Index vs. Navigation in XPath Evaluation

Norman May, Matthias Brantner, Alexander Böhm, Carl-Christian Kanne, and Guido Moerkotte

University of Mannheim
68131 Mannheim, Germany
norman|msb|boehm|cc|moer@db.informatik.uni-mannheim.de

Abstract. A well-known rule of thumb claims, it is better to scan than to use an index when more than 10% of the data are accessed. This rule was formulated for relational databases. But is it still valid for XML queries? In this paper we develop similar rules of thumb for XML queries by experimentally comparing different execution strategies, e.g. using navigation or indices. These rules can be used immediately for heuristic optimization of XML queries, and in the long run, they may serve as a foundation for cost-based query optimization in XQuery.

1 Motivation

XPath is used as a stand-alone query language and as part of XSLT or XQuery to address parts of an XML document. As an increasing number of applications rely on one of these query languages, efficient evaluation of XPath expressions has emerged as a focal point of research. In particular, efficient retrieval techniques must be chosen when queries access large instances of XML documents. Three core aspects influence their performance:

1. storage structures [2, 11, 12, 16, 8, 21, 28, 7],
2. algorithms to evaluate XPath queries [1, 15, 16, 10, 27, 6], and
3. an optimizer that selects a cost-optimal retrieval method given 1 and 2.

As the (incomplete) list of citations indicates, many proposals exist for the first two aspects. However, research on optimization of XPath has just scratched the surface. The only cost-based optimizers we know [5, 29] are limited to index-based storage structures for which estimating access costs does not fundamentally differ from relational storage. Most query engines only perform heuristic optimizations. Instead, we expect to find better query execution plans when cost-based optimization is used. Nevertheless, it would already help if we had simple rules of thumb, as established for relational databases [14]. In particular, is it still true for XML queries that it is better to scan (or navigate) than to use an index when more than 10% of the data are accessed?

In this paper we analyze the performance of known techniques for the evaluation of structural XPath queries. Particularly, we use Natix [11] to compare the following two execution strategies: (1) navigation through the logical structure of the document and (2) index-based retrieval in conjunction with structural joins. Our findings can be applied to improve heuristics used in XPath query optimizers. In the long term, such experiments will help us to derive cost information which can be used by cost-based XQuery optimizers.

To be able to derive costs, it is important to understand the architecture that underlies query evaluation. Hence, we present an abstract model for storing XML documents in Sec. 2. This storage model includes indices which employ XML node labeling schemes. In Sec. 3 we proceed with an overview of NAL, the Natix physical algebra. The algebraic operators included in NAL form the building blocks for the query evaluation plans we investigate in Sec. 4. These evaluation plans are experimentally compared in Sec. 5. Our experiments show that there is not only one best choice for evaluating XPath expressions. But it is still true that index based-techniques are often superior when only small parts of the data are touched by a query. In contrast, full scans perform better when large fragments of the input qualify for the query result. Based on these results, we conclude the paper and discuss future work (Sec. 6).

2 The Storage Model

Since retrieving data is one of the main cost drivers for XML query evaluation, it is important to understand how XML data is stored and how it can be accessed. In this section we briefly survey fundamental techniques available to access XML data.

2.1 Physical XML Storage

The most general approach to store XML data maps the logical structure of an XML document to physical storage structures. These physical storage structures offer primitives to directly navigate through the logical structure of the XML document. Consequently, it is possible to reconstruct the logical structure of the original document from its physical representation. While there are many possible implementations for this mapping, e.g. [4, 11, 16, 8, 7, 27, 28], we believe that some general ideas are shared by all storage schemes:

1. Fragments of the XML document are assigned to storage units.
2. The document structure can be reconstructed from the physical storage.
3. The cost to access a part of the document depends on its storage location.

Consider the XML document depicted in Fig. 1 as an ordered labeled tree. The nodes in the tree correspond to element nodes, and the edges represent parent-child relationships. A main memory representation, e.g. DOM, assigns an address in main memory to each node in the tree. Pointers connect a node to its children, siblings, or parent. It is quite easy to see that the logical document structure can be reconstructed from this representation by traversing the pointers. Given the node with subscript 1.1.1.1, one can imagine that accessing its first child might be cheaper than accessing a node that is further away in the

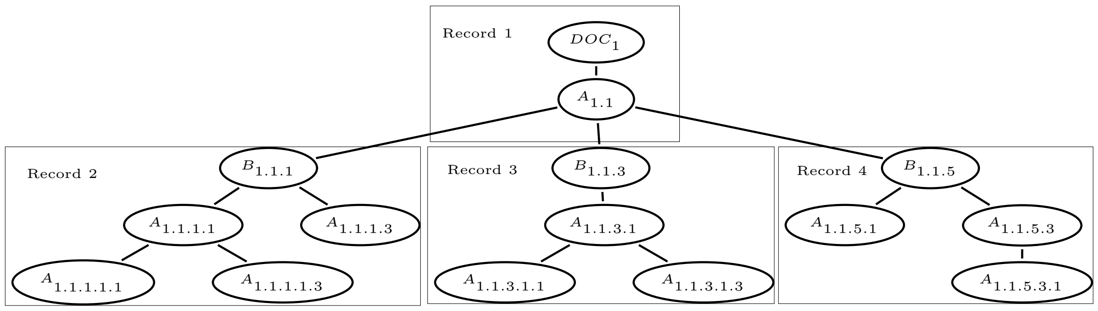


Fig. 1. The tree representation of an XML document

document structure, e.g. the node with subscript 1.1.5.3. The latter operation might even require disk I/O if the target node does not reside in physical main memory.

Natix assigns fragments of XML documents to records of variable size and stores them on physical pages on disk [11]. The logical structure of the document is preserved by a structured storage inside the records and by references to other records. A physical storage address (NID) is assigned to each node. It is used to implement these references or to directly access XML nodes. The document in Fig. 1, for instance, is mapped to four physical records. It is possible to traverse nodes inside a record or to navigate to other records in the XML store. Navigation to a record on another page results in disk I/O if this page is not yet in the main memory buffer. The cost of this page access depends on the characteristics of the storage device and the location of the page on disk.

2.2 XML Node Labels

For efficient XML query evaluation, we want to support indices. Additionally, we require an efficient processing of updates. Hence, we employ logical node identifiers (LIDs) to abstract from physical storage locations. In Natix we have implemented LIDs as ORDPATH IDs [9]. As an incarnation of Dewey IDs [3], they satisfy the two requirements mentioned above. Furthermore, ORDPATH IDs support efficient query processing because given two LIDs, it is possible to test their structural relationship with respect to any XPath axis. In the document shown in Fig. 1, the subscript of each node represents the ORDPATH ID assigned to this node.

2.3 Indexing XML

We index XML documents in B^+-trees as proposed by Chien at. al [2]. There exist many other proposals for indexing XML documents, e.g. [12, 21]. But we believe that our approach provides a solid performance for a wide range of queries and at the same time supports efficient concurrent updates.

The index we use in Natix is an implementation of a B^+-tree with sibling links based on the algorithms proposed by Jaluta et. al [20]. This B-link index allows storing keys of variable size and performs online rebalancing and deallocation operations in case of underutilized nodes. These operations are especially beneficial as the index performance does not deteriorate due to document updates, and explicit garbage collection operations become obsolete. Other specific features useful for processing ORDPATH IDs include key-range scans exploiting the sibling links and high concurrency by restricting the number of latches to a minimum.

The general idea of our indexing scheme is shown in Fig. 2. We create two indices: one called *Tag2Lid* and the other *Lid2Nid*.

Tag2Lid maps tag names to LIDs. The key value is the tag name, and the indexed value is the LID. For the same tag name, LIDs are returned in document order, i.e. in ascending order.

Lid2Nid maps LIDs to their physical storage address. This index is optional when the storage manager directly provides access to XML fragments based on their LID. However, for generality we will explicitly use this index to locate result nodes of an XPath expression.

Tag2Lid		Lid2Nid	
Tag	Lid	Lid	Nid
A	1.1	1	1
A	1.1.1.1	1.1	2
A	1.1.1.1.1	1.1.1	3
...	...	1.1.1.1	4
A	1.1.5.3	...	
A	1.1.5.3.1	1.1.5	12
B	1.1.1	1.1.5.1	13
B	1.1.3	1.1.5.3	14
B	1.1.5	1.1.5.3.1	15
DOC	1		

Fig. 2. Indexing XML documents

3 Execution Model

Besides the physical storage structures, the cost of a query execution plan is determined by the algorithms it uses to access the data and to evaluate the query. Hence, we also need to comment on the available algorithms. We present an algebraic approach to XPath evaluation because both an algebraic query optimizer and an evaluation engine is easy to extend with new operators [1, 27]. Additionally, we want to benefit from the experiences gathered with cost-based algebraic optimizers built for relational and object-oriented databases. The Natix Physical Algebra includes all algebraic operators we need to implement the query execution plans (QEPs) discussed in Sec. 4.

3.1 Architecture and Notation

Our physical algebra works on sequences of tuples. Each tuple contains a set of variable bindings representing the attributes of the tuple. Some physical operators extend these bindings by appending attribute-value pairs to a tuple. All operators are implemented as conventional iterators [13]. Tuples are passed in a pipelined fashion from the argument operator to its consumer. The set of supported operators we cover here comprises the common algorithms used to execute XPath queries [1, 10, 27].

To test structural relationships between LIDs or XML nodes, we arrange for the following notations: We denote with $n \Downarrow m$ that m is descendent of n and with $n \downarrow m$ that m is child of n.

3.2 General Operators

The following operators are used independently of the XPath evaluation method. The *Singleton Scan* ($\Box$) returns a singleton sequence consisting of the empty tuple. We denote the *Map* operator by $\chi_{a:e_2}(e_1)$. It extends each tuple t in e_1 with attribute a. The value bound to a is computed by evaluating the subscript e_2 with variable bindings taken from t. For example, we use the Map operator to dereference physical node ids, i.e. $\chi_{n:*a}(e_1)$, or to retrieve the root node of a document, i.e. $\chi_{r:root}(e_1)$. To remove duplicates on a set of attributes A, we use a *Duplicate Elimination* operator ($\Pi^D_A(e)$). To rename a set of attributes A into another set of attributes A', we use $\Pi_{A':A}(e)$. Some evaluation strategies require us to reestablish document order on a sequence of nodes bound to attribute a. Therefore, we employ the sort operator ($Sort_a$). The *D-Join*, denoted by $e_1 \underset{\rightarrow}{\bowtie} e_2$, joins each tuple t in e_1 with all tuples returned by the evaluation of e_2. The result returned by e_2 depends on the attribute bindings of t. Similar to the MapConcat operator in [27], it is used to concatenate the evaluation of location steps inside a location path.

3.3 XPath Navigation Operators

In this paper we use the *Unnest-Map* operator $\Upsilon_{cn:c/a::n}(e)$ to evaluate XPath location steps by navigation. Given a context node stored in variable c of a tuple $t \in e$, it evaluates axis a and applies node test n to the remaining candidate nodes. Each result node is bound to variable cn in the result tuple. During the evaluation of a location step the operator navigates through the document that potentially contains result nodes. This traversal is done for every context node.

3.4 Index-Aware Operators

For efficient XPath evaluation the *Structural Join* ($e_1 \bowtie^{ST-J}_p e_2$) was proposed [6]. It joins one sequence of tuples of context nodes, e_1, with a sequence of candidate nodes, e_2. Both sequences are sorted in document order. Predicate p tests the axis step relation that must hold between nodes of the two sequences. We classify the Structural Join as index-aware because it solely uses information provided by the LIDs and thus only relies on index information.

We employ the *IndexScan* ($Idx_{n;A;p;rp}$) to access data stored in a B-link tree named n, e.g. the index “Tag2Lid” or “Lid2Nid”. A is a set of attribute bindings established by the scan. It must be a subset of the attributes defined in the schema of the index. Predicate p optionally tests the upper and lower bound. Residual predicate rp is an optional predicate applied to each tuple before it is passed to the consumer operator.

4 Query Execution Plans

In Sections 2 and 3 we have laid the foundation for several evaluation techniques available for XPath. We now discuss the following types of query execution plans (QEPs):

1. Navigate through the XML document (e.g. in a DOM-like fashion) [1].
2. Use indices to access the candidate nodes of each navigation step and relate them by join operations to evaluate the query. If there are multiple navigation steps, we have two more choices:
   (a) Access indices in the order specified in the query.
   (b) Reorder the index accesses and possibly sort the result nodes at the end [29].

In our opinion, these types of QEPs comprise a wide variety of XPath evaluation techniques that have not been compared yet. For each alternative mentioned above, we present a QEP for the query `/DOC/TAG1/TAG2`. Even this simple query allows us to point out the advantages of each alternative. The reason is that each QEP exploits structural relationships, selectivities of location steps, or physical storage characteristics to different degrees. As we will see in our experiments, there is no single plan that is consistently faster than the other alternatives.

4.1 Plan Using Navigation

The most direct translation of the XPath expression results in a navigational plan [1]. The result of the *stacked translation* of the query into our algebra is depicted in Fig. 3. The topmost operator of the QEP is a D-Join which initializes the context for the XPath query evaluation to the root node. The right argument is evaluated with the bindings taken from this context. The stacked translation results in a sequence of Unnest-Map operators, each of which evaluates one location step. In general, to compute the resulting node set, duplicates have to be removed, and the result nodes have to be sorted by document order. In our example query duplicate elimination or sorting can be avoided [17, 18].

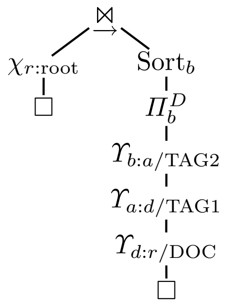
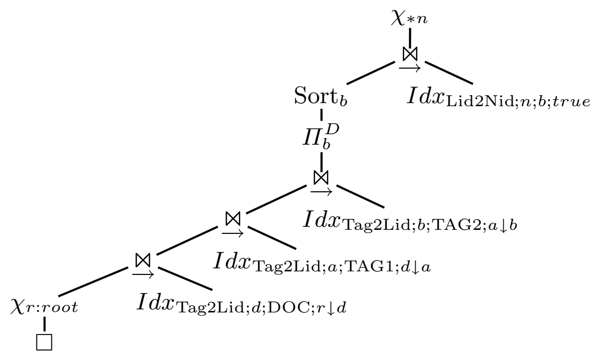


Fig. 3. Plan using navigation

When the QEP is evaluated, each Unnest-Map operator traverses some part of the document, starting at the current context node. E.g. during a child step all children of the current context node will be visited. When a node satisfies all node tests, it is passed to the next operator where it may serve as another context node.

This evaluation strategy has three basic consequences: (1) Non-matching nodes may implicitly prune parts of the document from the traversal. Thereby, accessing physical pages is avoided for potentially large parts of the document. (2) Location steps may visit intermediate nodes that will never be part of a matching location path. E.g. for descendant steps we have to look at all descendant nodes of the context node. (3) During the document traversal, visiting



χ_{*n}
$\overrightarrow{\bowtie}$
Sort_b
$Idx_{\mathrm{Lid2Nid};n;b;true}$
Π_b^D
$\overrightarrow{\bowtie}$
$Idx_{\mathrm{Tag2Lid};b;\mathrm{TAG2};a\downarrow b}$
$\overrightarrow{\bowtie}$
$Idx_{\mathrm{Tag2Lid};a;\mathrm{TAG1};d\downarrow a}$
$\overrightarrow{\bowtie}$
$\chi_{r:root}$
$Idx_{\mathrm{Tag2Lid};d;\mathrm{DOC};r\downarrow d}$
$\square$

Fig. 4. Plan with index access in order

physical pages may lead to random I/O and multiple visits to the same physical page.

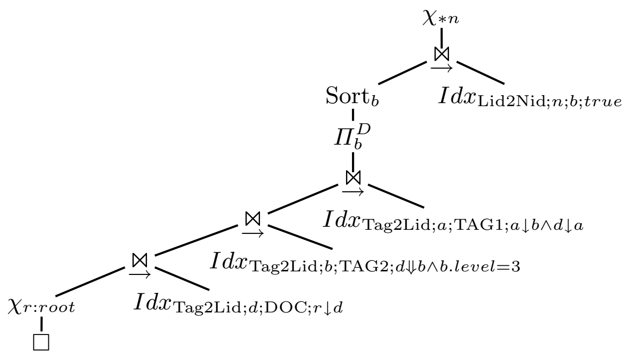
4.2 Plan Using Index

The motivation for using an index is to retrieve only nodes with tag names that satisfy a query predicate. The translation into a plan using an index is an application of the *canonical translation* presented in [1] or the XQuery translation of [10].

The result of this translation is shown in Fig. 4. The data flow of the QEP goes from the bottom-left leaf node upwards to the root of the QEP. First, the root node is initialized as context node. This context can be used to restrict the range scan in the index "Tag2Lid". This index access is performed in the dependent part of each D-Join in the plan. We have to apply the residual predicate to each node retrieved from the index. Together with the range predicate, this test completes the structural test between context node and document node. Before all physical nodes are retrieved, we possibly have to perform a duplicate elimination and a sort [18]. Finally, we employ the index "Lid2Nid" to get the physical nodes of the query result and access the physical nodes on disk using a Map operator. Note that some queries do not require this final dereferencing step, e.g. quantified queries or queries with count aggregate. This can be used in favour of such queries.

The index-based technique has the following properties: (1) It only considers nodes which can match the node tests in the query. (2) The index is repeatedly accessed for each context node. This results in random I/O, as the same index is accessed for different location steps. (3) Context information can be used to prune the set of candidate nodes. This depends on the availability of e.g. level information for axis steps to sibling nodes. (4) Parts of structural queries can be answered solely based on LIDs. Hence, less information needs to be stored in the index. This potentially decreases the required I/O bandwidth. (5) Additional I/O is needed to retrieve the result nodes of the query.

We now turn our attention to index-based QEPs in which we reorder location steps. We treat the reordering of location steps separately because there are two main issues that limit the value of join reordering for XPath expressions: (1) Join



χ_{*n}
$\overrightarrow{\bowtie}$
Sort_b
$Idx_{\mathrm{Lid2Nid};n;b;true}$
Π_b^D
$\overrightarrow{\bowtie}$
$Idx_{\mathrm{Tag2Lid};a;\mathrm{TAG1};a\downarrow b\wedge d\downarrow a}$
$\overrightarrow{\bowtie}$
$Idx_{\mathrm{Tag2Lid};b;\mathrm{TAG2};d\Downarrow b\wedge b.level=3}$
$\overrightarrow{\bowtie}$
$\chi_{r:root}$
$Idx_{\mathrm{Tag2Lid};d;\mathrm{DOC};r\downarrow d}$
$\square$

Fig. 5. Plan with index access reordered

ordering in general is known to be NP hard. When we allow to sort by document order at the end, the search space contains $O((2n)!/n!)$ bushy join trees and $O(n!)$ left deep join trees [24, 25, 22] containing n joins. Here, we consider one scan for each location step and include cross products. (2) The quality of the generated plans heavily depends on the precision of cardinality estimates [19]. However, good methods for cardinality estimation are known only for restricted classes of XPath [26, 30].

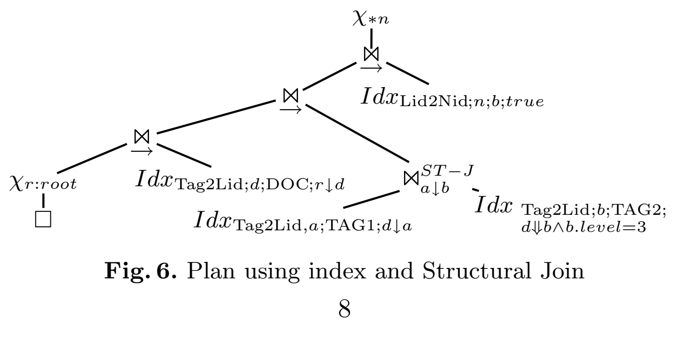
The reordered translation is shown in Fig. 5. There are three differences to the previous plan: (1) We reordered the axis steps. (2) The residual predicates had to be adjusted. (3) To establish the document order, we need a final sort.

The potential value of reordering axis steps stems from the possibility of evaluating axis steps with low result cardinality first to minimize the number of lookup operations in the index. The additional freedom of reordering location steps has to be payed with an additional sort operation (which is always needed now) and less restrictive structural predicates. Hence, it is not clear which strategy is better in which case. In general, this decision should be based on costs.

4.3 Plan Using Index and Structural Join

The plans discussed in the previous section access the index for each context node. Thanks to the Structural Join, we can evaluate a location step with a single scan of each input and we still have full the freedom to choose the most efficient plan among all bushy join constructed with Structural Joins [6, 29].

One possible QEP using Structural Joins is depicted in Fig. 6. In this plan, a Structural Join is performed between the nodes with tag name TAG1 and TAG2.



χ_{*n}
$\overrightarrow{\bowtie}$
$\overrightarrow{\bowtie}$
$Idx_{\mathrm{Lid2Nid};n;b;true}$
$\overrightarrow{\bowtie}$
$\chi_{r:root}$
$Idx_{\mathrm{Tag2Lid};d;\mathrm{DOC};r\downarrow d}$
$\bowtie^{ST-J}_{a\downarrow b}$
$Idx_{\mathrm{Tag2Lid},a;\mathrm{TAG1};d\downarrow a}$
$Idx_{\mathrm{Tag2Lid};b;\mathrm{TAG2};d\Downarrow b\wedge b.level=3}$
$\square$

Fig. 6. Plan using index and Structural Join

Since both input sequences are sorted in document order, the Structural Join can compute its result with one scan through both sequences and some additional buffering. The resulting node sequence is sorted by document order and does not produce duplicates.

5 Experiments

We now compare the performance characteristics of the QEPs (see Sec. 4) for three XPath queries on synthetic data sets. First, we discuss the structure and characteristics of the synthetic documents and introduce the XPath queries. Then, we discuss the experimental results gathered from executing each type of QEP on each query.

We have executed all queries on Natix Release 2.1 [11]. We used a buffer size of 8MB. Each query was executed three times with cold buffer. We report the average of all three evaluations. Our execution environment was a PC with two 3GHz Intel Xeon CPUs, 1 GB of RAM, 34GB hard disc (FUJITSU MAS3367NC) running SUSE Linux with kernel 2.6.11-smp.

5.1 Data Set

To get precise performance characteristics for each of the evaluation strategies, the queries access generated data sets. This allows us to tune the selectivity of each XPath location step individually.

The input documents used in our experiments were generated by the XDG document generator implemented by our group.[1] It allows to specify several parameters, i.e. the number of nodes, the document depth, the fan-out of each element, and the number of different tag names.

Conceptually, the generator creates as many child nodes as defined by the parameter "Fan-out" and resumes with a recursive call for each child. When the depth of the recursive calls reaches the specified parameter value "Depth", no recursive calls are executed any more. The frequency of occurrences of tag names decreases by a factor 2 for each subsequent tag name. E.g. the argument "C" for parameter "Elements" means that the tag names `A`, `B`, and `C` are used in the document where every second node gets tag name `A`, every fourth node gets tag name `B`, and so on. To get up to 100%, nodes with tag name `A` are generated. In our setup this means that exactly 50.1% of the nodes are `A` nodes. The tool generates new nodes until the limit for the number of nodes (#Nodes) is reached.

In principle, this generator might introduce correlations between predicates such that the distribution of tag names strongly depends on the parent nodes. For our data sets this is not the case for tag names `A`, `B`, `C`, and `D`. However, the remaining tag names only occur as leaf nodes.

We generated documents of four sizes with the parameters summarized in Fig. 7. In this figure, we give the size of the generated text. This setup allows

[1] available for download at http://db.informatik.uni-mannheim.de/xdg.html

Document Parameter		Document Instance			
Parameter	Description	0.327MB	3.46MB	36.5MB	384MB
#Nodes	# of generated XML elements	10,000	100,000	1,000,000	10,000,000
Depth	max. depth of the document	4	5	6	7
Fan-out	# of children per element node	10			
Elements	# of different tag names	"J" (10)			

Fig. 7. Parameters and characteristics of generated documents

us to control the selectivity of each location step between 50% and 0.1% by changing the name test of each location step.

5.2 XPath Queries

We have compared the performance characteristics of the QEPs discussed in Sec. 4 for the following three XPath query patterns:

Q1: /descendant::TAG. This query reveals the impact of the access patterns of the QEPs because when evaluating this query structural information is unimportant. The navigational plan visits the whole document to access all potential result nodes. In contrast, the index-based plans only visit specific parts of the document, i.e. those including nodes with tag name `TAG`.

The main difference is that the navigational plan performs random I/O in the worst case, whereas the index-based QEP can directly retrieve the requested nodes by a range scan on the "Tag2Lid" index.

Q2: /DOC/TAG1/TAG2. With this query we investigate (1) how well each plan alternative exploits structural properties demanded by the query, and (2) how reordering navigation steps effects query performance.

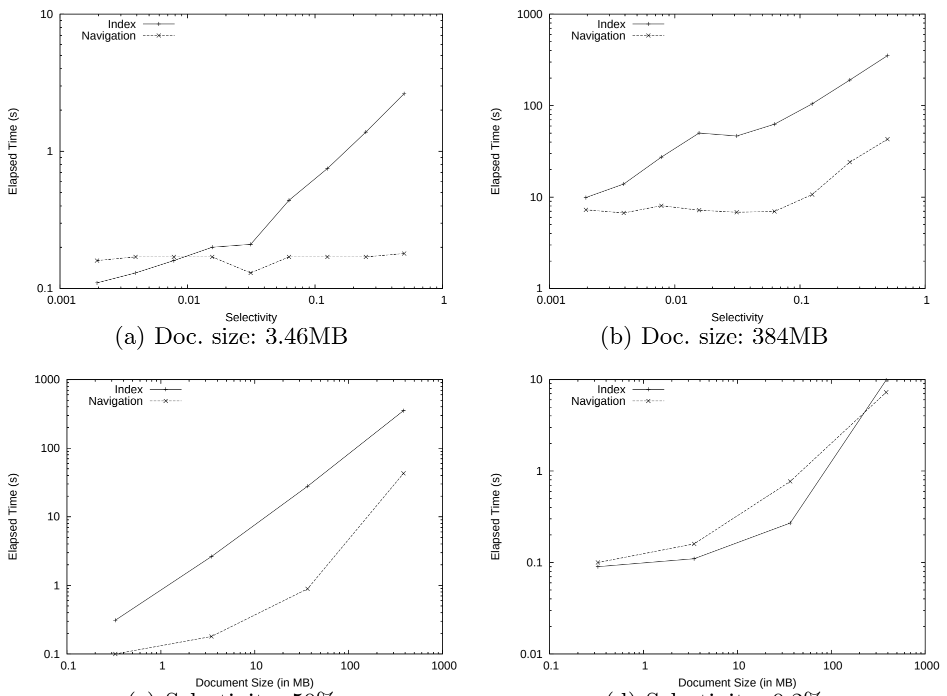
Q3: /DOC/descendant::TAG1/descendant::TAG2. In addition to query Q2, the cost of evaluating each step in this query is potentially much higher because level information is less useful here. As both descendant axis steps potentially visit large parts of the document, we expect optimizations that can reduce the I/O involved here to be very important.

We restrict ourselves to the child axis and the descendent axis because only for these two axes precise selectivity estimation techniques are known, e.g. [30, 26]. To make our experimental results comprehensible, we ignore the other XPath axes because we cannot easily compute the selectivity of an axis step with respect to some arbitrary context node.

5.3 Experimental Results

Query Q1. Fig. 8 shows the results for query Q1. In Figs. 8(a) and 8(b) we compare the performance of the navigational plan and the index-based plan for two document sizes, i.e. 3.46MB and 384MB.

For small selectivities on the smaller document, the index-based plan performs better than the navigational plan. As we make the node test less selective,



(a) Doc. size: 3.46MB (b) Doc. size: 384MB

(c) Selectivity: 50% (d) Selectivity: 0.2%

Fig. 8. Query Q1 (/descendant::TAG)

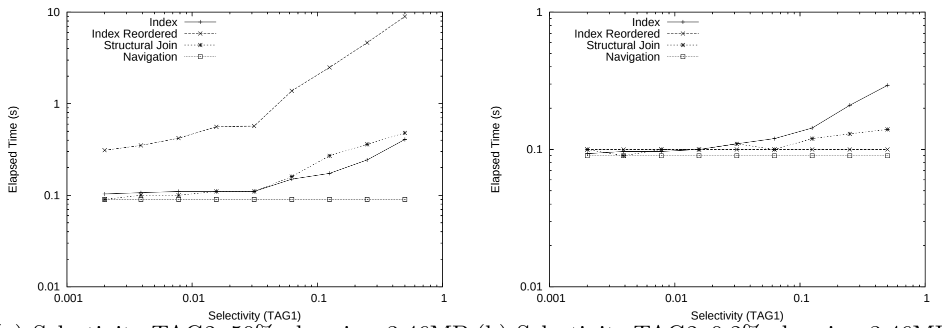
the index-based approach needs more time to evaluate the query while the execution time of the navigational plan remains nearly constant. The break-even is reached at a selectivity of about 1%. For larger selectivities, the navigational plan outperforms the index-based plan. All these results agree with our experience in the relational world.

In Figs. 8(c) and 8(d) we plot the query execution times for specific selectivities of 50% and 0.2% over varying document sizes.

Again, the results confirm that index-based evaluation is superior only for selective queries. However, since two indices and the document are accessed, more buffer pages have to replaced due to lack of space, and additional I/O is needed. As a result, the performance of the index-based plan suffers on the largest document instance.

Query Q2. Fig. 9 contains the results of our experiments for query Q2. We restrict ourselves to two document sizes (3.46MB and 384MB) because the results on the other documents do not provide any additional insight. In our exposition we keep the selectivity of the second location step (`TAG2`) constant at 50% (see Figs. 9(a) and 9(c)) and at 0.2% (see Figs. 9(b) and 9(d)). We only modified the selectivity of the first location step (`TAG1`).

The navigational QEP has an almost constant execution time independent of the selectivity. This is a direct consequence of its evaluation strategy: this


(a) Selectivity TAG2: 50%, doc size: 3.46MB (b) Selectivity TAG2: 0.2%, doc size: 3.46MB

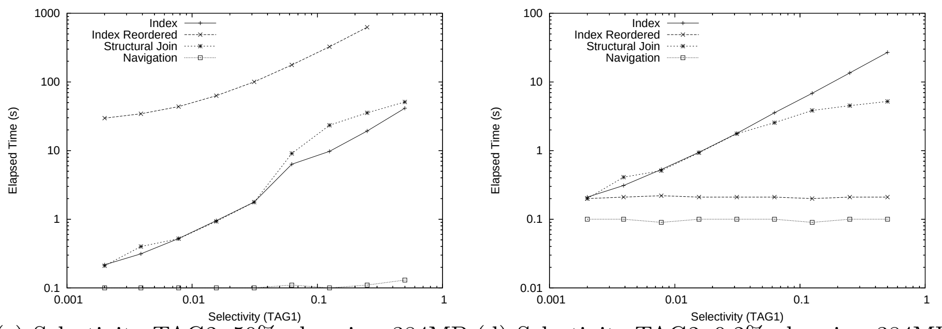
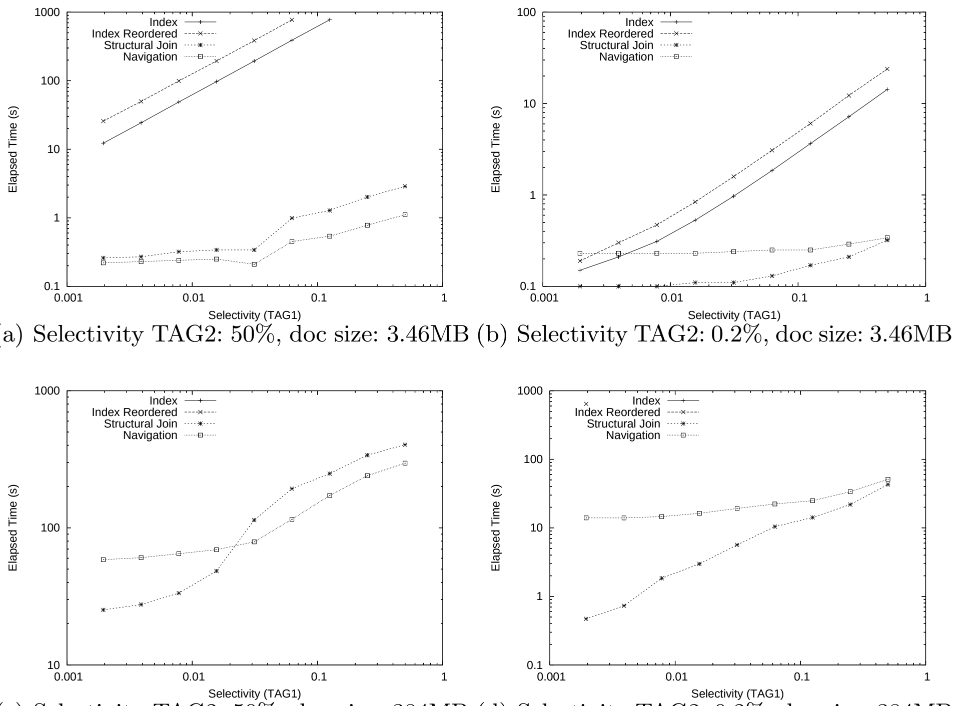

(c) Selectivity TAG2: 50%, doc size: 384MB (d) Selectivity TAG2: 0.2%, doc size: 384MB

Fig. 9. Query Q2 (/DOC/TAG1/TAG2)

QEP has to inspect the same set of nodes to compute its result, no matter how selective each step is. The navigational QEP dominates all other QEPs because it effectively prunes the document fragments that are not relevant for the query result. This holds true particularly for the deeply nested document where this plan only visits the three upper levels of the document.

The value of reordering location steps becomes apparent when we compare the execution times of the naive and the reordered version of the index-based plans. In the experiments depicted in Figs. 9(a) and 9(c), the reordered version is up to ten times slower than the naive plan because the reordered QEP performs the more expensive scan first (selectivity 50%). In Figs. 9(b) and 9(d) we observe exactly the reverse behavior because the second location step is very selective (selectivity = 0.2%). However, the differences are smaller because the naive plan can use more information to restrict the index scan. In all experiments the Structural Join behaves similar to the naive index-based evaluation technique. The index based technique is competitive because none of the navigation steps produces duplicates. Hence, no redundant index lookup is performed.

The advantage of the navigational plan is partially a consequence of the document structure. For shallow documents where we increase the number of children per node, we expect similar behavior as for query Q3.



(a) Selectivity TAG2: 50%, doc size: 3.46MB (b) Selectivity TAG2: 0.2%, doc size: 3.46MB

(c) Selectivity TAG2: 50%, doc size: 384MB (d) Selectivity TAG2: 0.2%, doc size: 384MB

Fig. 10. Query Q3 (/DOC/descendant::TAG1/descendant::TAG2)

Query Q3. For our final experiment, we replace the last two child steps of Q2 by descendant steps: `/DOC/descendant::TAG1/descendant::TAG2`. The execution times for these three alternatives are shown in Fig. 10. We present the results for the same selectivities and document sizes as the figures for query Q2 do.

First, we discuss the navigational approach: We observe that in all figures the evaluation times of the navigational QEP increase with an increasing selectivity of the name tests. The reason for this is that the XPath expression generates more context nodes for which the whole subtree is traversed. For larger selectivities, fewer subtrees are pruned during the traversal.

In contrast, the index-based QEPs retrieve exactly the candidate nodes with the correct tag name. However, only in Fig. 10(b) both index-based strategies are faster than navigation for very small selectivities on a small document. The reason for this is that our naive index-based execution strategy is not aware of structural relationships of context nodes. As a result, the same nodes are returned repeatedly by index lookups. We conclude that the simple index-based QEPs do not result in an acceptable query performance.

Fortunately, the plan based on Structural Joins avoids these superfluous index lookups. For large selectivities (of 50%) of the second step, the Structural Join is faster when the selectivity of the first step is smaller than about 2% and the document is large (Fig. 10(c)). However, when we keep the selectivity of the second step at 0.2%, the Structural Join clearly outperforms the navigational query independent of the selectivity of the first step and the document size. This advantage is larger when the selectivity of the location steps are small.

6 Conclusion and Future Work

There are many possibilities to optimize XPath expressions. Several indexing and evaluation techniques have been proposed. However, we still lack cost-based optimizers that compare alternative query evaluation plans based on costs. As a starting point, we have compared two basic evaluation techniques: navigation-based and index-based XPath evaluation.

Our experiments show that there is no overall winner: Each technique has its individual strengths and weaknesses. In general, navigation is a good choice when large parts of the input document belong to the query result or when navigation can avoid visiting large parts of the document. Unfortunately, there is no simple rule where the break-even point between both alternatives lies. However, a selectivity of about 1% to 10% for elements of a given tag name still seem to be decisive when navigation must be traded against index usage. For space reasons, we could only compare a limited number of evaluation techniques. Certainly other query evaluation techniques should be included into benchmarking, e.g. see [23].

In previous work, sophisticated cardinality estimation techniques were developed for the child axis and the descendent axis. We need to extend these results and the results of this paper to include cost information. Using statistics or analyzing the query processing algorithms are possibilities to achieve this goal. We then could integrate XQuery processing techniques into cost-based query optimizers and overcome using heuristics as is currently the case.

Acknowledgements We would like to thank the anonymous referees who noticed a weak spot in our experiments and thus helped us to improve our query evaluation plans. We also thank Simone Seeger for her comments on the manuscript.

References

1. M. Brantner, C-C. Kanne, S. Helmer, and G. Moerkotte. Full-fledged algebraic XPath processing in Natix. In *Proc. ICDE*, pages 705–716, 2005.
2. S. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proc. VLDB*, pages 263–274, 2002.
3. E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic XML trees. In *Proc. of the ACM PODS*, pages 271–281, 2002.
4. A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proc. of the ACM SIGMOD*, pages 431–442, 1999.
5. A. Halverson et.al. Mixed mode XML query processing. In *Proc. VLDB*, pages 225–236, 2003.
6. D. Srivastava et.al. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. ICDE*, pages 141–152, 2002.

7. F. Ozcan et.al. System RX: One part relational, one part XML. In *Proc. of the ACM SIGMOD*, pages 347–358, 2005.
8. H. Jagadish et.al. Timber: A native XML database. *VLDB Journal*, 11(4):274–291, December 2002.
9. P. E. O'Neil et.al. ORDPATHs: Insert-friendly XML node labels. In *Proc. of the ACM SIGMOD*, pages 903–908, 2004.
10. S. Pal et.al. XQuery implementation in a relational database system. In *Proc. VLDB*, pages 1175–1186, 2005.
11. T. Fiebig et.al. Anatomy of a native XML base management system. *VLDB Journal*, 11(4):292–314, December 2002. available at: http://db.informatik.uni-mannheim.de/natix.html.
12. R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proc. VLDB*, pages 436–445, 1997.
13. G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.
14. J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Record*, 26(4):63–68, 1997.
15. T. Grust. Accelerating XPath location steps. In *Proc. of the ACM SIGMOD*, pages 109–120, 2002.
16. T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational DBMS to watch its (axis) steps. In *Proc. VLDB*, pages 524–525, 2003.
17. S. Helmer, C-C. Kanne, and G. Moerkotte. Optimized translation of XPath expressions into algebraic expressions parameterized by programs containing navigational primitives. In *Proc. of WISE*, pages 215–224, 2002.
18. J. Hidders and P. Michiels. Avoiding unnecessary ordering operations in XPath. In *DBPL*, pages 54–74, 2003.
19. Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proc. of the ACM SIGMOD*, pages 268–277, 1991.
20. I. Jaluta, S. Sippu, and E. Soisalon-Soininen. Concurrency control and recovery for balanced B-link trees. *VLDB Journal*, 14(2):257–277, 2005.
21. Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proc. VLDB*, pages 361–370, 2001.
22. N. May, S. Helmer, C-C. Kanne, and G. Moerkotte. XQuery processing in Natix with an emphasis on join ordering. In $< XIME - P/ >$, pages 49–54, 2004.
23. P. Michiels, G. A. Mihăilă, and J. Siméon. Put a tree pattern in your algebra. Technical report, Univ. of Antwerp, TR-06-09, Belgium, 2006.
24. K. Ono and G. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proc. VLDB*, pages 314–325, 1990.
25. A. Pellenkoft, C. Galindo-Legaria, and M. Kersten. The complexity of transformation-based join enumeration. In *Proc. VLDB*, pages 306–315, 1997.
26. N. Polyzotis and M. Garofalakis. XCluster synopses for structured XML content. In *Proc. ICDE*, pages 406–507, 2006.
27. C. Re, J. Simeon, and M. Fernandez. A complete and efficient algebraic compiler for XQuery. In *Proc. ICDE*, pages 138–149, 2006.
28. F. Tian, D. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of alternative XML storage strategies. *SIGMOD Record*, 31(1), 2002.
29. Y. Wu, J. Patel, and H. Jagadish. Structural join order selection for XML query optimization. In *Proc. ICDE*, pages 443–454, 2003.
30. N. Zhang, T. Özsu, A. Aboulnaga, and I. F. Alyas. XSeed: Accurate and fast cardinality estimation for XPath queries. In *Proc. ICDE*, pages 168–179, 2006.