Benefits of Path Summaries in an XML Query Optimizer Supporting Multiple Access Methods

Attila Barta
Dept. of Computer Science
University of Toronto
10 King's College Rd., M5S
3G4, Toronto, ON, Canada
atibarta@cs.toronto.edu

Mariano P. Consens
Information Engineering, MIE
University of Toronto
5 King's College Rd., M5S
3G8, Toronto, ON, Canada
consens@cs.toronto.edu

Alberto O. Mendelzon
Dep. of Computer Science
University of Toronto
10 King's College Rd., M5S
3G4, Toronto, ON, Canada
mendel@cs.toronto.edu

Abstract

We compare several optimization strategies implemented in an XML query evaluation system. The strategies incorporate the use of path summaries into the query optimizer, and rely on heuristics that exploit data statistics.
We present experimental results that demonstrate a wide range of performance improvements for the different strategies supported. In addition, we compare the speedups obtained using path summaries with those reported for index-based methods. The comparison shows that low-cost path summaries combined with optimization strategies achieve essentially the same benefits as more expensive index structures.

1. Introduction

Over the past few years the Extendible Markup Language (XML) has become the dominant data format for information exchange. With the proliferation of data in this format comes the motivation to query and manipulate XML documents. XQuery and its widely adopted XPath subset constitute the predominant proposal for a native XML query language standard.

The XQuery related work is extremely diverse. It ranges from native XML databases (e.g. Timber [16], Niagara [15], Natix [12], BEA/SQRL [13], ToX [4]), to XQuery systems (e.g. Galax [20]) and work that addresses certain aspects of XQuery processing such as XPath, twig queries and index structures for XML query evaluation. Moreover, because extensive work in the area addresses only some aspects of XQuery evaluation, for the rest of the paper we will use the XQuery and XML query terms interchangeable.

An important aspect of XML query processing is the encoding used. That is, in a native XML system, XML documents can be pre-parsed into special purpose data structures in order to speedup query execution. The most employed such data structure relies on an element encoding derived from the notation used in region algebras [9]: an inverted-file-like structure with element name, start, end and level. Utilizing this type of data structures for evaluating path expressions requires joins between lists of encoded elements, referred to as *containment queries* [33] or *structural joins* [3]. Also based on region encoding are the *stack algorithms* that improve on structural joins by using stacks for intermediate results. In this context, *PathStack* is proven to be optimal for processing single path queries [7]. Because, in stack algorithms the region algebra encoded elements are treated as streams, for the remain on this paper we will refer to XML documents encoded in these structures as *element encoded streams*, for elements and attributes, and *value streams* for CDATA.
An XQuery expression frequently contains several XPath sub-expressions, such as the example given in Figure 1.

```
for $x in document("file:/supplier.xml")//supplier,
    $y in document("file: /catalog.xml")//item
where $x/supplier_no = $y/supplier_no and
        $x/city  = "Newark" and  $x/state = "New Jersey"
return  <result> { $y/name } { $y/description }  </result>
```

Figure 1: Sample XQuery expression

The example query joins a *supplier* document and a *catalog* document based on *supplier_no*, returning the *name* and *description* of items in the catalog for those suppliers located in Newark, New Jersey. The XPath expressions that occur in the query above and apply to the supplier document are: *'//supplier'*, *'//supplier/city'*,

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

'//supplier/supplier_no' and *'//supplier/state'*. A fragment of a sample supplier document is shown in Figure 2.

Although each of the XPath expressions in an XQuery query, such as the example above, can be computed separately, a much better approach is to group XPath queries into a so called *twig query*, known also as *pattern tree*. The group of XPath expressions that apply to a certain document (such as *suppliers* in the example above) can be computed in a single access method invocation through the document. Most of the XPath processors as well as structural join algorithms support this approach [e.g. 7, 18]. The most cited approach for twig query processing is *TwigStack* [7] a generalization of PathStack from one path to a twig query. TwigStack can be further improved by using index structures (XB trees) to index the encoded element stream and hence supporting the skipping of irrelevant nodes. The index-based algorithm is known as *TwigStackXB* [7].

```
<suppliers>
    <supplier> <supplier_no> 1 </supplier_no>
        <name> Gore </name>
        <city> Newark </city> <state> Delaware </state>
    </supplier>
    <supplier> <supplier_no> 2 </supplier_no>
        <name> Dupont </name>
        <city> Newark </city> <state> New Jersey </state>
    </supplier>
</suppliers>
```

Figure 2: Sample *supplier.xml* document

Building the encoded element data structures is not a computationally expensive process - requiring usually one and at most two passes over the document. Recently, novel *node index* structures designed to evaluate twig-queries and not only single path queries have been proposed, such as ViST [31] and PRIX [28]. Node index construction requires extensive pre-processing, but in return, these indexes can yield significant speedups when compared with access methods such as TwigStack. ViST uses a prefix tree and virtual tries stored on disk as B-trees to achieve this improved performance. Similarly, PRIX also employs tries stored on disk while using an encoding based on Prufer sequences.

Path summaries are another important work in XML query processing. A *path summary*, also known as a *structural summary* or as a *path index*, represents a summarization of the paths that actually occur in a document. That is, for each distinct path in an XML document there is a distinct path in the path summary [e.g. 14, 24, 30] or an approximation of it [e.g. 19, 27]. Path summaries are used as back-ends that is, the XML query is evaluated by traversing the path summary alone. Each path summary implementation uses a particular strategy to evaluate the query. The common denominator of these strategies is that they use path pruning in order to limit the search space. Moreover, the majority of path summaries only support individual path expressions evaluation that is, no twig query support.

The most work in XML query optimization is in the context of the native XML databases. Thus, the Timber optimizer chooses the best order for the structural joins [2]. The Niagara optimizer incorporates a cost-model for evaluating path expressions [15]. Systems like Natix [12] and BEA/SQRL [13] use built in rules for optimization. However, to the best of our knowledge, the most complex optimizer up-to-date is the Lore optimizer which considered different evaluation strategies for each branch in an XML tree, as well as an aggressive plan pruning strategy in order to reduce the search space [23]. There are query optimization techniques in path summaries too. For instance, query re-writing [11] and path expansion [22] were also proposed for optimizing the evaluation of path summaries. A somehow similar approach is projecting the XML document according to the path expressions [21]. Also relevant are query evaluation techniques based on summaries and compression [34, 35].

In this paper we focus on strategies for XML query optimization. These strategies are presented in the context of *two-level optimization* that we proposed in [6]. In the two-level optimization strategy, the higher level consists of the traditional join order selection together with the cost-based selection of access methods. The lower level consists in a cost based selection of evaluation plans for XPath expressions, i.e. twig queries. Although, separating the access method selection from the join order selection, was also proposed in the relational mode, e.g. "multi-level" or "multi-faceted" optimization, in the XML context there is a major difference. That is, in the relational model each access method has a fixed cost and the optimizer is aware of it, while in the XML model, due to the XPath expressions that are embedded into the access methods, the access methods do not have fixed costs, rather the cost varies with the XPath evaluation strategy. This observation suggests an additional layer of optimization at the access method level, hence the two-level optimization model.

The two-level optimization and the optimization strategies that we present in this paper are incorporated in ToXop, a query optimizer that is part of ToX, a native XML database under development at University of Toronto [4].

In this paper we extend the work presented in [6] and we describe an approach to XML query optimization that incorporates several novel characteristics. The main contributions of this paper are:

- We propose the usage of path summaries into the optimizer in order to exploit schema information. In this respect we split the path summaries into a schema part and a node instance part. The schema part is used for optimization while the node instance part is used as back-end.
- We propose two novel optimization strategies: holistic path summary pruning and access-order

selection. The former reduces the plan search space by identifying early the portions of the document that will be part of the answer. This approach differs from the traditional path summary pruning by separating the path pruning from the query evaluation per se. Consequently, the holistic path pruning can be used outside a path summary, e.g. with an element encoded stream. The latter optimization strategy uses data statistics and (simple) cost-based heuristic in order to compute an efficient plan.

- We present experimental results that establish the benefits of the optimizer-driven early pruning of path summaries as well as substantial benefits for the heuristic based optimization. The speedup attributable to these optimization strategies is of the same order of magnitude as the speedup obtained by querying node indexes. This is a surprisingly positive result, considering how much cheaper (in storage and index construction costs) path summaries are compared to node indexes.

The paper's contributions can be summarized as follows: using path summaries, simple statistics and simple cost-based heuristics we can achieve speedups in the same order of magnitude as more "heavy" index based systems. In summary: with little (effort to preprocess XML data) you can achieve a lot (of performance improvement in query processing)!

We note some connections between the work that we present in the XML context with earlier work in query optimization for Object Oriented Databases (OODB) [8, 10]. For instance, Access Support Relations (ASRs) provide an indexing mechanism for paths in the same manner as XML path summaries. Despite the similarities, in the OODB context the schema information is known, while this is frequently not the case in the XML context. Moreover, ASRs may cover only some paths from the database instance, while in the XML context the structural summaries cover all paths from the document instance.

Using path summaries in the query evaluation is not entirely new: the approach was also proposed in the context of the Lore system [23]. The path summaries used by ToXop and DataGuides [14] (the path summaries used in Lore) are essentially the same structure (when considering XML trees). However, the Lore system evaluated queries using DataGuides in combination with other additional index structures. Furthermore, the stack-based evaluation algorithms for twig queries were not considered at the time when the optimization techniques for Lore were proposed. Hence one of our contributions is a novel combination of all of these proposed techniques

In the following section we introduce the usage of path summaries in the query optimizer. Section 3 describes briefly the ToXop optimizer. Section 4 and 5 introduce the holistic path summary pruning and access-order selection strategies. Section 6 contains experimental results. We conclude by mentioning future research in Section 7.

2. Path summaries into the query optimizer

The path summaries proposed in the literature can be classified as exact path summaries [e.g. 14, 24, 30] or approximate path summaries [e.g. 19, 27]. The exact path summaries record each distinct path in the XML document while approximate path summaries record only an approximation of the paths, usually paths up to a certain depth. The common denominator of both categories is that they are built for evaluating regular path expressions. That is, path summaries are used as back-ends. However, in the XML query context the exact path summaries can be used not only as back-ends but as existing schemas for a given XML document or collections of documents. By existing schema we understand the underlying structure of the document, rather than its DTD or XML Schema. The concept of existing schema in the XML model is similar with the concept of schema in the relational model.

Because schema information is essential to query optimization and because exact path summaries are exiting schemata, we propose the incorporation of the exact path summaries into the optimizer. In this respect we use ToXin [30], an exact path summary as the existing schema in ToXop, our query optimizer. Both ToXin and ToXop are part of ToX a native XML database [4].

The ToXin path summary mirrors the structure of the document thus, for each element type or attribute type there is at least one ToXin node to represent it. The case when an element generates more than one ToXin node is encountered when the element is part of two distinct paths. In addition there are text nodes, which are generated from the text content of the elements. For each ToXin node there is an *Instance Table*, which records the occurrences of the element or attribute, as well as its parent instance. Moreover, text nodes and attribute nodes also have a *Value Table*, which records the content of each node. Each ToXin node has a node ID obtained in depth first (pre-order) traversal.

The majority of path summaries are tied to a particular path expressions evaluation strategy. In the ToXin case this evaluation is bottom-up and works as follows: first, the given path expression is checked if it matches any path from the ToXin tree. Second, the corresponding Instance and Value Tables are selected. Next, the evaluation process is performed in a bottom-up manner. Thus, first the predicates are evaluated against the Value Table. Then, for the selected records in the Value Table the corresponding records from the Instance Table are selected and so on until the root is reached. The advantage of using this strategy occurs in the case of selective predicates on one of the leaf nodes, thus only a few number of parent nodes have to be evaluated.

Most path summaries, like ToXin, are designed to evaluate individual path expressions. However, in the presence of several path expressions to be evaluated in one pass (i.e. twig queries), a particular evaluation might

not be the most efficient, thus defying the role of a query optimizer. With this observation in mind, we added an additional structure to ToXin, namely navigational tables, NAV for short, in order to support top/down navigation. As a result of this enhancement, top-down evaluation strategy can now be taken into consideration by a query optimizer. This approach was also considered by the Lore optimizer, where each individual path in a XML tree could have been evaluated using either a top-down or a bottom-up strategy. However, considering all possible combinations is extremely costly, and even employing an aggressive plan pruning technique, as proposed in the Lore optimizer, the number is still significant. In this respect in section 4 and 5 we present simple heuristics that reduce the search space considerably and (proved by the experimental data) still produce efficient query plans.

To this point, ToXin provides schema information and multiple evaluation strategies; however it misses one important component for query optimization, namely statistics. There is extensive work in collecting the appropriate statistics for XML documents [e.g. 1, 26, 32], however the statistics that we use are rather simplistic nevertheless expressive enough to sustain the optimization strategies that we employ. The statistics that we collect for each element are: number of instances for the element (NCARD), number of distinct values for the element (ICARD) and fan-out (Fout) – average number of sub-element instances for each sub-element.

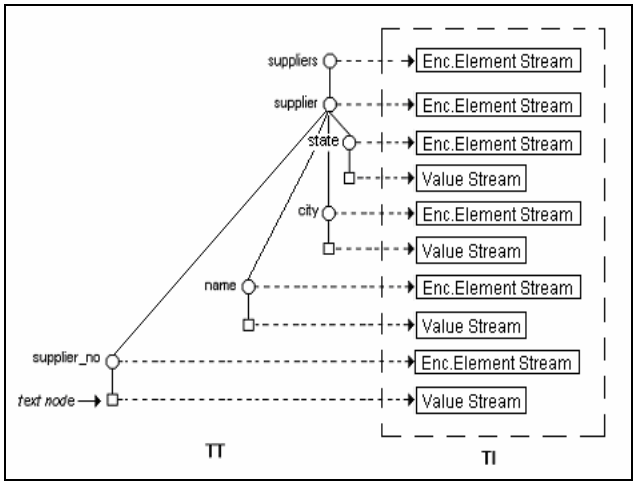

Figure 3: A ToXin for the *supplier.xml* document

ToXin, similar to other path summaries, was designed to be used as a back-end. Nevertheless, because ToXin reflects the structure of the document we propose the usage of ToXin as the existing schema for the document. However, keeping all data from node instances might be a burden. This is the reason why we divide the ToXin structure into two components: the ToXin summary tree (TT) and the node instance data structures (collectively, TI). According to this split, the TT is the existing schema of the document and given that it is augmented with statistics we can use the TT as system catalog, in a similar manner with the usage of system catalogs in relational query optimization.

Up to this point, the TI structures contain the Instance, NAV and Value tables, which are the original ToXin data structures. However, these data structures can be generalized. That is, instead of storing the node instances in a certain encoding we can use any encoding. For instance, the node instances can be stored in a region algebra encoding. Consequently, as presented in Figure 3, we use the more generic term of Encoded Element Stream and Value Stream for the encodings that we use for element/attributes and values.

To conclude the description of the ToXin trees that we use: ToXin trees employed in ToXop are augmented with statistical information about the XML document that they summarize and have their structure split into two components, the tree part itself, called TT, and the XML node instance information, called TI, which can be encoded according to the processing algorithms used.

3. The ToXop query optimizer

ToXop is the query optimizer in ToX [4]. ToXop was designed with relational query optimization in mind in order to exploit the benefits of decades of research in this area. ToXop has two sets of operators, logical operators and physical operators, and an open optimization technique, which permits different optimization strategies to be plugged-in. The logical operators are the Tree Algebra for XML (TAX) algebra operators, while the physical operators are back-end specific (i.e. for data access) or implementation specific (i.e. a join operator can be implemented as a merge-join or as a double pipelined-join.) In this paper we concentrate only on the ToXop physical operators, moreover on the ToXop access methods and the optimization strategies that they employ.

The logical algebra used in ToXop is essentially the TAX algebra [17]. TAX is also the logical algebra employed by Timber [16]. However, as we will describe below, ToXop relies on different access methods than Timber and employs a novel optimization approach that exploits structural summaries instead of using structural joins. In the TAX algebra, each logical operator L takes as input a collection of trees or a document D and a pattern tree PT and outputs a collection of n witness trees:

$L(D, PT) = [WT^1, WT^2, \ldots , WT^n]$

The witness trees are those sub-trees that satisfy the pattern tree. Because each collection of trees can be transformed into one document (by adding a virtual root), in this paper we treat a document D and a collection of trees as the same.

As an example, consider the TAX algebra expression below, which is a translation of the query in Figure 1:

Projection *//item/name, //item/description*
(Join *//item/supplier_no = //supplier/supplier_no* *(*
Selection('supplier.xml', PT1),
Selection('catalog.xml', PT2)))

Where *PT1* and *PT2* are pattern trees, a concatenation of XPath expressions augmented with value predicates. In Figure 4 we present the pattern tree *PT1* associated with variable *$x* from the query in Figure 1. *PT1* is obtained from concatenating the four XPath expressions: *//supplier*, *//supplier/supplier_no,* *//supplier/city,* and *//supplier/state.*

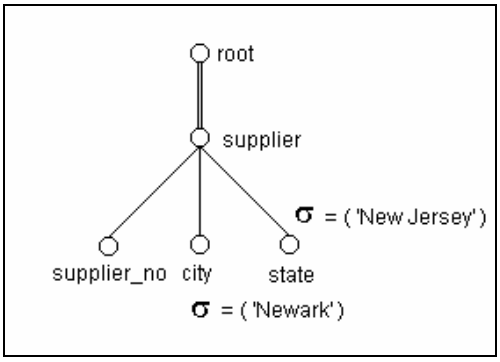

Figure 4: The pattern tree *PT1*

The query optimization strategy in ToXop is likewise open. That is, any query optimization paradigm can be plugged-in. In the current implementation we employ the two-level optimization strategy (see section 1), in which the lower level uses the optimization strategies that we present in this paper, while the upper level (the join order selection) uses a System R like optimization strategy.

Finally, a note with regard to TAX and the optimization strategies that we present in this paper. Although, we present these strategies in the context of TAX, these strategies can be applied in the context of any XML query algebra that recognizes twig queries, i.e. pattern trees in the TAX parlance.

4. Holistic path summary pruning

In this paper we propose two query optimization strategies, namely holistic path summary pruning and access-order selection. Path summary pruning is by no means novel; any path summary performs certain path pruning as part of the path expression evaluation. The difference between the holistic pruning and the traditional one is that the traditional path summary pruning is intrinsic to a particular path summary evaluation method, while the holistic path summary pruning is generic, thus any path summary evaluation method can be used afterwards. For instance, both TwigStackScan and ToXinScan (two of the ToXop access methods that we describe later) use holistic path summary pruning as their first evaluation step while performing a distinct second evaluation step according to the encoding that each employs.

Because holistic path pruning is generic, it can be used in conjunction with any path evaluation algorithm in order to reduce the search space. For instance, consider the stack algorithms (i.e. PathStack, TwigStack). The stack algorithms work on region algebra encoded element streams. For each element that appears in the query, the algorithms will load the encoded element stream corresponding to the element. If an element referred in the query occurs in a document within different distinct paths, any stack algorithm will load and probe the streams from all distinct paths where the element occurs, despite the fact that only one of these streams contributes to the answer. To exemplify this behavior we use a more complex example than the ones in the sections above. In Figure 5a we present a fragment from the DBLP data set [25] with the *author*'s text value removed for simplicity.

The twig query from Figure 5b is induced by the following query *'//inproceedings[/author="Jim Gray"] [/year="1990"]'*. That is, we retrieve the entire *inproceedings* element where the *author* is '*Jim Gray*' and the *year* is *'1990'*. In Figure 5c we present a fragment from a path summary for the DBLP data set. We use the neutral notation *R_a1, R_a2,* etc. to represent any encoding that might be used to store author information in the path summary. For instance, information for the author *a1* might be encoded in the TI structures described in section 2, or it might be represented by a region algebra encoding similar to the one used by the stack algorithms. In Figure 5d we present two fragments of encoded element streams associated with the *author* (T_{author}) and *year* (T_{year}) elements, as used by the stack algorithms.

Note that *author* and *year* appear in two structurally different sub-trees, i.e. they appear in *article* sub-trees as well as *inproceedings* sub-trees. The advantage of having a path summary is that we can distinguish between these different occurrences of *author* and *year*. In Figure 5c we highlight this information by drawing in a thick line to the parts of the path summary where the target elements lie. In contrast, a stack algorithm will load all the *year* and *author* encoded-element streams for all possible parents, not only for *inproceedings.* This is illustrated in Figure 5d by the authors *a3* and *a4* and the year *y3*, which are descendents of *article* but not *inproceedings,* but are in the streams nonetheless.

From the example above it results that an access method that incorporates a stack algorithm could exhibit significant improvement by using holistic schema pruning. In section 6 we present experimental data that suggests that, when applicable, holistic path summary improves TwigStack by an order of magnitude.

The ToXop access methods that employ the holistic path summary pruning are the SumScan access methods. SumScan is a generic access method that operates on ToXin path summary structures. In ToXop we have two implementation of SumScan, namely TwigStackScan and ToXinScan. SumScan is a ToXop access method, thus it takes as input a document D (where TT_D and TI_D are available) and a pattern tree PT, and outputs a sequence of witness trees that satisfy the pattern tree PT. The signature of the SumScan operator is:

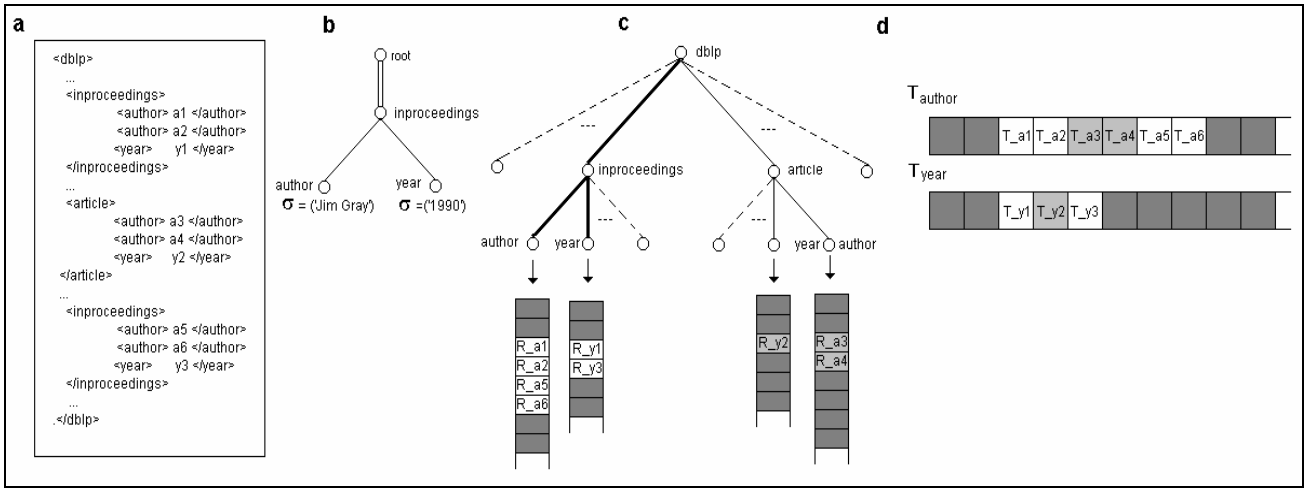

Figure 5: a) A fragment from a DBLP document; b) a twig query; c) the ToXin tree for the document fragment; d) element encoded streams for the document

$$SumScan((TT_D, TI_D), PT) = Scan(PruneToXinTree((TT_D, TI_D), PT)) = [WT^1, WT^2, \dots , WT^n]$$

SumScan is the result of composing two operators: *PruneToXinTree* and *Scan*. As the name suggest, PruneToXinTree is the operator that prunes the path summary while Scan is an implementation specific operator that evaluates the query on the pruned path summary or element encoded stream or any other encoding used. Consequently, the implementation of Scan determines the flavor of the SumScan implementation, for instance TwigStack is the Scan implementation for TwigStackScan access method.

In order to present how PruneToXinTree works we have to introduce the notion of matched ToXin tree. For a given pattern tree PT and a ToXin tree Tx, we call *matched ToXin Trees* (*MTT*) those sub-trees of Tx that satisfy the pattern tree *PT*. Moreover, the nodes of the *MTT* are adorned with the corresponding node selection predicates from the pattern tree *PT*.

Therefore, PruneToXinTree takes a pattern tree PT and a ToXin tree, noted (TT_D, TI_D), and outputs the k matched ToXin trees, noted as $MTX^i_{D,PT}$, that satisfy the pattern tree PT:

$$PruneToXinTree ((TT_D, TI_D), PT) = [MTX^1_{D,PT}, MTX^2_{D,PT}, \dots , MTX^k_{D,PT}]$$

In Figure 6 we present the result of pruning the ToXin tree generated from the document from Figure 2 according to the pattern from Figure 4.

An MTT node is a data structure that contains: a pointer to an encoded element stream (Instance and NAV Tables for ToXinScan, region-algebra encoded streams for TwigStackScan); a pointer to a value stream (Value Tables for ToXinScan, region-algebra encoded streams for TwigStackScan); statistical information and selection predicates. True to the ToXin inheritance, the MTT structure is also split into a tree structure TT and a data structure TI. The advantage of having the structure split is that the pruning operation is executed only on the TT tree, thus on a very small data structure. After the pruning is completed, only the corresponding TI data structures are retained.

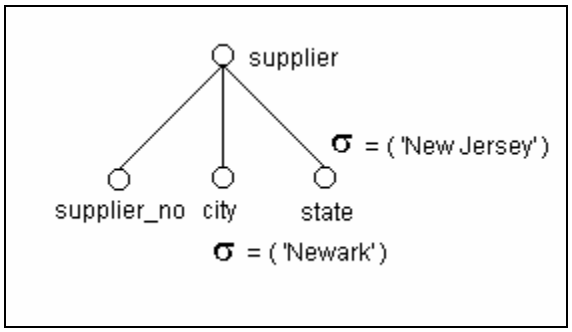

Figure 6: A matched ToXin tree (MTT)

5. Access-order selection

The access-order selection optimization strategy works in conjunction with the holistic path summary pruning and it is based on the observation that paths in XML documents can be computed using either bottom-up evaluation or top-down evaluation. The idea behind the access-order selection strategy is to use heuristics based on data statistics to determine which is the proper evaluation to employ. We have to note here that this approach is not novel. The Lore query optimizer also addressed this same problem [23]. However, the Lore optimizer used a "heavy" approach. That is, the Lore optimizer inserted a Glue operator at each branch of the XML document in order to determine the order of the evaluation, which resulted in a large query plan search space. For instance, as reported in [23] for a document with level 7, the total number of possible plans is in the range of 8 billion, while using the Lore pruning strategy this number was reduced to 948, which is still a large number. Using our heuristic

we have a plan right away and, as we present in section 6, using this plan we obtain a performance comparable with the state of the art.

Another difference from the Lore optimizer is that we have already reduced the search space by using the holistic path summary pruning. Consequently, we have to compute plans only for the MTTs and not for the entire ToXin tree. Computing the plan comprises by: computing the right order selection and the right direction selection. We present both of these concepts by means of example. For instance, for the MTT from Figure 6, in order to retrieve all nodes that satisfy the selection predicates we can proceed as follows. First we evaluate the path *'/supplier/state'* then *'/supplier/city'* then *'/supplier/supplier_no'*. Finally, we intersect all instances of the *'supplier'* node obtained from the evaluation of the paths and the predicates. Evidently, this is an inefficient method of evaluating the tree. The reason is that we have two predicates on nodes *state* and *city*. Consequently, there is a possibility that one of these predicates has higher selectivity. In this case, we would like to evaluate the higher selectivity path first and then evaluate the next path only for those instances of the *supplier* node that satisfy the first path. We call this process the *right order selection* and it is the first part of the access order selection.

The second part of access order selection constitutes the *right direction selection.* Assume that we evaluate first the path *'/supplier/state';* this means that we have a set of *supplier* nodes for which there are child *state* nodes that satisfy the predicate on *state.* The next step is to evaluate the path '*/supplier/city*'. There are two options. The first one is to use a bottom-up evaluation and intersect the *supplier* nodes selected by the '*/supplier/city*' path with those supplier nodes selected by the *'/supplier/state'* path. The second approach is to perform a top-down evaluation. A top-down evaluation works as follows: for those nodes supplier that satisfy the *'/supplier/state'* path, we select their corresponding *city* children nodes and then for these *city* nodes only we evaluate the predicate on *city*.

The *right order selection* and *right direction selection* are part of the access-order selection optimization strategy. In order to compute them, one approach is to exhaustively enumerate all possible combinations of order and direction among the MTT edges. Nevertheless, this approach is time consuming. Consequently, in the tradition of relational query optimization, heuristics can be used. In order to present these heuristics, first we have to introduce a number of terms.

Definition 1: in the context of a MTT, given a node *n* and a set of predicates *S* attached to the node *n*, we call *node selectivity factor* for node *n*, noted F_n, the expected fraction of instances of the node *n* that satisfy the predicate set *S*.

Node selectivity factor is a similar concept with the *selectivity factor* in System R. For instance, in the case when the predicate set *S* is constituted by a single equality predicate and assuming a uniform distribution of the values of the node *n* (similar to System R) then: $F_n = 1/ICARD_n$, where $ICARD_n$ is the number of distinct values in the Value Table associated to the node *n*.

The choice of a model to the node selectivity factor is orthogonal to this work. However, the selectivity model that we employ in ToXop can be found in [5].

Using the selectivity factor, the number of instances of node *n* that satisfy the predicate set S, noted $N_{n,S}$, can be computed as follows: $N_{n,S} = F_n * NCARD_n$, where $NCARD_n$ is the total number (cardinality) of the *n* elements.

Definition 2: in the context of a MTT, assume a node *p* and a node *n*, such that node *n* is a child node of *p*. We call *parent selectivity* of the (child) node *n*, noted $(S_p)_n$, the fraction of the node *p*'s instances, that are selected after evaluating the path expression that stems from the parent *p* and the (child) node *n* is part of it.

The parent selectivity for a (child) node *n* and a parent node *p* is computed as follows:

$$(S_p)_n = N_{n,S} * (1/Fout_{p,n}) = F_n * NCARD_n * (1/Fout_{p,n})$$

Where $Fout_{p,n}$ is the fan-out of parent node *p* for the child node *n* and it represents the average number of children nodes *n* for an instance of a parent node *p*. Considering a uniform distribution of children nodes, the fan-out of parent node *p* for children nodes *n* is computed as follows: $Fout_{p,n} = NCARD_n / NCARD_p$. Replacing the fan-out in the parent selectivity formula we obtain:

$$(S_p)_n = F_n * NCARD_p$$

In the access-order selection optimization strategy the parent selectivity is important because we would like to evaluate first child nodes with higher parent selectivity.

Definition 3: we call *joint cost* of two path expressions that stem from the same root, the cost of evaluating first a path using a bottom-up evaluation plus the cost of evaluating the second path using a top-down evaluation.

The notation for the joint cost is $C_{first\_child\text{-}parent\text{-}second\_child}$. The meaning of the notation is that the path '/parent/first_child' will be evaluated first, using a bottom-up evaluation while the path '/parent/second_child' will be evaluated second using a top-down evaluation.

Based on the definitions above we can present the heuristics that we employ. These heuristics are based on a *uniform distribution* assumption for node instances. By uniform distribution we understand that child nodes with a common parent have approximately the same number of instances. That is, we make the assumption that the fan-out (number of children instances) for all instances of a certain parent node are approximately the same. We refer to these heuristics as *uniform distribution heuristics*. These heuristics employ the following two properties (that follow from the assumption).

Property 1: in the case of a uniform distribution, for a MTT rooted in node *a* with nodes *b* and *c* as children, if node *b* has a higher selectivity than node *c*, then: the parent selectivity of node *b* is higher than the parent selectivity of node *c* and $C_{bac} < C_{cab}$.

Property 2: in the case of a uniform distribution, for a MTT rooted in node *a* with nodes *b* and *c* as children, if node *b* has a higher parent selectivity than node *c*, then the cost of evaluating *c* top-down is less than the cost of evaluating *c* bottom-up.

Using the uniform distribution heuristic we can restrict the search space for the access order selection. Subsequently, the search algorithm works as follows: first, we sort the children according to parent selectivity; second, we evaluate the path with the lowest selectivity using a bottom-up evaluation; next, we evaluate all other paths, in the selectivity order, using a top-down evaluation.

The ToXop operator that employs both holistic path summary pruning and access-order selection is ToXinScan. The signature of the operator is as follows:

$$\text{ToXinScan}(\ (TT_D, TI_D), PT) = \text{Traverse}(\ \text{ComputePlan}(\ \text{PruneToXinTree}(\ (TT_D, TI_D), PT\ )$$

Where the PruneToXinTree operator performs the holistic path summary pruning, the ComputePlan operator computes the access order plan (i.e. the query plan) and the Traverse operator evaluates the MTTs according to the access order plan. An MTT augmented with the access order plan, i.e. a plan augment tree, is presented in Figure 7. Presenting in detail each of these operators is beyond the scope of this paper; more details on each of these operators and the algorithms that they employ can be found in [5].

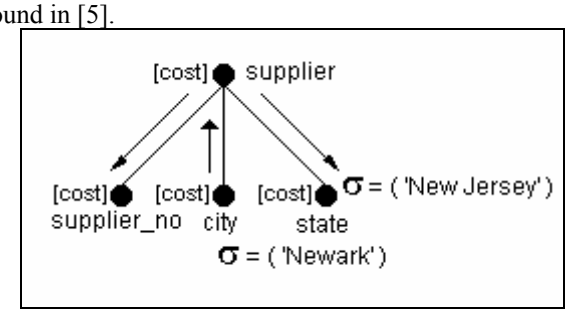

Figure 7: A plan augmented MTT tree

6 Experimental results

6.1 Experimental setup

We ran our experiments on 1.6 GHz Pentium M processor with 1GB of RAM running Windows XP Professional version 2002 Service Pack 1. The implementation of all algorithms (including TwigStack) was done in Java using Sun's j2re1.4.1_02. For all experiments the following JVM settings were used: "-Xmx896m -Xms896m".

6.2 Data sets

The data sets that we used are presented in Table 1. We obtained the DBLP and the SWISSPROT data from the University of Washington XML repository [25]. Both data sets are large, with millions of nodes, and they are shallow. However, these data sets differ in their structure; while DBLP is very regular in its structure, with five different kinds of structures that repeat many times, SWISSPROT is rather irregular, with many one-of-a-kind structures.

The third data set that we used is XMARK. We used *xmlgen* with factors ranging from 0.1 to 1.9 and then we removed the content of all Text tags except the first word from each. When the factor is not specified (e.g. Table 2) we refer to XMARK documents generated with a factor of 1.9.

6.3 Queries

The queries that we used are presented in Table 2. Queries 1 through 7 are inspired from the PRIX papers [28, 29] while the rest are queries on the XMARK data set. The queries from Table 2 can be classified into the following categories:

- *Punctual queries* (Q3, Q4, Q5, Q8) that is, queries that query only a small portion of the document and have a high selectivity, thus they return a small answer.
- *Low selectivity queries* (Q9, Q14) that is, they return a large answer.
- *Grouped twig queries* (Q11, Q12, Q13) that is, queries for which the nodes that might be part of the answer are grouped into a compact region in the document.
- *Scattered twig queries* (Q1, Q2, Q6, Q7, Q10) that is, queries for which the nodes that might be part of the answer are scattered through the entire document.

6.3 TwigStackScan vs. TwigStack

In order to establish the benefits of holistic path summary pruning we implemented a variant of TwigStack, that we call TwigStackScan, which uses holistic path summary pruning as well as the traditional TwigStack technique to compute twig queries. In order to incorporate the path summary information we extended the region algebra encoding by adding path summary information. We call this encoding as *extended region algebra encoding* and it has the following representation for elements and string values, respectively:

- [DocID, Term, Start, End, Level, SchemaID],
- [DocID, Term, TextValue, Start, Level, SchemaID]

Where *DocID* is the document ID; *Term* is the tag name; *Start/End* are the offsets in the document where the tag starts/ends measured as word count; *TextValue* is the CDATA or the attribute value; *Level* is the document level; *SchemaID* can be any numbering scheme; in TwigStackScan we use the TT node ID as SchemaID.

Dataset Name	Size (KB)	# of Elements	# of Attributes	# of Text	Total # of Nodes	Max-depth
DBLP	130,726	3,332,130	404,276	3,005,848	6,742,254	6
SWISSPROT	112,130	2,977,031	2,189,859	2,013,844	7,180,734	5
XMARK	112,486	2,769,710	726,783	1,478,252	4,974,745	10

Table 1: Datasets

	Query	Dataset	#of Twig Matches
Q1	//inproceedings[./author="Jim Gray"] [./year="1990"]/@key	DBLP	6
Q2	//www[./editor]/url	DBLP	5
Q3	//book/author[text() ="C.J. Date"]	DBLP	13
Q4	//inproceedings[./title/text() = "Semantic Analysis Patterns."]/author	DBLP	1
Q5	//Entry/Keyword[text() = "Rhizomelic chondrodysplasia punctata"]	SWISSPROT	3
Q6	//Entry/PFAM[@prim_id="PF00304"] [.//DISULFID/Descr]	SWISSPROT	6
Q7	//Entry[./Org="Piroplasmida"]//Author	SWISSPROT	13
Q8	//site/people/person[@id = "person0"]	XMARK	1
Q9	//site/people/person/name	XMARK	38,760
Q10	//regions/samerica/item[./location = "United States" AND ./@id AND ./name AND ./quantity AND ./payment]	XMARK	8
Q11	//person[@id = "person217" AND ./address [./city/text() = "Lubbock" AND ./country/text() = "United States]]/name	XMARK	1
Q12	//person[@id = "person20125" AND ./address [./city/text() = "Lubbock" AND ./country/text() = "United States]]/name	XMARK	1
Q13	//person[@id = "person48027" AND ./address [./city/text() = "Lubbock" AND ./country/text() = "United States]]/name	XMARK	1
Q14	//person[@id AND ./address [./city/text() = "Lubbock" AND ./country/text() = "United States]]/name	XMARK	80

Table 2: Twig queries in an XPath representation

Query Q8 is a "punctual" query, it retrieves *person* elements that are descendants of *site* elements with the *id* attribute equal to *'person0'*. Disregarding the document size and generation factor, for all XMark documents there is only one *person* element that satisfies this criterion. Since TwigStack filters early the element streams with regard to the selection predicate (*@id = "person0"* in this case) we would expect that the running time of TwigStack to be the same, independent of document size. From Figure 8a we conclude that this is not the case. While the running time of TwigStackScan remained relatively constant, independent of document size, the running time of TwigStack increased significantly with the size of the document. The explanation is that TwigStackScan retrieves and filters only the *id* attributes under the *person* tag, while TwigStack retrieves all *id* attributes, not only from the *person* tag but also from *category*, *item* and *open_auction* elements (for the path summaries, thus the exiting schemata, for the XMARK, DBLP and SWISSPROT documents used in this experiments please refer to [5]). The total number of *id* attribute instances evidently increases with document size, explaining the degradation in performance of the TwigStack algorithm.

We have to note as well a modest increase in TwigStackScan's execution time. This increase is due to the additional work that the algorithm has to perform because the size of the *people* and *person* streams increases with document size.

From Figure 8a we can deduct a first advantage of the holistic path summary pruning strategy, namely when the strategy is applicable, holistic path summary pruning assures a constant running time for a punctual query disregarding the document size.

In Table 3 we present the running times for TwigStack and TwigStackScan. For brevity, we omitted some of the queries from Table 2 because they yield similar results and while these queries are relevant for the performance of ToXinScan they are not for TwigStackScan. In Table 3 we also present the speedup of TwigStackScan versus TwigStack. The speedups can be categorized as: inexistent (e.g. query Q7); marginal that is, less than 2.0 (e.g. queries Q1, Q5, Q9 and Q14); significant that is, close to one order of magnitude (e.g. queries Q3, Q6, Q8, Q10 and Q11) and up to 75 (e.g. query Q2).

Based on the motivation for the holistic path summary pruning in the context of stack algorithms, we would expect to obtain significant speedups for the case when XPath nodes have multiple matches in a document and some nodes are part of the answer and some nodes are not. One such example is query Q1, where the elements *author* and *year* are not only part of *inproceedings* but also *article*, *book, mastersthesis* and *phdthesis*. However,

the speedup for query Q1 is not significant, only 1.49. The explanation lies in the fact that the main content of the DBLP data set is *inproceedings* thus adding the extra nodes does not produce a significant difference. Nevertheless, it is quite the opposite for query Q2 and Q3. In query Q3 the *book* content is relatively small, thus not adding the unnecessary nodes induces a significant speedup of 8.96. The speedup increases further for Q2 where the content referring to database software, encoded as *www* in the DBLP data set, is very small, reflected by the 75.38 speedup for TwigStackScan.

Query	TwigStack (ms)	TwigStackScan (ms)	TwigStack/ TwigStackScan
Q1	7,108	4,779	1.49
Q2	3,015	40	75.38
Q3	430	48	8.96
Q5	188	183	1.03
Q6	6,430	752	8.55
Q7	6,687	6,891	0.97
Q8	699	119	5.87
Q9	5,442	3,804	1.43
Q10	8,326	470	17.71
Q11	1,167	124	9.41
Q14	4,493	2,520	1.78

Table 3: The running times and the speedups for TwigStackScan vs. TwigStack

Pruning the path summary is not always beneficial. For instance, the speedup for query Q7 is 0.97, thus TwigStack performs better than TwigStackScan. The explanation relies in the fact that the *Org* and *Author* elements can be found only under the *Entry* element, thus the pruning does not help, moreover it ads an additional computational time that is reflected in the performance degradation. A similar situation is reflected by the speedup of 1.03 for query Q5, because the *Keyword* element is found only under the *Entry* element. The situation is the opposite for query Q6, where the *prim_id* attribute and the *Descr* element appear under many elements and not only under the *Entry* element, resulting in an 8.55 speedup.

A surprising result is exposed by queries Q9 and Q14. In both cases the elements from the query appear in many distinct paths, thus the pruning should be beneficial. However, the speedup is only marginal 1.43 and 1.78 respectively. Moreover, the queries are the low selectivity versions of queries Q8 and Q11, which perform to the expectation with speedups of 5.87 and 9.41. The explanation lies in the extensive computation performed by both queries due to the queries' low selectivity, for instance Q9 returns 38,760 twig matches. Thus, the advantage that the pruning is inducing by loading fewer nodes to be probed is overshadowed by the computational time per se.

6.4 ToXinScan vs. TwigStack

In Figure 8a we also plotted the performance of ToXinScan vs. TwigStack and as expected, on a punctual query the performance of ToXinScan remains constant while the performance of TwigStack degrades significantly.

In Figure 8b we plotted the speedup with document size of ToXinScan versus TwigStack for queries Q8, Q9 and Q10. As shown, there is a wide difference in speedup between these queries. At the bottom of the heap is Q8, the punctual query, with the lowest speedup. The explanation lies in the highly selective predicate on the *id* attribute. That is, TwigStack consumes time only by filtering an additional number of *id* attributes (three more *id* streams, to be exact) but in the end retains only one instance of the *id* attribute, the one that satisfies the predicate. Overall ToXinScan's improvement ranges between 3 and 9 times.

The second query, Q9, is similar to Q8 but missing the predicate on the *id* attribute, thus it is a low selectivity query. Because there is no longer a predicate, TwigStack does not perform any filtering and loads and probes all streams for any occurrences of the *id* attribute. This behavior resulted in a significant performance advantage for ToXinScan reflected in the graph, in the range of 20 to 60 times.

The best improvement is exhibited by query Q10, a heavy twig query, from 33 to 122 times. The reason is that each of the element tags in the query appears in many sections of the document in different paths, thus TwigStack does an extensive computation that does not yield any useful matches.

Queries Q11, Q12 and Q13 have a similar twig structures with different values for the selection predicates on the *id* attribute. The first value "217" is at the beginning of the area that has to be queried, namely the sub-tree under the *person* element; the "21,125" value is in the middle of this area while the value "48,027" is at the end. Because the queries have the same twig structure one might expect that the performances of these queries to be similar. Nevertheless, it is not the case, and as it can be inferred from Figure 8c; while the performance of ToXinScan remains constant disregarding the value of the selection predicate, the performance of TwigStack degrades significantly. The explanation lies in the fact that ToXinScan performs an optimization, thus rightly identifies the position of the answer nodes.

In Table 4 we present the speedups of ToXinScan versus TwigStack for all queries from the query set. As can be inferred from the table, ToXinScan outperforms TwigStack marginally for punctual queries, with speedups from 2 to 9. The explanation relies in the reduced computation that has to be perform in order to process the answer, thus ToXinScan's optimization strategy does not produce a significant difference.

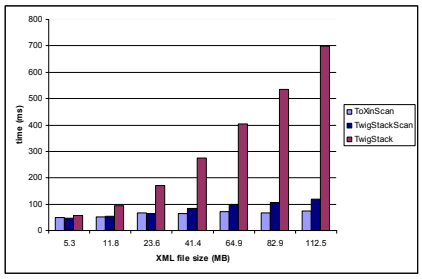

Figure 8a: Q8 execution time (ms) for TwigStack, TwigStackScan, ToXinScan with document size

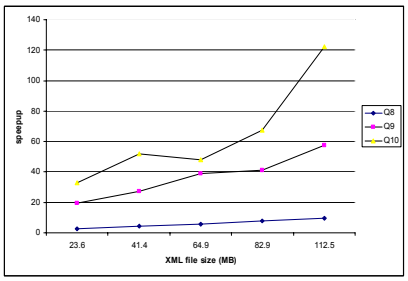

Figure 8b: Speedup TwigStack/ToXinScan for Q8, Q9, Q10 with document size

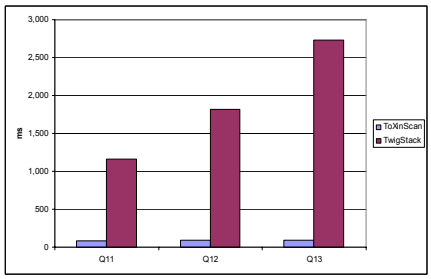

Figure 8c: Execution times (ms) for queries Q11, Q12 and Q13 for ToXinScan and TwigStack

The computation required for grouped twig queries is slightly higher than punctual queries, thus the optimization starts to pay-off inducing speedups ranging from 12 to 28. For low selectivity queries and scattered twig queries the optimization strategy proves extremely beneficial inducing speedups ranging from 51 to 122.

Query	TwigStack (ms)	ToXinScan (ms)	TwigStack/ ToXinScan
Q1	7,108	130	54.68
Q2	3,015	39	77.31
Q3	386	90	4.29
Q4	430	46	9.35
Q5	188	87	2.16
Q6	6,430	80	80.37
Q7	6,687	131	51.05
Q8	699	75	9.32
Q9	5,442	95	57.28
Q10	8,326	68	122.44
Q11	1,167	90	12.97
Q12	1,816	92	19.74
Q13	2,746	95	28.80
Q14	4,493	93	48.31

Table 4: ToXinScan vs. TwigStack

6.5 ToXinScan vs. PRIX

It is no surprise that the speedup of PRIX over TwigStack, due to the use of a full index, is considerable. What comes as a surprise is that employing a path summary, with no index data structures, and simple heuristics we achieve a speedup of the same order of magnitude, while performing less work, since building the PRIX data structures is quite computationally intensive. In Table 5 we present the speedups of ToXinScan and PRIX versus TwigStack. The speedups of PRIX versus TwigStack are taken from the PRIX papers [28, 29]. The comparison between ToXinScan and TwigStack was performed with our implementation of both methods

As can be inferred from Table 5, both ToXinScan and PRIX achieve a speedup in the range of one to two orders of magnitude over the respective TwigStack implementation. The relative size of the PRIX vs. ToXinScan speedups depends on the query.

Query	TwigStack/ ToXinScan	TwigStack/PRIX (from [28, 29])
Q1	54.68	14.01
Q2	77.31	145.00
Q5	80.37	43.15

Table 5: Speedup of TwigStack / ToXinScan vs. TwigStack / PRIX

6. Conclusions

We present experimental results that characterize the performance of an XML query optimizer that takes advantage of path summaries in conjunction with two optimization strategies: holistic path summary pruning and access-order selection.

The use of path summaries augmented with data statistics in the ToXop optimizer provides similar advantages to the use of system catalog information in relational query optimizers.

Path summaries enable our first optimization strategy, holistic path summary pruning, which reduces considerable the query plan search space by identifying the regions from the document that contain the query answer. We provide experimental data that suggest that, when applicable, this strategy can easily yield improvements of an order of magnitude.

Access-order selection, the second optimization strategy that we propose, works in conjunction with holistic path pruning, and further reduces the plan search space by using cost-based heuristics. Employing access-order selection can easily yield improvements of two orders of magnitude with respect to algorithms that work on encoded-element streams. Moreover, the improvements are in the same order of magnitude as those achieved by state-of-the-art indexing techniques for XML documents. The advantage of our proposed approach lies in the reduced cost of creating a path summary (which can be done with essentially one pass over the document), compared with indexing techniques that require a

substantially larger amount of computation to generate the indexes. In summary: with little (effort to preprocess XML data) you can achieve a lot (of performance improvement in query processing)!

Future work includes incorporating and experimenting with a wider variety of native XML indexing access methods within the uniform framework described here. We are also interested in extending the use of XML-specific statistics to support better cost models and cost estimates.

Bibliography

[1] A. Aboulnaga, A. Alameldeen, J. F. Naughton, "Estimating the Selectivity of XML Path Expressions for Internet Scale Applications", *Proc. VLDB, 2001.*

[2] S. Al-Khalifa, H.V. Jagadish, "Combining Operators in XML Query Processing", *Proc. VLDB, 2002.*

[3] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, Divesh Srivastava, Y. Wu, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching", *Proc. ICDE, 2002.*

[34] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, A. Pugliese, "Efficient Query Evaluation over Compressed XML Data", *Proc. EDBT 2004*.

[4] D. Barbosa, A. Barta, A.O. Mendelzon, G.A. Mihaila, F. Rizzolo, P. Rodriguez-Gianolli, "ToX - The Toronto XML Engine", *Proc. Int. Workshop on Inf. Integration on the Web, 2001*.

[5] A. Barta**,** *"*Access Methods for XML Query Optimization*", Ph.D. Thesis, U. of Toronto, 2005.*

[6] A. Barta**,** M.P. Consens, A.O. Mendelzon, *"*XML Query Optimization Using Path Indexes*", Proc. XIME-P 2004*.

[7] N. Bruno, D. Srivastava, N. Koudas, "Holistic Twig Joins: Optimal XML Pattern Matching", *Proc. SIGMOD, 2002.*

[35] P. Buneman, B. Choi, W. Fan, R. Hutchison, R. Mann, S. Viglas, "Vectorizing and Querying Large XML Repositories*", Proc. ICDE 2005.*

[8] V. Christophides, S. Cluet, G. Moerkotte, "Evaluating Queries with Generalized Path Expressions", *Proc. SIGMOD, 1996.*

[9] M. P. Consens, T. Milo, "Algebras for Querying Text Regions", *Proc. PODS, 1995.*

[10] A. Deutsch, L. Popa, V. Tannen, "Physical Data Independence, Constraints, and Optimization with Universal Plans", *Proc. VLDB, 1999.*

[11] M.F. Fernandez, D. Suciu, "Optimizing Regular Path Expressions Using Graph Schemas", *Proc. ICDE, 1998.*

[12] T. Fiebig, S. Helmer, C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, T. Westmann, "Natix: A Technology Overview", *Proc. Web, Web-Services, and Database Systems, 2002.*

[13] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, G. Agrawal, "The BEA/XQRL Streaming XQuery Processor", *Proc. VLDB, 2003.*

[14] R. Goldman, J. Widom, "DataGuides, "Enabling Query Formulation and Optimization in Semistructured Databases", *Proc. VLDB, 1997.*

[15] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A.N. Rao, F. Tian, S. Viglas, Y. Wang, J.F. Naughton, D. J. DeWitt, "Mixed Mode XML Query Processing", *Proc. VLDB, 2003.*

[16] H.V. Jagadish, S. Al-Khalifa, A. Chapman, L.V.S. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, C. Yu, "TIMBER: A Native XML Database", *VLDBJ, 2003.*

[17] H.V. Jagadish, Laks V.S. Lakshmanan, Divesh Srivastava, K. Thompson, "TAX: A Tree Algebra for XML", *Proc. DBPL, 2001.*

[18] H. Jiang, W. Wang, H. Lu, J.X. Yu, "Holistic Twig Joins on Indexed XML Documents", *Proc. VLDB, 2003.*

[19] R. Kaushik, P. Shenoy, P. Bohannon, E. Gudes, "Exploiting Local Similarity for Indexing Paths in Graph-Structured Data", *Proc. ICDE, 2002.*

[20] Lucent's Galax, http://db.bell-labs.com/galax/

[21] A. Marian, J. Siméon, "Projecting XML Documents", *Proc. VLDB, 2003.*

[22] J. McHugh, J. Widom, "Compile-Time Path Expansion in Lore", Proc. Workshop on Q. Proc. for Semistr. Data and Non-Standard Data Formats, 1999.

[23] J. McHugh, J. Widom, "Query Optimization for XML" *Proc. VLDB, 1999.*

[24] T. Milo, D. Suciu, "Index Structures for Path Expressions", *Proc. ICDT, 1999.*

[25] G. Miklau, "U.W. XML Repository", http://www.cs.washington.edu/research/xmldatasets.

[26] N. Polyzotis, M.N. Garofalakis, Y.E. Ioannidis, "Selectivity Estimation for XML Twigs", *Proc. ICDE, 2004.*

[27] C. Qun, A. Lim, K. W. Ong, "D(k)-Index: An Adaptive Structural Summary for Graph-Structured Data", *Proc. VLDB, 2003.*

[28] P.R. Rao, B. Moon, "PRIX: Indexing and Querying XML Using Prufer Sequences", *Proc. ICDE, 2004.*

[29] P.R. Rao, B. Moon, "PRIX: Indexing and Querying XML Using Prufer Sequences", *Technical Report TR-03-06, U. of Arizona, Tucson, 2003.*

[30] F. Rizzolo, A.O. Mendelzon, "Indexing XML Data with ToXin", *Proc. WebDB, 2001.*

[31] H. Wang, S. Park, W. Fan, P.S. Yu, "VIST: A Dynamic Index Method for Querying XML Data by Tree Structures", *Proc. SIGMOD, 2003.*

[32] Y. Wu, J. Patel, H.V. Jagadish, "Using Histograms to Estimate Answer Size for XML Queries", *Journal of Information Science 2002.*

[33] C. Zhang, J.F. Naughton, D.J. DeWitt, Q. Luo, G.M. Lohman, "On Supporting Containment Queries in Relational Database Management Systems", *Proc. VLDB, 2001.*