# XML Query Optimization Using Path Indexes

Attila Barta
Dep. 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

With the growing interest in native XML query processing comes an increased awareness of the lack of maturity in XML optimizers. We believe that there is a significant opportunity to adapt and extend mature relational optimization techniques in XML systems.

In this paper we introduce a novel two-level approach to cost-based optimization. The higher level consists of the traditional join order selection together with the cost-based selection of access methods. The lower level cost-based optimization is entirely performed within an original access method that takes advantage of XML path indexes. A *path index*, also known as a *structural index* or as a *structural summary*, represents a summarization of the paths that actually occur in an XML document. Using path indexes in XML optimization helps to constrain the query plan search space and allows the exploitation of cost models based on XML-specific statistics.

The optimization approach is described in the context of ToXop, a cost-based optimizer for ToX that seamlessly incorporates both streaming (single-pass) and path index based pattern matching evaluation strategies for XQuery.

## 1. INTRODUCTION

Over the past few years the Extendible Markup Language (XML) has became 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 [28] is the predominant proposal for a native XML query language standard. Query optimization proved to be the foundation of success for Relational Database Management Systems. However, while there are many XQuery implementations, the vast majority of them lack a query optimizer.

We believe that there is a significant opportunity to adapt and extend mature relational optimization techniques in XML systems. As in relational systems, we can regard XML query optimization as a combination of two stages: access method selection and join order selection (these separate stages are a simplification that applies for Select-Project-Join SQL queries with no sub-queries). In relation systems, the data access operators also incorporate selection and projection operations.



Informal Proceedings of the *First International Workshop on XQuery Implementation, Experience, and Perspectives (XIME-P)*, June 11–18, 2004, Paris, France.

This approach can be applied to the XML model as well, with one major difference: in the XML model the selection predicates are XPath expressions. The evaluation of these path expressions by different access methods used within XML engines represents the bulk of the research in XML query evaluation.

Methods for evaluating XPath expressions have been developed with the assumption that the query processor has to apply the selection condition while streaming through the document. There is extensive work on streaming XPath processors [5, 9, 12, 15, 21, 22]. These implementations use automata in order to evaluate on the fly a number of XPath expressions. Any of these processors can be encapsulated into an access method.

When a native XML system controls the storage of the XML documents, the stream processing requirement can be relaxed. Consequently, the XML documents can be pre-parsed into special purpose data structures in order to speed up query execution. One such data structure is derived from the notation used in region algebras [7], where an inverted file like structure with the element name, the begin, start and level of the text region of each element is used. Utilizing this type of data structures evaluating path expressions is equivalent to joins between lists of element regions, referred to as *containment queries* [27] or *structural joins* [2].

In the relational model, a SQL query is translated into a relational algebra expression and then optimized. Many of the XQuery implementations use the same approach, although many of the algebras in the literature are simply an API or are based on some procedural semantics. For instance, Galax [30] translates XQuery to XQuery Core [29]. Other XQuery systems, such as Timber [16], Niagara [14], Natix [10] and ToX [3] rely on a logical XML algebra. In between are systems like BEA/SQRL [11] that translates XQuery into an internal representation and performs optimization based on heuristics. Ultimately, XQuery systems implement a set of access methods.

```
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 = "Toronto"
    and  $x/province = "Ontario"
  return
    <result> { $y/name } { $y/description }  </result>
```

Figure 1: Sample XQuery expression

An XQuery expression frequently contains several XPath sub-expressions, such as the example given in Figure 1.

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 Toronto, Ontario. The XPath expressions that occur in the query above and apply to the supplier document are: *'//supplier'*, *'//supplier/supplier_no'*, *'//supplier/city'*, and *'//supplier/province'*. A fragment of a sample supplier document is shown in Figure 2.

```
<suppliers>
    <supplier>
        <supplier_no> 1 </supplier_no>
        <name> Magna </name>
        <city> Toronto </city>
        <province> Ontario </province>
    </supplier>
    <supplier>
        <supplier_no> 2 </supplier_no>
        <name> Ford Canada </name>
        <city> Oakville </city>
        <province> Ontario </province>
    </supplier>
    <supplier>
        <supplier_no> 3 </supplier_no>
        <name> MEC </name>
        <city> Vancouver </city>
        <province>British Columbia</province >
    </supplier>
</suppliers>
```

Figure 2: Sample *supplier.xml* document

Although each of the XPath expressions in an XQuery expression such as the example above can be computed separately, a much better approach is to group XPath queries. The group of XPath expressions that apply to a certain document (such as suppliers in the example above) can be computed in a single pass through the document. Most of the XPath processors as well as novel structural join algorithms support this approach [4, 15, 16].

Relational query optimization relies on schema information. In the XML data model the information given by XML Schemas (or by DTDs) specifies the conditions for validity of documents. However, an XML Schema is not necessarily very descriptive about the structures occurring in a collection of XML documents. As such, XML Schema information is not well-suited for optimization. However, there are different data structures employable as schema, namely path indexes. A *path index*, also known as a *structural index* or as a *structural summary*, 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 index [13, 20] or an approximation of it [18, 24]. Because, a path index reflects the existing schema of the document, it can be used for optimization in a similar fashion than a relational schema.

In this paper, we describe a comprehensive approach to XML query optimization that incorporates several novel characteristics. The description in the paper focuses on ToXop, a cost-based optimizer for ToX.

The key novel contributions showcased by ToXop are:

- An original two-level approach to cost-based optimization. The higher level consists of the traditional join order selection together with the cost-based selection of access methods. The lower level cost-based optimization is entirely performed within the access method that takes advantage of the path indexes.
- The high level cost-based selection is based on just two access methods specialized for XML query processing. The first access method supports streaming (single-pass) pattern matching in XML documents, while the second access method takes advantage of path index structures for matching patterns in pre-processed XML documents.
- The lower level cost-based optimization determines how the path index will be traversed. This choice is encapsulated in the second access method described above. This access method also encapsulates the pruning of the patterns to be matched against the path index. Incorporating the path index in this lower level optimization has the benefits of constraining the query plan search space and exploiting a cost model based on XML-specific statistics
- The two access methods used by ToXop share the same data model as the operators in the native XML logical algebra. They operate on collections of trees at both levels, which should be contrasted with XML implementations in the literature that employ structural join access methods, which operate on collections of trees at the logical level, but switch to operations on collections of nodes at the physical level.

We have to note here a similarity of the work that we present in the XML context, with earlier the work in query optimization for Object Oriented Databases (OODB) [6, 8, 19]. For instance, Access Support Relations (ASRs) provide an indexing mechanism for paths like XML path indexes. 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 path indexes cover all paths from the document instance. Finally, previous work on optimization that incorporates ASRs does not employ the two-level cost-based access method selection introduced in this paper.

In the following section we introduce the cost-based access method selection employed by ToXop (the high level optimization). Section 3 describes the use of path indexes within an access method that performs the low-level cost-based optimization. We conclude by mentioning future research in Section 4.

## 2. COST-BASED ACCESS METHOD SELECTION

In this section we describe the cost-based access method selection used by ToXop, an XML query optimizer that exploits ToXin, a path index. ToXop and ToXin are part of a larger project: the Toronto XML Server (ToX), a native DBMS for XML under development at the University of Toronto [3].

The logical algebra used in ToXop is essentially the Tree Algebra for XML (TAX) [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 two level optimization approach that exploits path indexes 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 3 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/province*.

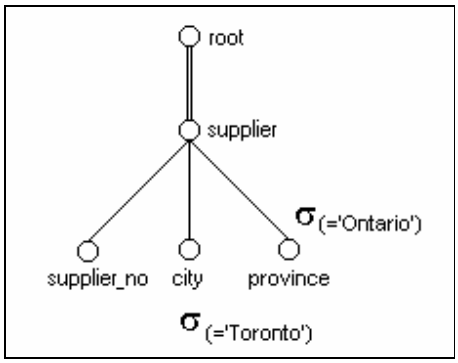


Figure 3: The Pattern Tree *PT1*

In a relational optimizer, each logical operator can be implemented by many access methods. The purpose of access method selection is to choose a low cost alternative among those that are applicable. Relational optimizers also group several logical operators within a single access method. For instance, the relational access method Scan does group functionality from the relational algebra (logical) operators Selection and Projection. Finally, in relational implementations, the logical operators and the corresponding access methods operate on the same data model, namely relations.

In the ToXop framework, both logical operators and access methods operate on the same XML model, namely collections of trees. This approach is novel, since in other XML optimization frameworks the access methods do not operate on the same XML model as the logical operators. For instance, access methods based on structural join operate on sets of nodes.

In order to perform a cost based optimization we need catalog information. In the relational model the catalog information contains schema information and statistics on data distribution. However, in the XML model the existing schema specifications, DTD and XML-Schema, are merely used for document validation and do not reflect the existing schema of the document. In order to overcome this limitation we employ a path index (specifically, ToXin) as document schema. In ToXin, for each distinct path in the XML document there is a distinct path in the index [25]. ToXin has a tree structure, reflecting a simplified XML data model.

Figure 4 has an example ToXin tree for the supplier document shown in Figure 2.

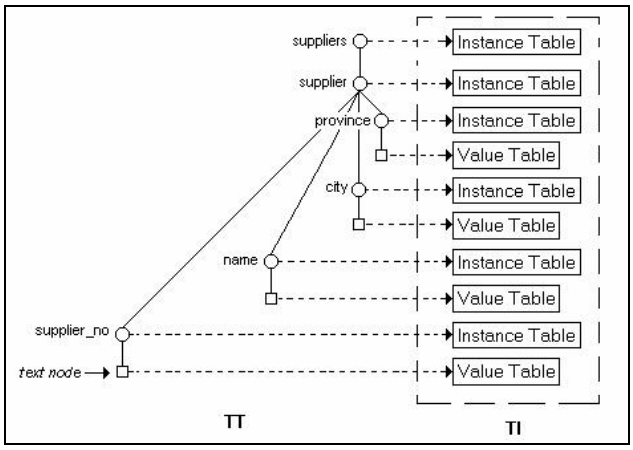


Figure 4: A ToXin tree for the *supplier.xml* document.

We divide the ToXin tree into two structures: the ToXin tree itself (***TT***) and a set of associated *instance* and *value* tables (collectively, ***TI***). Each node in *TT* points either to an Instance Table, which contains the parent/child information to support forward and backward navigation, or it points to a Value Table, which records the content of each node.

The ToXin index has two important properties. First, it summarizes the structure of a document instance. Second, a path query can be answered by traversing the ToXin tree only. These two properties suggest that we can employ ToXin as both a catalog and an index.

In order to use ToXin as a comprehensive catalog we need to collect statistics on the data. Whereas in the relational model the statistics to gather have been extensively studied, in the XML model the statistics are still a subject of research [1, 23, 26]. In the initial version of ToXop we use a simple set of statistics such as the number of instances for an element, the number of distinct values for an element and its fan-out. We collect these statistics

in an augmented version of the TT structure. The statistical information is gathered at document parsing time. The root node of the TT structure contains one piece of additional information: the size of the document.

ToXop supports two basic access methods: ToXStream and ToXinScan. The first access method supports streaming (single-pass) pattern matching in XML documents, while the second access method takes advantage of path index structures for matching patterns in pre-processed XML documents. These two access methods also support grouping the functionality of several logical operators such as Selection and Projection. ToXop makes a cost-based decision to determine which one of the two access methods to use in a given plan.

ToXStream is a variant of TurboXPath, an XPath processor that works on streaming data [15]. ToXStream uses a bounded memory automaton-like data structure and evaluates the pattern tree in a single pass through the document. ToXStream is well suited to work on documents that are streamed or documents that have not yet been pre-parsed to create index structures.

The ToXStream access method has a similar signature as a logical algebra operation. It takes as input a document D and a pattern tree PT and outputs a collection of n witness trees in D that satisfy the pattern tree PT:

$$\text{ToXStream}(D, PT) = [\ WT^1, WT^2, \ldots, WT^n\ ]$$

ToXStream can also produce, during its single pass over a document D, the ToXin index structures for D, $TT_D$ and $TI_D$. Therefore, once ToXStream processes a document, the corresponding $TT_D$ and $TI_D$ are available for subsequent operations.

ToXop can determine the cost of utilizing the ToXStream access method by looking at the size of the document to stream. Recall, that the root node of the $TT_D$ structure contains the size of the document. As we will see below, once the $TT_D$ structure is available, we will use it to retain the execution cost of the ToXinScan access method. In particular, the total cost of ToXinScan will be stored in the root node of $TT_D$. Therefore, ToXop uses a comparison of the costs at the root of the path index structure to select an access method.

## 3. PATH INDEXES IN THE OPTIMIZER

This section describes the access method in ToXop that uses path indexes and illustrates the low-level cost-based optimization.

ToXinScan is the access method that operates on the ToXin index structures. Thus, a ToXinScan operator 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:

$$\text{ToXinScan}(D,PT) = [\ WT^1, WT^2, \ldots, WT^n\ ]$$

ToXinScan is the result of composing three operators: PruneToXinTree, ComputePlan and Traverse.

The first step (PruneToXinTree) is to match the pattern in the query against the path index. For a given pattern tree PT and an augmented ToXin tree TT, we call *matched ToXin Trees* (*MTT*) those sub-trees of TT that satisfy the pattern tree *PT*. The nodes of *MTT* are adorned with the corresponding node selection predicates from the pattern tree *PT*.

Therefore, PruneToXinTree takes a pattern tree PT and an augmented ToXin tree TT and outputs the k sub-trees of TT that satisfy the pattern PT:

$$\text{PruneToXinTree}\ (TT_D, PT) = [TT^1_{D,PT}, TT^2_{D,PT}, \ldots, TT^k_{D,PT}]$$

In Figure 5 we present the result of pruning the ToXin tree from Figure 4 according to the pattern from Figure 3.

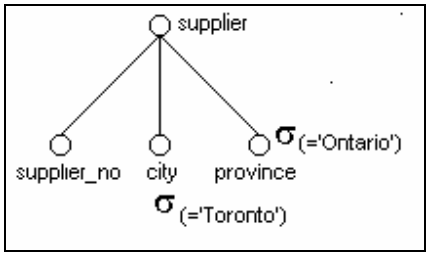


Figure 5: A matched ToXin tree

The second step (ComputePlan) is to evaluate the execution cost of MTTs. First, we observe that the evaluation cost function works recursively on TTs (each node contains the cost of evaluating the sub-tree rooted in that node). In order to obtain the costs, the lower level cost-based optimization will have to determine how the path index will be traversed.

For the MTT from Figure 5, in order to retrieve all nodes that satisfy the selection predicates we can proceed as follows. First we evaluate the path *'/supplier/province'* 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 *province* 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/province'*; this means that we have a set of *supplier* nodes for which there are child *province* nodes that satisfy the predicate on *province*. 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/province'* 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/province'* 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 the lower level optimization in the two level optimization process. This optimization is an XML-specific technique that relies on the use of path indexes.

In Figure 6 we present a *plan augmented ToXin tree* that is, a ToXin tree with the right order/selection in place for each node and the lowest execution cost computed and stored for each node. In the example we assume that the predicate on the *city* node has higher selectivity than the predicate on the *province* node thus, we will evaluate the *'/supplier/city'* path first using a bottom-up evaluation and then the *'/supplier/province'* path using a top-down evaluation. The arrows in the figure reflect these choices of direction.

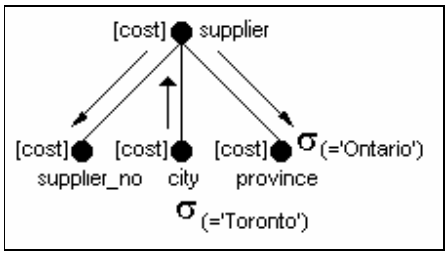


Figure 6: A plan augmented ToXin tree

Plan selection is done by the ComputePlan operator. It takes as input a matched toxin Tree $TT^{i}_{D,PT}$ and outputs a plan augmented ToXin tree $PTT^{i}_{D,PT}$:

$$ComputePlan(TT^{i}_{D,PT}) = PTT^{i}_{D,PT}$$

The last operator is Traverse. It takes a plan augmented ToXin tree $PTT^{i}_{D,PT}$, the corresponding instance and value table $TI_D$ and outputs a witness tree that is part of the answer to the query. Traverse has the following signature:

$$Traverse(PTT^{i}_{D,PT}, TI_D) = WTT^{i}$$

In summary, the ToXinScan access method is the composition:

```
ToXinScan(D,PT)=
  Traverse(
     ComputePlan( PruneToXinTree( getTT(D),PT) ),
     getTI(D) )
```

where getTT and getTI are two auxiliary functions that return $TT_D$ and $TI_D$, respectively.

ToXinScan can take advantage of pruning the patterns early on , selecting those parts of the document which are of interest, and drastically reducing the size of the trees manipulated by the execution engine. ToXinScan cam yield benefits by constraining the query plan search space and exploiting a cost model based on XML-specific statistics.

The current ToXop prototype includes an implementation of the ToXinScan operator. In Figure 7 we present and intermediate output (pretty printed as XML) of the ToXop optimizer describing the *supplier* node of a plan augmented tree that is part of the optimization process.

```
<ToxinNode>
  <id>2</id>
  <name>supplier</name>
  <level>1</level>
  <type>3</type>
  <parentID>1</parentID>
  <cost> 5.0 </cost>
  <children>
    <child>
      <id>3</id>
      <order>3</order> <direction>down</direction>
    </child>
    <child>
      <id>4</id>
      <order>1</order> <direction>up    </direction>
    </child>
    <child>
      <id>5</id>
       <order>2</order> <direction>down</direction>
    </child>
  </children>
  <statistics>
    <!-- no text content, thus no values -->
    <distinctValue> null </distinctValue>
    <fanOut>
      <value>1.0</value>
      <value>1.0</value>
      <value>1.0</value>
    </fanOut>
  </statistics>
</ToxinNode>
```

Figure 7: The supplier node augmented with plan information

## 4. CONCLUSIONS

We have presented on-going work on a two level approach to XML optimization used by ToXop, a cost-based XML query optimizer. The higher level consists of the cost-based selection of two XML access methods: ToXStream and ToXinScan. ToXStream supports single pass evaluation through XML documents, while ToXinScan operates on ToXin trees, a path index structure for XML documents. Both access methods use ToXin trees as a system catalog. The higher optimization level also includes traditional join order selection.

The lower level optimization determines how the path index will be traversed. This choice is encapsulated in the second access method, ToXinScan. This access method also encapsulates the pruning of the patterns to be matched against the path index. Exploiting the path index in this lower level optimization brings the two crucial benefits of constraining the search space of query plans and of exploiting a cost model based on XML-specific statistics.

Our immediate plans for ToXop begin with an experimental evaluation of the optimizer's behavior. Future work includes incorporating a wider variety of native XML indexing access methods (beyond path indexes) into a uniform framework such as the one described here. We are also interested in extending the use of XML-specific statistics to support better cost models and cost estimates.

## REFERENCES


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

[2] 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, San Jose, CA, 2002.*

[3] D. Barbosa, A. Barta, A. Mendelzon, G. Mihaila, F. Rizzolo, P. Rodriguez-Gianolli, "ToX - The Toronto XML Engine", *International Workshop on Information Integration on the Web, Rio de Janeiro, 2001.*

[4] N. Bruno, L. Gravano, N. Koudas, D. Srivastava, "Navigation vs. Index-Based XML Multi-Query Processing", *Proc. ICDE, Bangalore, India, 2003.*

[5] J. Chen, D. J. DeWitt, F. Tian, Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases", *Proc. ACM SIGMOD, Dallas, TX, 2000.*

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

[7] M. P. Consens, T. Milo, "Algebras for Querying Text Regions", *Proc. PODS , San Jose, CA, 1995.*

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

[9] Y. Diao, P. M. Fischer, M. J. Franklin, R. To, "YFilter: Efficient and Scalable Filtering of XML Documents", *Proc. ICDE, San Jose, CA, 2002.*

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

[11] 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, Berlin, Germany, 2003.*

[12] A. K. Gupta, Dan Suciu, "Stream Processing of XPath Queries with Predicates", *Proc. ACM SIGMOD, San-Diego, CA, 2003.*

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

[14] 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, Berlin, Germany, 2003.*

[15] V. Josifovski, M. Fontoura, A. Barta, "Querying XML Streams", *VLDB Journal, 2004.*

[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", *VLDB Journal 2003.*

[17] H.V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson, "TAX: A Tree Algebra for XML", *Proc. of DBPL'01, Rome, Italy, 2001.*

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

[19] A. Kemper, G. Moerkotte, "Advanced Query Processing in Object Bases Using Access Support Relations", *Proc. VLDB, Brisbane, Australia, 1990.*

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

[21] D. Olteanu, T. Kiesling, F. Bry, "An Evaluation of Regular Path Expressions with Qualifiers against XML Streams", *Proc. ICDE, Bangalore, India, 2003.*

[22] F. Peng, S. S. Chawathe, "XPath Queries on Streaming Data", *Proc. VLDB, Berlin, Germany, 2003.*

[23] N. Polyzotis, M. N. Garofalakis, "Statistical synopses for graph-structured XML databases", *Proc. ACM SIGMOD, Madison, WI, 2002.*

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

[25] F. Rizzolo, A. Mendelzon, "Indexing XML Data with ToXin", *Fourth International Workshop on the Web and Databases, Santa Barbara, CA. 2001.*

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

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

[28] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, "XQuery 1.0: An XML Query Language", *World Wide Web Consortium*, http://www.w3.org/TR/xquery/

[29] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, Philip Wadler, "XQuery 1.0 and XPath 2.0 Formal Semantics", *", World Wide Web Consortium*, http://www.w3.org/TR/query-semantics/

[30] Lucent's Galax, *Bell Labs*, http://db.bell-labs.com/galax/