large-scale collections of interlinked XML documents. Existing strategies lack support for intra- or inter-document links, require large amounts of time to build or space to store the index, or cannot efficiently answer connection queries. This paper presents the *FliX* framework for connection indexing that supports large, heterogeneous document collections with many links, using the existing path indexes as building blocks. We introduce some example configurations of the framework that are appropriate for many important application scenarios. Experiments show the feasibility of our approach.

1 Introduction

1.1 XML on the Web

Some years ago, XML documents have mostly been used for exchanging data between different applications, hence the complete information was contained in a single document. Nowadays, as XML is increasingly used as a replacement for HTML on the Web or intranets, documents usually have XLinks or XPointers to data in other documents. In addition to such inter-document links, the XML standard allows intra-document links between elements of a single document (e.g., using attributes of type `id` and `idref`, or using an XLink).

While this increases the expressiveness of XML in intranets, in digital libraries, and on the Web, it is on the other hand one of the big challenges for XML retrieval. With information spread over several linked documents, an XML search engine should treat elements that are referenced through links similarly to "normal" child elements when evaluating path expressions in queries, which is typically done using some path indexing technique. However, two problems arise when taking links into account: (1) Links change the structure of XML documents, so they are no longer trees, but form a directed graph; (2) links generate interconnections of previously unconnected XML documents, yielding large sets of connected elements with long paths between them. While the latter problem can lead to path indexes that grow extremely large and take very long to build, the former even renders some of the established and highly efficient path indexes unusable. The framework that we are presenting in this paper can cope efficiently with large, heterogeneous collections of linked documents.

The second major challenge for XML retrieval is raised by the heterogeneity of data on the Web, but also in heterogeneous collections of data from different sources. As there is no universal standard for representing data in XML (and it is unlikely that there will ever be such a standard), schemas used to represent data widely vary across different data sources, and some do not provide a schema at all. Existing query languages for XML like XPath and XQuery are no longer appropriate for searching in such an environment as they cannot cope with the diversity of data. As an example, consider the query

```
/movie[title="Matrix: Revolutions"]/actor/movie
```

(i.e., find movies whose actors were also in the cast of "Matrix: Revolutions") may not give any result at all: One of the data sources may use the tag name `science-fiction` instead of `movie`, it may reflect the title of the movie as "Matrix 3", or the path between the elements may be longer than 1. To find as many relevant elements as possible (i.e., to get good recall), the system should automatically relax the query, which may yield the following extended query:

```
//∼movie[∼title∼"Matrix: Revolutions"]//∼actor//∼movie
```

This extended query has semantic vagueness for content and tag names (using the similarity operator $\sim$ as in the XML search language XXL [21]) as well as structural vagueness (by relaxing all `child` axes to `descendants-or-self` axes). Here, the $\sim$ operator means that elements whose tag names are semantically similar to the name in the query qualify for the result, too, but possibly with a lower relevance that is derived from the degree of semantic similarity. In the XXL search engine [17, 22, 23], similar words as well as similarity scores for them are extracted from an ontology, which can either be a general-purpose one like WordNet or an ontology specific to the topic of the query. As an example, an ontology for movies could state that `science-fiction` is a special case of a movie, so these tags have a high similarity score for the example query. We could also consider IMDB's [1] list of alternative movie titles which would allow us to conclude that a movie with title "Matrix 3" is a valid result to the first part of our example query.

Adding structural vagueness is motivated by the following example: A query like `movie/actor` can only be an approximation of what the user really wants, because the user cannot know the exact structure of the data. We therefore consider not only children as matches, but also descendants; the relevance of a result decreases with increasing path length. As an example, the relevance of a match `movie/cast/actor` could be 0.8, whereas the relevance of a match `movie/follows/movie/cast/actor` could be 0.2 (note that this result would not be relevant for the query). It is evident that evaluating queries with such structural vagueness requires the efficient evaluation of path expressions with the `descendants-or-self` axis, which is a weakness of most path indexes for linked XML documents. The FliX framework that we present in this paper is specifically optimized for this setting.

Beyond such simple structural vagueness, one can think of more elaborated approaches. As an example, paths that include at least one link traversal could be penalized, representing the notion that information within one document normally is more coherent than information spread over different documents. Furthermore, one could also consider inverting the direction, i.e., consider also `actor/acts_in/movie` relevant (with a lower similarity). Most generally, we could say that, for a pair of elements being relevant for a path query, only particular types of connection must exist, not necessarily a descendants relation. This has been considered before by Amer-Yahia et al. [2] and Shasha et al. [19, 20] for tree-based queries. However, in this paper, we focus on basic structural similarity that considers only descendants, but we plan to further investigate the issue in more advanced connection models, too.

There have been some proposals for indexing connections in collections of XML documents; see Section 2 for an overview. However, the choice of the "best" indexing strategy for a given collection of XML documents depends on a number of parameters: the size of the XML collection, the structure of the XML data (are there links, what is the link density, max depth of a document, number of tag names, distribution of the tag names, ...), and the query load (children or descendants queries, query patterns, locality). Therefore, there typically is no single "best" index for large, heterogeneous document collections. Instead, one index may be better suited for one part of the collection, while another may be best for another part. The existing indexes are not flexible enough to adapt to such a setting.

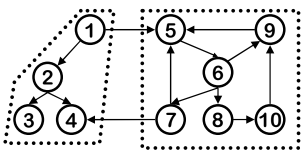

Fig. 1. Heterogeneous collection of ten XML documents)

The FliX[1] framework that we present in this paper overcomes this problem by combining indexes for parts of the collection, using the "best" index for each part. As an example, consider the document collection shown in Figure 1. It is easy to see that it consists of a part consisting of documents one to four that forms a tree, while the rest is closely interlinked. FliX partitions the document collection into the so-called meta documents as shown in the figure and chooses, depending on its configuration, for example a pre/postorder based index [10, 11] for the tree-like partition, and a HOPI index [18] for the partition with many links. This optimal choice of the index for a meta document allows faster evaluation of queries as well as more compact indexes. Using these indexes, queries are executed by first evaluating the query within a meta document and then following the links between meta documents at run time. This approach highly reduces the time required to return the most relevant results to the client.

[1] Framework for indexing large collections of interlinked XML documents

1.2 Contribution and Outline of the Paper

The paper presents a solution to the highly relevant problem of connection indexing in large, heterogeneous collections of XML documents. Specifically, we make the following important contributions:

- We present the FliX framework to index large, heterogeneous collections of interlinked XML documents and, ultimately, the Web, that reuses existing indexing strategies as building blocks. FliX is extensible and can be tailored to the needs of the application and to the structure of the data to be indexed. It efficiently supports XPath's `descendants-or-self` axis which is highly important for evaluating queries with structural vagueness, but can handle other query types, too.
- We show instantiations of the framework that strongly improve path index performance for practically relevant cases. This includes an extension of the well-known pre-/postorder index [10, 11] to documents with links.
- We have implemented the framework and carried out preliminary experiments on real XML data from DBLP. The results indicate that our framework outperforms existing indexing strategies in terms of query performance as well as index size.

The rest of the paper is organized as follows. In Section 2, we review related work. In Section 3, we introduce *FliX*, our flexible framework for indexing connections in linked XML documents. In Section 4, we give details how FliX is built, and Section 5 explains query evaluation in FliX. In Section 6, we discuss some preliminary experimental results.

2 Indexing the Structure of XML Documents

2.1 XML Data Model

We consider a graph $G_d = (V_d, E_d)$ for each XML document d in the collection, where 1) the vertex set V_d consists of all elements of d plus all elements of other documents that are referenced within d and 2) the edge set E_d includes all parent-child relationships between elements as well as links from elements in d to other elements of d or external elements. Then, a collection of XML documents $X = \{d_1, \ldots, d_n\}$ is represented by the union $G_X = (V_X, E_X)$ of the graphs $G_1, ..., G_n$ where V_X is the union of the V_{d_i} and E_X is the union of the E_{d_i}.

2.2 Existing Path Indexes

Several XML path indexes have been proposed in the literature, from relatively simple ways to highly efficient and space-effective structures. While they are typically quite efficient in evaluating simple path expressions, these approaches widely differ in space utilisation, support for paths with wildcards, and support for links.

Pre/Postorder Scheme (PPO). This path index proposed by Grust [10, 11] computes the *pre order* $pre(e)$ and the *post order* $post(e)$ for each element e of a single XML document without links, by traversing the document in depth-first order. Thus, building the index takes time $O(|E_X|)$, and space consumption is

$O(|V_X|)$. All XPath axes can be evaluated using these numbers, e.g., there is a path from x to y iff $pre(x) < pre(y)$ and $post(x) > post(y)$. This index structure is highly efficient for reachability queries and can, with slight additions, also provide the distance between two nodes.

HOPI. This index [18] makes use of a compact representation of reachability and distance information in graphs proposed by Cohen et al. [6]. The index maintains for each element e two sets of elements $L_{in}(e)$ and $L_{out}(e)$ (so-called *labels*) such that there is a path from x to y iff $L_{out}(x) \cap L_{in}(y) \neq \emptyset$. These labels can also be augmented with distance information to compute the distance of two elements. HOPI provides a divide-and-conquer algorithm for index creation that is reasonably fast as long as the document collection is not too large. Querying the index for reachability of two elements is very fast for most elements. Experiments [18] indicate that the HOPI index is usually an order of magnitude more compact than the transitive closure for document sets with relatively few links, and even more compact if the number of links increases. As there is no exact algorithm to compute HOPI's size (without actually building the index), it has to be estimated from the size of the transitive closure. A randomized algorithm to estimate this has been proposed by Edith Cohen [5]. However, for our current prototype we have not yet applied such elaborated methods for size estimation. Sayed and Unland [16] propose another path index that also applies Cohen's idea [6], but without providing an efficient algorithm for index building as HOPI does.

Others. There have been a number of other proposals for path indexes, among them Dataguides [9], APEX [4], Index Fabric [7], and the Index Definition Scheme [12] that can be used to define special indexes (e.g. 1–Index, A(k)–Index, D(k)–Index [15], F&B–Index) with k being the maximum length of supported paths. However, none of these approaches is explicitly optimized for the `descendants-or-self` axis, and therefore none of them efficiently supports them.

As discussed in Section 1, the choice of the "right" index for a structurally homogeneous document collection (like a meta document in FliX) depends on a number of parameters. As a first rule of thumb, if there are no links, PPO will typically be the best choice; if all paths are short or do not contain wildcards, APEX or an instance of the Index Definition Scheme will do fine; if we expect to evaluate long paths and queries with wildcards which is the case when we add structural vagueness to queries, we may want to use HOPI. However, HOPI's size may grow large for large document sets and, more importantly, the time to build HOPI superlinearly increases with increasing number of documents.

3 Flexible Indexing of XML Documents with FliX

3.1 FliX Concepts

As we pointed out in Section 2, the existing path index structures cannot efficiently support descendant queries on large collections of interlinked XML documents. Additionally, index size and the time to build the index grow enormously for some of the index structures if the document collection is huge, and it is usually impossible to find an "optimal" indexing strategy for a heterogeneous collection.

The FliX framework presented in this paper aims to solve these problems. It first divides the document set into carefully chosen fragments (so-called meta documents) where each meta document contains some or all of the links between its documents. Additionally, FliX maintains the set of remaining inter- or intra-document links that are not contained in any meta document. After that, an index is built for each meta document, using the "best" available indexing strategy given the characteristics of the meta document. A descendant query is then evaluated first on the local indexes (which will probably return the "best" results, i.e., elements that are connected with short paths). After that, results spanning multiple meta documents are evaluated by following links between meta documents at run-time.

FliX returns results approximately ordered by ascending distance. As soon as a new result is found, it is returned to the client for further processing. This is especially important as users are typically not interested in seeing all results of a query (which often would be way too many with documents from the Web), but may be satisfied with the top k results, with k usually less than 100. A search engine using FliX for indexing can therefore return the best results early to the user and may even stop the execution when it can determine that it has produced the top k results (e.g., using an algorithm similar to Fagin's threshold algorithm [8] with only sequential reads), or alternatively when the user decides to stop the query. In FliX, the decoupling between the client and the framework is implemented using a multithreaded architecture where the client thread reads from a list in which FliX inserts the results.

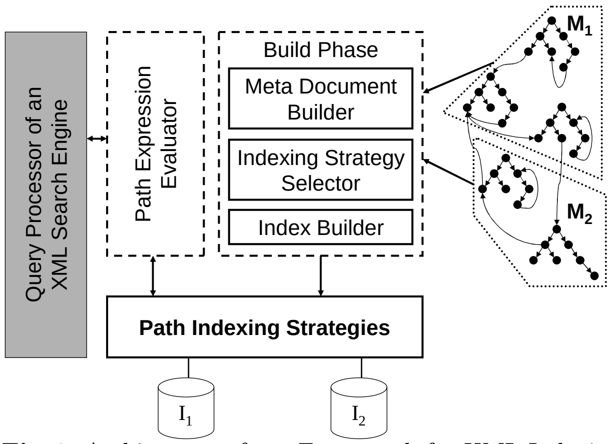
Query Processor of an XML Search Engine
Path Expression Evaluator
Build Phase
Meta Document Builder
Indexing Strategy Selector
Index Builder
Path Indexing Strategies
I_1
I_2
M_1
M_2

Fig. 2. Architecture of our Framework for XML Indexing

3.2 FliX Architecture

The architecture of FliX is shown in Figure 2 and consists of the following core components:

– Several *Path Indexing Strategies PIS* $S_1, \ldots, S_s$, among them PPO, APEX and HOPI, that support the XPath axes and return results in ascending order of distance.
– The *Meta Document Builder MDB* that automatically builds an "optimal" set of *meta documents* $M = \{M_1, \ldots, M_m\}$, where each meta document consists of a distinct subset of the set X of interlinked XML documents to be indexed.
– The *Indexing Strategy Selector ISS* that automatically selects, for each M_i of the meta documents, the "optimal" indexing strategy S_{s_i}, based on structure, size and other properties of the meta documents.
– The *Index Builder IB* that builds *index structures* $I_1, \ldots, I_m$ for each of the meta documents, using the automatically selected indexing strategy for each meta document.
– The *Path Expression Evaluator PEE* that evaluates path expressions with the `descendants-or-self` axis and returns results either approximately or exactly ordered by ascending path length.

The Meta Document Builder, the Indexing Strategy Selector, and the Index Builder are used in the *Build Phase*, while the Path Expression Evaluator is used in the *Query Evaluation Phase*. We present details in the following sections.

4 The Build Phase

The build phase consists of building the meta documents, selecting the optimal indexing strategy for each document, and building the indexes.

4.1 Meta Document Builder and Index Strategy Selector

Building the meta documents and selecting the best indexing strategies for them are closely intertwined. Both heavily depend on 1) the structure of the document collection, e.g., the number of documents, the distribution of the document sizes, link structure, and the average number of links per document, and 2) on the query load, e.g., which axes dominate the load, how long the typical result paths are, how often do result paths cross document borders, and so on. Additionally, certain algorithms to build meta documents may rule out the usage of some index strategies. As an example, if the meta documents are built to form graphs, not trees, then PPO can no longer be used. To overcome these problems in FliX, we have predefined several *configurations* of the framework, i.e., a strategy to build meta documents together with a set of indexing strategies that can be applied. Each configuration fits a certain kind of collection structure. The ultimate goal is that FliX can itself determine the "optimal" configuration for the actual application or, if the collection is too heterogeneous, automaticaly build homogeneous partitions of the collection. However, even though we have made some progress in that direction, we are still far away from having a solution; so in our current implementation, an administrator must decide which configuration to use.

When the configuration to use has been fixed, the problem of finding the "best" meta documents can be stated as follows: Given the set of interlinked XML documents X, an upper bound for (disk or main) memory available for index structures, and an upper bound for index creation time, find an "optimal" set of meta documents $M = \{M_1, \ldots, M_m\}$, where each meta document consists of a distinct subset of X. Here, the optimality of M means that, when indexing each M_i with the best possible indexing strategy, the average time for evaluating an arbitrary path expression over the complete document collection X is minimized. Additionally, total space consumption of all indexes and accumulated time to build the indexes must not exceed the respective upper bound. This optimization problem is hard to solve exactly, because the set cover problem is NP-hard, therefore each configuration comes with its own approximation algorithm that is presented in more detail in Section 4.3.

When the selection of the meta documents has been finished, we explicitly construct the meta documents as input for the Index Builder. To do this, we join the XML data graphs of all the documents within a meta document into the data graph of the meta document (i.e., we unify the sets of nodes and the sets of edges). Additionally, we (conceptually) replace each document-internal link within one of the documents as well as each inter-document link between two documents with a corresponding new edge in the XML data graph of the meta document. For nodes of the meta element that have links to elements of other meta documents, we store this information with the element together with the target elements of the links.

4.2 Index Builder (IB)

When the choice of the meta documents and of the corresponding indexing strategies has been made, we build all the index structures. Additionally, we maintain, for each meta document M_i, the set L_i of elements with outgoing links that are not reflected in the index. For a given element a of the meta document, the set $L(a)$ denotes all elements in the same meta document that are descendants a and have such an outgoing link. The index structures are extended to support querying for $L(a)$ of a given element a, which is (conceptually) computed by intersecting the set of descendants of a and L_i.

4.3 Example Configurations of FliX

Naive. The naive, albeit useful configuration of FliX considers each document in the XML document collection as a separate meta document. Depending on the structure of each document, either a PPO index is built if the document does not contain any intra-document links, or a HOPI or APEX index is built. This approach can be useful if documents are relatively large, the number of inter-document links is small, and queries usually do not cross document boundaries. As an example, the INEX benchmark collection of XML documents [13] would be a good candidate for using this configuration.

Maximal PPO. The most efficient path index is PPO, but it cannot be used with linked documents. However, a closer analysis shows that in some cases, the resulting XML graph G_X forms still a tree even in the presence of links, because the graph of the documents and the links between them is a tree and all links point to root elements. In other cases, removal of a small number of links leads to this situation. Figure 3 shows an example for this: If we removed the edge between documents 5 and 4, the resulting graph would form a tree. This opens up two possible configurations of FliX: (1) remove edges until the remaining graph forms a single tree and index it with PPO, or (2) using a greedy algorithm, build partitions of the graph of the documents such that each partition forms a tree, is as large as possible, and the number of partition-crossing edges is small; each partition then forms a meta document that is indexed with PPO.

This approach can be useful if there are relatively few links in the collection, like currently in the DBLP collection [14] where most documents are isolated.

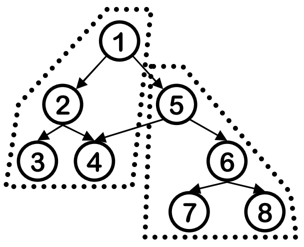

Fig. 3. Efficient partitioning of a document graph that is almost a tree

Unconnected HOPI. In the first step of the divide-and-conquer algorithm to build HOPI [18], partitions of the XML graph are built such that each partition does not exceed a configurable size and the number of partition-crossing edges is small. The second step then builds HOPI indexes for the partitions, and the last step then joins the subindexes to the index for the complete graph. It is straightforward to stop the algorithm after the second step and use the partitions as meta documents. This approach is useful when most documents contain links.

Hybrid Partitions. This configuration combines the second option of Maximal PPO with Unconnected HOPI. It first builds as many partitions as possible that can be indexed with PPO, and indexes the remaining documents with Unconnected HOPI. It is suited best for mixed settings like in Figure 1 where some parts of the collection contain few links, where others are densely linked.

5 Query Execution with FliX

Our index framework is optimized for the `descendants-or-self` axis used in path expressions of the form `a//B` [2], i.e., find all successors of element `a` that are of type `B`. This includes wildcard queries of the form `a//*` to find all successors of element `a`. A special case of this is to decide if two given elements `a` and `b` are connected, i.e., evaluating the reachability query `a//b`, and to determine the length of the shortest path between these elements. It also supports the immediate evaluation of a path expression like `A//B` where only the type of the starting element is fixed, but slightly less efficiently. The algorithms can be adapted easily for other cases, e.g., to support the child axis as in `a/B`, or to support the corresponding reverse axes like `ancestors-or-self`.

We describe the algorithm to evaluate the `descendants-or-self` axis in the following Section 5.1. After that, we shortly sketch in Section 5.2 how to evaluate other expression types.

[2] This is an XPath syntax where lower case letters represent fixed elements, upper case letters represent element types, and `//` represents XPath's `descendants-or-self` axis.

```
StreamedList PEE.findDescendantsByName(Element a, Type B)
  PriorityQueue IE;
  IE.insert(a,0);

  while (IE not empty)
  do
    Element e=IE.extractMin();
    MetaDocument M=e.meta();
    Index IND=M.Index();
    Set R=IND.findReachableElementsByName(e,B);
    for all r in R
    do
      return r;
    od
    Set L=IND.findReachableLinks(e);
    for all l in L
    do
      IE.insert(l,e.dist()+IND.dist(e,l)+1);
    od
  od
```

Fig. 4. Query evaluation algorithm of the PEE

5.1 Finding Descendants in Approximate Order

In the Query Execution Phase, the *Path Expression Evaluator (PEE)* computes the answer to reachability or distance queries of type `a//B` or `a//*`, for a start element a and some element type B. To calculate the result of the query, it makes use of the indexes built for the meta documents. The PEE returns a stream of result elements $e_1, e_2, \ldots$ that satisfy the query, approximately ordered by their distance to the start node a. In this subsection, we present a detailed description of the algorithm to evaluate a query (depicted in Figure 4). Note that a similar algorithm can be applied to find ancestors of a given node.

The PEE maintains a list IE of intermediate elements, ordered by ascending distance to the start element a of the query. The priority of an element in the queue denotes the minimal distance that any descendant of the element may have to the start node a. We denote the distance of two elements u and v by $\text{dist}(u, v)$. IE is initialized with a at priority 0. In the main loop of its algorithm, the PEE picks the first element e from IE and determines the meta document M in which e resides. Using the index for M, the PEE then computes the set R of all elements in M that are reachable from e and satisfy the search condition, sorted by ascending distance to e; the elements of R are returned to the client. Additionally, the set L of elements with outgoing links that are reachable from e is computed using the index for M. This step ignores all links that are already represented in the index for M. For each element $l \in L$, the destination(s) of its outgoing link(s) are then inserted into IE with priority $\text{dist}(a, e) + \text{dist}(e, l) + 1$, and the main loop continues until either IE runs empty or, if the client has specified a distance threshold, if the first entry in the queue has a larger distance than this threshold.

This algorithm does not return the result elements in perfectly ascending distance to a, because it returns the results within a single meta document as one block, regardless of their distance. However, this algorithm maintains a reasonably good approximation by processing the intermediate elements in ascending distance to a.

An important part of the algorithm, *duplicate elimination*, is left out in Figure 4. Duplicate results may occur because there may be cycles in the link structure or other multiple paths between documents, so that the algorithm visits the same meta document more than once. A straightforward approach would be to store all results retrieved so far in an internal data structure and, prior to returning a new result to the client, check if that result is already present in the result list. However, as there is no limit on the number of results for a query, this may require a lot of additional memory that could be used much more efficiently for caching. In our algorithm, the PEE stores only the entry points to meta documents that are visited throughout the evaluation of a query. When it then visits a meta document M_i through element e (i.e., e is a node in M_i to which a link from another meta document points), it first checks the relation of e to elements from M_i in the list of entry nodes. If e or an ancestor of e have been visited before, all descendants of e have already been returned to the client, so e can be immediately dropped. Otherwise, for every descendant d of e found in this step, the PEE returns d only to the client if d is not a descendant of another entry node. All these checks can be efficiently made using the connection index for M_i.

The maximum time to evaluate a query heavily depends on the selection of the meta documents. In an ideal setting, the MDB will have prepared the meta documents such that processing time is minimized for most queries, so only a few queries should require many hops across inter-document links. Additionally, as the best results are typically within a relatively small distance from the start element, the query processor of the search engine may decide to cancel the query after a certain amount of results have been returned or after the distance of the current result is beyond a certain threshold.

5.2 Evaluation of Other Expressions

Connection Test. Testing if two elements a and b are connected is straightforward: The priority queue is initialized with a at priority 0, and the algorithm proceeds until finds b. As this may result in a traversal of the complete document collection at worst, the maximal depth of the search should be limited by the client. This can be done easily because the client uses the length of the path between a and b to compute the relevance of this pair of elements, so it can compute a threshold for the path length beyond which the resulting relevance is negligible.

Even though the threshold leads to quick answers in most cases, there is still more room for optimizations. As an example, one could start two evaluations instead of one: the first starts at a and looks for descendants, while the second one starts at b and looks for ancestors. Depending on the structure of documents, either of them may be the best.

Evaluating `A//B` expressions. To evaluate queries like `A//B` where only the type of the starting element is fixed, only the initial step of the PEE has to be modified: The PEE determines all elements of type A and inserts them into the priority queue with priority 0, then the evaluation algorithm is run.

6 Preliminary Experiments

In this section, we present the results of some preliminary experiments on small sets of data with our current implementation of FliX that has not yet been optimized. Even though these experiments are far from being exhaustive, they strongly indicate the viability of our indexing framework.

We compared our approach to two other indexing strategies: an extended version of HOPI that supports distance information and a database-backed implementation of APEX (without optimizations for frequent queries), both applied to the complete data collection. We tested four configurations of FliX: The naive configuration where a PPO index is built for each document (PPO-naive), a simple implementation of Maximal PPO, and two variants of Unconnected HOPI with partition sizes set to 5,000 (HOPI-5000) and 20,000 nodes (HOPI-20000). We implemented all strategies as multithreaded, Java-based applications that store all information in database tables and do not explicitly cache information in main memory, so the absolute execution times for queries are relatively big. All our experiments were run on a Windows-based PC with a 1.6GHz Pentium-M processor and 1 GByte RAM. We used an Oracle 9.2 database than ran on a Windows-based server with a two 3GHz Pentium IV processors, 2GB of RAM, and a 240 GB 4-way SCSI RAID-0 disk array.

The data collection used for the experiments was extracted from the DBLP collection [14]. We generated one XML document for each 2nd-level element of DBLP (`article`,`inproceedings`,...) and chose the corresponding documents for publications in EDBT, ICDE, SIGMOD and VLDB and articles in TODS and VLDB-Journal. The resulting collection consisted of 6,210 documents with 168,991 elements and 25,368 inter-document links with an overall size of 27 megabytes.

index	HOPI	APEX	PPO-naive	HOPI-5000	HOPI-20000	Maximal PPO
size [MB]	266.6	2.2	1.9	5.0	17.9	1.9

Table 1. Index sizes

Table 1 shows the database storage required for the different indexes. The HOPI index is huge, but it is still more than an order of magnitude smaller than storing the complete transitive closure. The HOPI-5000 configuration requires only about twice as much space as the APEX index, and naive PPO and Maximal PPO are even smaller. It is evident from these figures that using FliX can save a lot of space as compared to the HOPI index, and that the Maximal PPO configuration is as space efficient as PPO.

To assess the performance of FliX, we submitted a query to determine all `article` descendants of Mohan's VLDB '99 paper about ARIES. Figure 5 shows the time that the different indexes needed to return up to 100 results for this query. HOPI was clearly the fastest to return all results, with an almost constant time of 0.6 seconds. However, the FliX configurations HOPI-5000 and HOPI-20000 outperformed HOPI in the time needed to return the first results, and they clearly improved on APEX. Even though it uses larger meta documents, HOPI-20000 is not constantly better than HOPI-5000; we attribute this to the randomized selection of partitions by the HOPI index builder, with the partitioning chosen for HOPI-20000 being less optimal for this query than the one for HOPI-5000. Maximal PPO returned the first results even faster than the other indexes because these results were in the first, large trees MaximalPPO considered, but its performance for later results was not as good because, unlike the unconnected HOPIs, it had to follow many links at run-time. As both connected HOPI configurations and Maximal PPO are only approximative algorithms, we also checked the error rate (i.e., fraction of all results that were returned in wrong order); it was 8.2% for HOPI-5000, 10.4% for HOPI-20000, and 13.3% for Maximal PPO, which is tolerable for most applications. Naive PPO was constantly slower than the other indexes, because documents were so small that the quick index lookup per document with PPO could not compensate for the additional overhead of following all links at runtime. Other experiments with different start elements and different tag names showed similar results.

We also experimented with testing if two nodes are connected. Here, we found the same performance trend as before, only with lower absolute numbers.

These experimental results have shown that FliX can increase performance of answering descendants or connection queries, at the price of a moderate increase in storage cost and some error in the order of results.

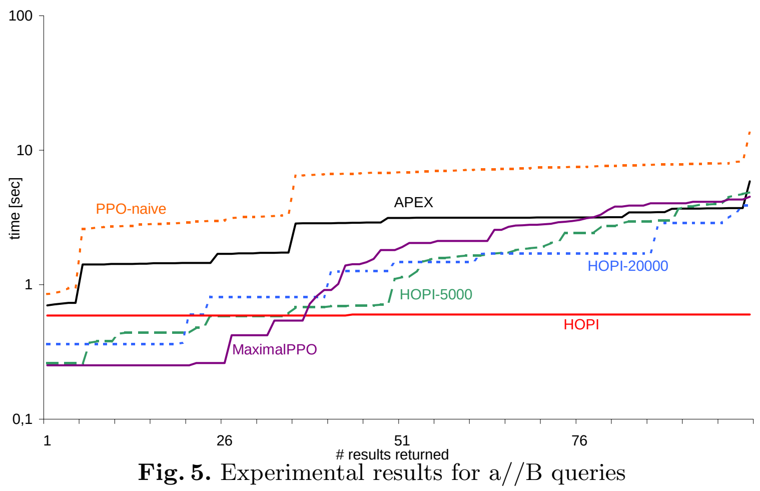

Fig. 5. Experimental results for a//B queries

7 Conclusion and Future Work

This paper presents important steps towards indexing connections in large sets of interlinked XML documents, a task where today's existing index structures fall short. Given the proliferation of XML as data format for large-scale document collections and on the Web, such an index structure is badly needed for efficient retrieval in future applications.

Our future work will aim to improve FliX in the following regards:

– We plan to investigate more sophisticated algorithms for building meta documents, including automatic methods that analyze the document collection, identify homogeneous subcollections, and choose the best indexing strategy for each subcollection. This may include partitioning each subcollection into fragments for which the estimated index size and the estimated time to build the index is tolerable, which requires advanced estimators for both. Another option that we have in mind is building meta documents on the element level, i.e., ignore the artifical boundary of documents and combine semantically related, connected elements into a single meta document.
– We are looking into further optimizing FliX's query evaluation algorithm, for instance returning results exactly sorted instead of approximately, eliminating unnecessary link traversals when checking connectivity of two nodes, and caching results of frequent (sub-)queries.
– We consider adding some self-tuning functionality to FliX: If it turns out in the query evaluation engine that most queries have to follow many links, then the choice of meta documents is no longer optimal for the current query load. In this case, the build phase should start again, taking statistics on the query load into account.
– We plan to extend FliX to handle not only XPath axes, but also more general concepts of connectivity as sketched in Section 1.
– We will carry out additional experiments to test the scalablity of FliX with larger sets of documents and to test the adaptivity of FliX with more heterogeneous document collections.

References

[1] The Internet Movie Database. `http://www.imdb.org`, downloaded 10/25/2003.

[2] S. Amer-Yahia et al. Tree pattern relaxation. In C. S. Jensen et al. (eds.), *8th Int. Conference on Extending Database Technology (EDBT)*, pages 496–513, 2002.

[3] H. Blanken, T. Grabs, H.-J. Schek, R. Schenkel, and G. Weikum (eds.). *Intelligent Search on XML Data*, volume 2818 of *LNCS*. Sept. 2003.

[4] C.-W. Chung et al. APEX: An adaptive path index for XML data. In M. J. Franklin et al. (eds.), *ACM SIGMOD Int. Conference on Management of Data*, pages 121–132, 2002.

[5] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, Dec. 1997.

[6] E. Cohen et al. Reachability and distance queries via 2-hop labels. In D. Eppstein (ed.), *13th ACM-SIAM Symposium on Discrete algorithms (SODA)*, pages 937–946, 2002.

[7] B. Cooper et al. A fast index for semistructured data. In P. M. G. Apers et al. (eds.), *27th Int. Conference on Very Large Data Bases*, pages 341–350, 2001.

[8] R. Fagin et al. Optimal aggregation algorithms for middleware. In *20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 2001.

[9] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In M. Jarke et al. (eds.), *23rd Int. Conference on Very Large Data Bases*, pages 436–445, 1997.

[10] T. Grust. Accelerating XPath location steps. In M. J. Franklin et al. (eds.), *ACM SIGMOD Int. Conference on Management of Data*, pages 109–120, 2002.

[11] T. Grust and M. van Keulen. Tree awareness for relational DBMS kernels: Staircase join. In Blanken et al. [3], pages 231–245.

[12] R. Kaushik et al. Covering indexes for branching path queries. In M. J. Franklin et al. (eds.), *ACM SIGMOD international conference on Management of data*, pages 133–144, 2002.

[13] G. Kazai et al. The INEX evaluation initiative. In Blanken et al. [3], pages 279–293.

[14] M. Ley. DBLP XML Records. http://dblp.uni-trier.de/xml/. Downloaded Sep 1st, 2003.

[15] C. Qun et al. D(k)-index: An adaptive structural summary for graph-structured data. In A. Y. Halevy et al. (eds.), *ACM SIGMOD International Conference on Management of Data*, pages 134–144, 2003.

[16] A. Sayed and R. Unland. Index-support on XML-documents containing links. In *46th Int. IEEE Midwest Symposium On Circuits and Systems*, 2003.

[17] R. Schenkel, A. Theobald, and G. Weikum. Ontology-enabled XML search. In Blanken et al. [3], pages 119–131.

[18] R. Schenkel, A. Theobald, and G. Weikum. HOPI: An efficient connection index for complex XML document collections. In *9th Int. Conference on Extending Database Technology (EDBT)*, pages 237–255, 2004.

[19] D. Shasha et al. ATreeGrep: Approximate searching in unordered trees. In *14th Int. Conference on Scientific and Statistical Database Management*, pages 89–98, 2002.

[20] D. Shasha and K. Zhang. Approximate tree pattern matching. In A. Apostolico and Z. Galil (eds.), *Pattern Matching Algorithms*, pages 341–371. Oxford University Press, 1997.

[21] A. Theobald and G. Weikum. Adding Relevance to XML. In D. Suciu and G. Vossen (eds.), *3rd Int. Workshop WebDB 2000*, volume 1997 of *LNCS*, pages 105–124, 2000.

[22] A. Theobald and G. Weikum. The index-based XXL search engine for querying XML data with relevance ranking. In C. S. Jensen et al. (eds.), *8th Int. Conference on Extending Database Technology*, volume 2287 of *LNCS*, pages 477–495. 2002.

[23] A. Theobald and G. Weikum. The XXL search engine: Ranked retrieval of XML data using indexes and ontologies. In M. J. Franklin et al. (eds.), *ACM SIGMOD Int. Conference on Management of Data*, 2002.