XQuery Optimization in Relational Database Systems

Riham Abdel Kader
Supervised by Maurice van Keulen
Univeristy of Twente
P.O. Box 217
7500 AE Enschede, The Netherlands
r.abdelkader@utwente.nl

ABSTRACT

With the increasing need for manipulating and exchanging XML data, the topic of processing XML documents and optimizing XML queries is being studied by many researchers. There are still, however, many open issues in query optimization in the context of XML database systems. In this paper, we give an overview on the state of XML query optimization by reviewing the different optimization techniques developed by previous research and employed by existing XML database systems. We subsequently enumerate some of the still unresolved problems in the optimization process. We conclude this paper with a description of our current work in which we focus on improving the evaluation of XPath expressions by rewriting the path at the algebraic level.

1. INTRODUCTION

The eXtensible Markup Language (XML) defined and standardized by the World Wide Web Consortium (W3C) has proven to be a good model for data format and exchange in distributed systems and over the Web due to its tree structure and flexibility. The large spread of XML implies a growth in the size of the data exchanged, stored and subsequently processed. Thus there is a pressing need for XML optimization techniques and much current research is focused on developing such techniques.

Several languages have been defined for selecting and transforming XML data, of which two languages XPath and XQuery are standardized by W3C . While XPath can only perform selections on an XML document, XQuery supports richer operations (joins, aggregations, and element construction).

Several systems for processing XQueries over stored documents have been proposed. These systems differ in several aspects, two of which are the storage layout (*i.e.*, relational or native), and the adopted algebra (*i.e.*, tree-based or tuple-based). In a relational approach, XML data is mapped to and stored as relational tables while native systems have a dedicated tree-based storage structure. Specific optimization techniques have been developed for many systems. Extended relational systems have an advantage over native engines in that they can make use of the mature relational technology. These optimization techniques, however, fall short in meeting the complexity of the XML data and the development of new techniques or the extension of the already existing ones is required.

This PhD research focuses on XQuery optimization in the context of relational database systems. Our aim is to develop algebraic equivalence rules for query rewriting, accurate cost models for estimating plan costs based on collected statistics and a plan enumeration and selection technique that chooses the best plan at runtime. Besides the more generic scientific contributions applicable to other systems, the result of this research is meant to be incorporated in the relational database system MonetDB/XQuery.

In this paper, we first describe the state of art for some optimization techniques. Section 3 lists some of the unresolved problems in XQuery optimization and in Section 4 we describe the motivation and the work we are currently doing to develop an equivalence rule for algebraically rewriting XPath expressions.

VLDB '07, September 23-28, 2007, Vienna, Austria.

2. EXISTING XML OPTIMIZATION TECHNIQUES

2.1 Optimization Based on Equivalence Rewriting

In XML, equivalence rewriting can be accomplished at two different levels: XQuery core level which is equivalent to syntactical rewriting, and algebraic level. The XQuery core language defined in [5] is a proper subset of the XQuery language and consists of a set of simple expressions to which the more complex XQuery expressions are rewritten before being translated into a logical plan.

2.1.1 XPath Rewriting

Several XML processing systems can not handle XPath expressions with reverse axes, such as parent, ancestor, preceding, etc. Moreover, some types of systems (*e.g.*, streaming systems) are not efficient in evaluating this group of axes. An example of an XQuery core level equivalence rewriting is the technique described in [13]. It tries to overcome the first problem and to optimize the processing in the second situation by defining two sets of equivalence rules to rewrite an XPath expression involving reverse axes to an equivalent reverse-axis-free one. One set of rules rewrites the XPath to an expression including a join operator while the other set rewrites the XPath to one containing more location steps.

2.1.2 Order and Duplicate Elimination

One important requirement with optimization potential is that nodes returned from XPath or XQuery evaluation should be in document-order unless the user explicitly disregards order. The XQuery standard specifies that an XPath is translated to an XQuery core expression such that every step in the XPath is followed by a distinct-docorder operator. This operator maintains not only the document order but also the uniqueness of the nodes throughout the path evaluation. Although having these operators after each step keeps the intermediate results duplicate-free, it might slow down the system's performance. An alternative is to sort and eliminate duplicates only at the end of the path evaluation but this will result in performing unnecessary and redundant work which may also degrade the system's performance. The technique proposed in [6] decides after which steps in the plan to keep these expensive operators and after which they can be safely removed, such that the evaluation time of the plan is minimized. Properties like ordered, and duplicate free are assigned to a path expression if its evaluation returns a set of nodes that conforms to these properties. The described technique is an automaton-based algorithm called DDO which, given a starting set of nodes having a certain property, can infer the property of the set resulting from applying a certain XPath axis. By inferring the properties of the result, it can be decided if the presence of a sort and/or duplicate elimination operator is necessary after this step. Although this proposal can handle all XPath axes, it is limited by one constraint: the input to the start state in the automaton can only be a singleton node.

Other systems tackle the order and duplicate-free problem at a lower level. The duplicate elimination and order problem in MonetDB/XQuery, for instance, is partially solved inside the staircase join itself [10]. This operator is implemented such that it generates results sorted in document order and free of duplicates. Order is preserved by using the pre/post numbering of nodes, while distinctness is guaranteed by pruning out the elements from the input context nodes for which the application of the axis step in question will return identical results. Structural joins [2] for example, used in Timber, are also implemented such that they return the result sorted either in reverse or in document order depending on the used variant. If the result is output in document order then no sort operator is needed. Holistic Twig joins [3], also implemented in Timber for evaluating XPath expressions, output a result sorted in document order. These two classes of operators, however, do not guarantee a duplicate-free result.

2.1.3 Join Reordering

Join reordering is a very important and widely used optimization technique in databases that use an algebra with joins, thus it is very natural to study its application in the XML context. One fundamental difference for ordering joins between relational and XML systems lies in the fact that XQuery returns its generated result in document order and thus requires the use of order-preserving joins. These types of joins are associative but not commutative, which reduces the number of alternative plans that can be generated by the optimizer. Moreover, adopting the general heuristic used by optimizers in relational database systems that only considers left-deep trees will limit the search space even more. These two constraints enforced in the process of generating alternative plans increase the chance of missing the optimal or suboptimal plan. The work presented in [12, 21] tackles the problem of join reordering in the Natix and Timber systems. The proposed solution is to extend the search space by considering plans that are not left-deep and using joins that are not order-preserving since order can be recovered at a relatively small cost by adding a sort operator at the appropriate position in the plan. The conclusion made by the authors is that for optimizing the evaluation of XQueries contrary to relational database systems, exploring a larger search space increases the chance to find a better plan.

2.1.4 Plan Simplification

The work on algebraic equivalences in the context of MonetDB/XQuery is presented in [9]. The paper argues that plans generated by Pathfinder are large in size which makes the application of classical rewriting techniques rather difficult and limited. The used operators, however, are simple and restricted variants of relational operators which allows the use of inference rules to infer properties for intermediate results and operators, like key, cardinality, denseness, and functional and multivalued dependencies. Given the inferred properties of a single operator in the plan, predefined equivalence rules simplify it by pruning for example some obsolete input columns, and/or replace it with a simpler less expensive operator (*e.g.*, replace a join with a projection). The multivalued dependency property is used to detect the presence of an invariable item sequence e inside a loop, which does not depend on the loop variable, and hence can be removed from the loop.

2.2 Cardinality Estimation and Cost Model

An accurate cost model is needed for an optimizer to choose the best plan. Cardinality estimation greatly affects on the accuracy of the cost model. Therefore a lot of research was devoted to the problem of cardinality estimation of XPath expressions in XML databases.

2.2.1 Cardinality estimation

The main difference in cardinality estimation between relational and XML databases lies in the fact that path queries specify structure constraints in addition to value constraints. Thus the optimizer should collect statistics about both value distribution and structural relationships between elements.

An early work on selectivity estimation of simple path expressions found in [4] estimates the number of twig matches in a node-labeled tree by using a summary data structure referred to as Correlated Subpath Tree (CST). The problem of cardinality estimation on twigs is reduced to the problem of estimating substring selectivity. The proposed technique stores count statistics about frequently occurring subpaths and maintains the correlation among the subpaths sharing the same root. The disadvantages of this approach are that the whole CST must be built before being pruned, and it does not handle wildcards.

The work in [1] proposes two different synopses, path trees and Markov tables, to summarize XML documents. A path tree represents the structure of an XML document where path tree vertices are associated with the cardinality of these nodes in the document. Markov tables store cardinality statistics for subpaths of lengths up to a certain value. The selectivity of longer paths is estimated by combining the cardinality statistics of several subpaths. Path trees and

Markov tables may grow large and hence can be summarized by replacing some non frequent vertices and subpaths by wildcards nodes and *-paths correlated with information about the deleted statistics. This technique supports only simple linear path queries.

StatiX [8] is a framework providing selectivity estimation for XQueries in the presence of an XML schema by transforming the schema such that statistics are collected at different levels of granularity. The approach uses a histogram to maintain information on both the structure and values in the document. The application of this technique, however, is restricted to a small subset of the XQuery grammar.

The technique proposed in [20] builds a two-dimensional position histogram based on the start and end labels assigned to nodes using a certain numbering scheme. The histograms are used to estimate the result sizes of path expressions that use descendant and/or ancestor steps only and can not be adopted for more general path expressions.

The Bloom histogram synopsis proposed in [19] provides efficient and accurate cardinality estimation for XPath expressions and supports updates on the underlying XML database by reflecting any change in the document through updates to the dynamic summary component. This approach, however, can handle only simple path expressions.

The XSketch synopsis proposed in [14] estimates the selectivity of complex XPath expressions over graph-structured XML data by capturing the key structural information (i.e., label path and branching) in the graphs. The construction algorithm of the XSketch synopsis employs a heuristic based on greedy forward selection. It generates a label-split graph from the XML tree by merging all XML nodes with the same label into one vertex, and then successively refines the graph by exploiting localized backward and forward stability properties in the graph. The synopsis is built using a sample of path queries which makes it dependent on the generated set. The work in [15] augments each node in the structural XSketch synopsis with distribution information on the element values in the XML graph to estimate the cardinality of path queries that also contain value-based constraints.

TreeSketch [16], another synopsis developed by the same authors, also employs a structural clustering technique. Unlike XSketch, it is based on count-stability which is a refinement of forward stability. The construction of the synopses starts by building a count-stable summary graph of an XML data tree and then incrementally merges element clusters that are closely similar in their subtree structure until the memory budget is met. XSketch and TreeSketch present two shortcomings: the construction time of the synopsis is high and updating it is also expensive. TreeSketch, however, has one advantage over XSketch: it is orders of magnitude more accurate in estimating results cardinality and needs less time to construct.

The work in [7] describes another synopsis for XML documents to estimate the selectivity of path expressions. The approach supports branching XPath queries including all axis types. The construction of the synopsis lies in translating the XML document to an SLT (Straight Line Tree) grammar by using a tree compression algorithm. An XPath query is then converted into tree automata which in combination with the SLT grammar can be used to estimate the results of the query.

Except for [7, 19], all existing approaches can not handle updates to the underlying database. Only two of the proposals described above [8, 14] support XPath queries with value constraints. None of these approaches perform well on recursive data sets, that is if they are recursion-aware.

2.2.2 Cost Model

Developing cost models for operators must take into consideration, among other parameters, the physical layout of data, the way this data is retrieved from disk, the amount of memory available, and the algorithm implementing the operators' functionality. The fact that XML operators are much more complex due to the nature of the XML data renders the prediction and modeling of the data access a hard process. Therefore constructing an accurate cost model for XML query processing is far more difficult than developing cost models for relational databases. Most research on cost-based XML optimization has only focused, as we have seen in the previous section, on the problem of cardinality estimation. Comet, described in [22], is one of the few approaches known for defining a cost model for XML.

Comet is a statistical learning technique that can be used to model the CPU cost of complex XML operators. First a set of queries and data features critical in determining the cost of the operator need to be identified, then using statistics and analytical formulas feature values are estimated. Finally Comet employs the transform regression method to learn the functional relationship between the feature values and the operator's cost. The resulting cost function is then used to estimate the cost of the operator and can be updated through a process of query feedback to adapt to changes in query workload and system environment. However this approach can only estimate the CPU cost of operators and can not be used for determining I/O cost.

2.3 Plan Enumeration and Selection

Plan enumeration in Timber consists of enumerating the join orders in the plan. The work in [21] proposes five cost-based enumeration techniques which explore the space of execution plans by reordering join operators and guarantee the choice of the optimal or suboptimal plan. The difference between the proposed algorithms is the number of plans generated for choosing the optimal one, (*i.e.*, the time spent on the enumeration of plans), and the certainty of picking the optimal plan at the end. The first two algorithms, *Dynamic Programming* and *Dynamic Programming with Pruning*, enumerate respectively all or a subset of equivalent plans and make sure the optimal one is chosen. The next two algorithms apply some pruning techniques to decrease the search space but might fail in finding the optimal plan. The fifth algorithm reduces the search space by only considering fully-pipelined plans. The latter three algorithms trade off the optimality of the chosen plan for the time spent on enumeration. The authors conclude that the choice of the enumeration algorithm should depend on the query being executed: if the query is not too expensive, then the fully-pipelined algorithm can find a good plan in a short time. If the query's execution is slow then it is best to use the *Dynamic Programming with Pruning* algorithm to produce the optimal plan.

The work in [11] proposes three different plans for evaluating XPath expressions each using one type of navigational primitive. The first technique translates every location step in the path to an Unnested-Map operator while the second and third group operations requiring expensive I/O access to

the disk into a single operator, either a XScan or a XSchedule operator. The two types of operators are responsible for scheduling and managing inter-cluster operations such that the time spent accessing the physical layer is minimized, i.e., reducing the number of times a page is loaded from hard disk, and optimizing the order in which pages are accessed. XSchedule employs an asynchronous I/O while XScan involves a sequential scan to the data. The results show that XSchedule and XScan almost always outperform the simple method, while XSchedule outperforms XScan if the query is highly selective.

3. OPEN PROBLEMS

Although a lot of work has been done in the optimization of XML queries, several issues are still unresolved. We enumerate, in this section, some of the existing open problems.

3.1 Equivalence Rewriting

XPath rewriting As XPath is a central expression in XQuery, its evaluation performance has a big impact on the overall performance of the XQuery itself. Therefore one possibility to optimize the evaluation of XQueries is by rewriting XPath expressions into equivalent more efficient ones. Some work for syntactically rewriting XPath expressions, *e.g.* [13], has already been done; however, we argue that exploiting this rewriting at the relational algebraic level is a better choice and presents several advantages.

Element construction optimization It is common in XQuery to query XML fragments that were constructed by the query itself. The fact that element construction is an expensive operation that often requires the copying of a complete XML subtree, gives rise to the XQuery optimization question: how can we remove or reduce, if possible, intermediate XML fragment construction or push computation through the construction process. We illustrate this problem with the following simple XQuery example:

```
declare function local:foo($e as element()) as element()
{    <a>{$e}</a>    };
local:foo(<b>Hello World!</b>)//b
```

In this example, the construction of the XML element <a> is unnecessary, consequently the call to the function *foo* and the navigation to the *b* elements can be removed. A query rewriter module would like to find such unnecessary constructions and navigation and eliminate them. But if the navigation in the previous example attempts to access in the results returned by the *foo* function the element *a*, then this optimization is not possible and will lead to wrong results. It is also an open question whether it is better to perform this rewriting at the XQuery core or at the algebra level.

User defined recursive functions optimization XQuery is employed not only as a query language but also as a programming language, therefore it is not surprising to see that in different types of applications recursive functions are being defined by users to query XML data. Several research proposals have tackled the problem of optimizing user-defined functions in XQuery; however, little work focused on recursive functions. One of the techniques used to minimize the overhead of user-defined functions is inlining (*i.e.*, replacing the function call by the function code). This approach can be adopted for a specific class of recursive functions only in the presence of schema information, and can not be applied in all other cases since it is undecidable a priori when to break the recursion as this, in general, depends on the value of the passed data which can not be checked by the rewriter. A question arises here: can we perform this optimization at the algebra level and if so which recursive XQuery function definitions can be represented algebraically and what are the possible techniques to optimize these generated plans.

3.2 Cardinality Estimation and Cost Model

To make sure the rewriting phase will result in generating faster plans, the decisions made during the rewriting process should be based on collected statistics and a cost model. Since the accuracy of cardinality estimation greatly impacts the accuracy of the plan's cost estimation, a lot of research has been done to develop synopses to estimate the cardinality for XPath expressions. Each technique described in Section 2.2.1 has its own shortcomings, and thus there is a need for a better, compact synopsis that meets the following criteria: returns accurate estimations, covers a large subset of XQuery and not only XPath expressions, handles efficiently updates to the document and recursive data, is constructed at a low price, and supports queries with value constraints. Furthermore it is beneficial to complement the proposed approach for cardinality estimation with a technique that estimates the distribution of the returned data, which will result in more accurate predictions of the result size of subsequent operations. In fact this information is highly valuable for operators whose cardinality estimation greatly depends on the distribution of their input data like for example aggregation, and selection.

An accurate cost model is essential for selecting a good query plan. Little research has been done for modeling the cost of operators in XML database systems. The only proposed approach known to us that addresses this issue is Comet [22]; however, it only estimates CPU costs. Hence, a suitable cost model that takes into account both the CPU and I/O costs of operators in an XML database is needed.

3.3 Plan Selection

At compile time, several parameters such as size of input, selectivity and available resources are estimated by the optimizer in order to make its choice of the plan to be executed. The accuracy of these estimations is not always guaranteed and hence the static plan chosen by the optimizer is not always optimal or close to optimal. A technique called dynamic plan selection, which has been proposed in the context of relational database systems, postpones the selection of the best plan to runtime. According to our knowledge, no research has been done on this topic in the context of XML databases. Since predicting input cardinality and operator selectivity is even harder for XML than relational data, we think one interesting approach is to see to which extent dynamic plan selection can be applied for XML. With this technique, the plan generated by the optimizer will contain several subplans connected by a special operator (*e.g.*, a choice operator), where a choice between the equivalent subplans is made at runtime. In fact, MonetDB/XQuery presents a good platform to experiment with this approach since it supports materialization of intermediate results. The choice to shift to a better and semantically equivalent subplan can be made by considering, among other things, the size of

intermediate results. Several questions arise when adopting this technique, three of which are: how is the dynamic plan created? Which optimization decisions are postponed to runtime? How effective is this technique in practice?

4. XPATH REWRITING

One open problem we currently investigate is the possibility of rewriting XPaths at the algebraic level. In this section we describe the motivation of our work and the approach we adopt to perform this rewriting in MonetDB/XQuery while keeping our technique applicable to other relational database systems.

4.1 Motivating Example

To illustrate the need and benefits of XPath rewriting, we present a very simple motivating example. Consider the XPath //a/b and an XML document that consists of 1,001,000 elements of type *a* of which 1000 have a child element *b*. The other 1,000,000 *a* elements are leaves. In such a situation, it might be cheaper to execute the XPath //b[./parent::a] instead of //a/b. We claim that this is specifically true if the execution process of the XPath requires all nodes *a* to be looked up before returning all its children *b*. Experiments on processing two pairs of equivalent XPaths over the XML document described above were done using different systems and are shown in Table 1.

Our claim proved to be true in MonetDB/XQuery: it is cheaper to retrieve 1,000 *b* elements and then check the parents of type *a* than to retrieve 1,001,000 *a* elements and then navigate to the *b* children. The difference in the Saxon system is, however, not obvious nor remarkable. The other two systems, Galax and Qizx, show different results where it seems that the existence of a predicate in the XPath makes its evaluation more expensive and this takes over the benefit gained by first retrieving the *b* elements.

The results in this experiment lead to the following conclusion: the rewriting of XPath expressions to equivalent more efficient ones is a good optimization technique that should be based on some heuristics. Different systems need different heuristics. For example, in MonetDB/XQuery it is beneficial to rewrite the XPath such that the number of elements retrieved is minimized. Once the appropriate heuristic is defined, the question that arises is how to perform this rewriting such that, based on collected statistics, it increases the system's performance.

4.2 Rewriting Approach

4.2.1 XPath Level

Proposals for rewriting XPath expressions at the syntax or core level, e.g. [13], has three disadvantages. First it can only find rewritings that are expressible in XPath, hence missing some possible more efficient execution plans. Moreover, it is harder for the optimizer to use and benefit from the available statistics. Finally, even if statistical knowledge is used, applying the defined heuristic at this level, then translating the XQuery core plan to an algebra, and afterwards applying algebraic optimization rules might lead to unpredictable plans. The work in [17] showed that in some situations the decisions taken by query optimizers can be very unpredictable and the assumptions it makes do not always hold. Consequently it is safer to exploit the XPath symmetries at the algebra level where the application of rewriting rules can be synchronized with other optimization rules.

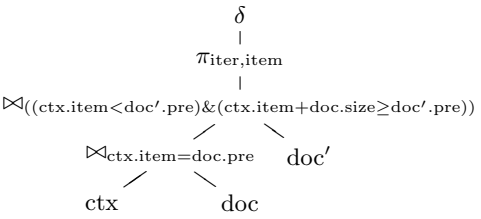

Figure 1: The relational equivalence of a descendant Staircase Join operator

4.2.2 Staircase Join Reordering

MonetDB/XQuery adopts a relational algebra extending it with, among others, the staircase join operator [10] to accelerate the evaluation of XPath location steps. Each step in an XPath expression is translated into either one or a pair of staircase joins. Therefore performing the rewriting at this level consists of reordering the staircase join operators in the plan. This might seem like a normal join reordering process but in fact is not since a staircase join (although named a join) does not share the same properties of a join:

- The staircase join is a right semijoin operator,
- It is not commutative nor associative [18].

Having these properties, staircase joins lack the flexibility that normal joins have for being easily reordered in the plan, hence the optimization possibilities offered by this approach largely coincide with those at the XQuery core level.

4.2.3 Purely Relational Rewriting

Another way to achieve the XPath rewriting idea is to perform the process at the relational level. This is accomplished by following several steps. The first consists of translating XQuery core plans to relational plans free of staircase joins by mapping XPath steps to a sequence of relational operators equivalent to a staircase join. The relational staircase join operator representing the descendant axis is shown in Figure 1. The table *ctx* consists of the context nodes from which we navigate to the next elements, and *doc* and *doc′* represent the queried XML document. Defining the sequences of relational operators equivalent for the other axis types is easily accomplished by introducing simple modifications to the plan.

The second step reorders the relational operators in the new plan by for example pushing selective operators down and striving to some normal form. Finally the last step tries to recognize the sequence of relational operators equivalent to the staircase join and maps back the ones it finds to staircase joins.

This approach has several advantages over the ones described above. First it is possible to find plans that are not expressible with the XPath syntax. A very simple rewrite would transform the plan for the XPath /a//b into one consisting of the path //b followed by a *join-based* ancestor step which uses an *index selection* returning only *a* nodes on *level* 1. Second the XPath rewriting technique we propose is general enough and can be applied to any relational database system. Moreover if dynamic plan selection is supported then the optimizer can rewrite the path expression into several equivalent ones and defer the choice of the optimal one to runtime.

	MonetDB/XQuery	Galax	Qizx	Saxon
//a/b	0.316	7.171	3.618	10.04
//b[./parent::a]	0.165	7.328	4.611	9.3
//b/parent::a	0.163	5.379	3.293	9.35
//a[./child::b]	0.541	11.247	5.308	10.09

Table 1: XPath Evaluation Time for Different XML Processing Systems

5. CONCLUSIONS

Although the topic of optimizing XQueries is getting a lot of attention from the database community, many problems are still unresolved. In this paper we have given an overview of some of the work accomplished in this field, and have identified some open problems. Two promising areas for optimization are element construction and user-defined recursive functions. The former optimization technique is to remove or reduce, if possible, unnecessary intermediate XML fragment construction and/or to push computation down the construction. The other optimization is to find the classes of user-defined recursive functions that can be compiled to algebraic operators and to define rewriting rules that optimize the generated plans.

Although a lot of work has been done to develop a synopsis for an XML document, the proposed techniques still present some shortcomings. We have enumerated the properties that a synopsis should have and we argue that the defined synopsis should return not only the cardinality of the result but also its distribution. An accurate cost model that takes into account both the CPU and I/O costs of operators in an XML database is also needed. We also propose to use a dynamic plan selection technique to delay some optimization decisions until runtime.

We are currently working on an approach to rewrite XPaths at the relational algebraic level such that more efficient plans that might not be expressible in XPath syntax can be derived. We claim that our solution is general enough and can be applied to other relational database systems, and it allows the decision of which "XPath expression" (subplan) to chose to be deferred to runtime. Moreover, this approach will offer an extra proof that the relational model and algebra are powerful enough to deal with all requirements that querying XML documents poses.

6. REFERENCES

[1] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *VLDB*, pages 591–600, 2001.

[2] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. *Int. Conf. on Data Engineering (ICDE)*, 2002.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *SIGMOD Conference*, pages 310–321, 2002.

[4] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. T. Ng, and D. Srivastava. Counting Twig Matches in a Tree. In *ICDE*, 2001.

[5] D. Draper, P. Frankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Simeon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. *W3C working draft, http://www.w3.org/TR/xquery-semantics/*, 2004.

[6] M. F. Fernández, J. Hidders, P. Michiels, J. Siméon, and R. Vercammen. Optimizing Sorting and Duplicate Elimination in XQuery Path Expressions. In *DEXA*, pages 554–563, 2005.

[7] D. Fisher and S. Maneth. Structural Selectivity Estimation for XML Documents. In *ICDE*, 2007.

[8] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. StatiX: Making XML Count. In *SIGMOD Conference*, pages 181–191, 2002.

[9] T. Grust. Purely Relational FLWORs. In *XIME-P*, 2005.

[10] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *VLDB*, pages 524–525, 2003.

[11] C.-C. Kanne, M. Brantner, and G. Moerkotte. Cost-Sensitive Reordering of Navigational Primitives. In *SIGMOD Conference*, pages 742–753, 2005.

[12] 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.

[13] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *EDBT Workshops*, pages 109–127, 2002.

[14] N. Polyzotis and M. N. Garofalakis. Statistical Synopses for Graph-Structured XML Databases. In *SIGMOD Conference*, pages 358–369, 2002.

[15] N. Polyzotis and M. N. Garofalakis. Structure and Value Synopses for XML Data Graphs. In *VLDB*, pages 466–477, 2002.

[16] N. Polyzotis, M. N. Garofalakis, and Y. E. Ioannidis. Approximate XML Query Answers. In *SIGMOD Conference*, pages 263–274, 2004.

[17] N. Reddy and J. R. Haritsa. Analyzing plan diagrams of database query optimizers. In *VLDB*, pages 1228–1240, 2005.

[18] R. Verhage. Relational Approach to XPath Query Optimization. Master Thesis, University of Twente, 2005.

[19] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Bloom Histogram: Path Selectivity Estimation for XML Data with Updates. In *VLDB*, pages 240–251, 2004.

[20] Y. Wu, J. M. Patel, and H. V. Jagadish. Estimating Answer Sizes for XML Queries. In *EDBT*, pages 590–608, 2002.

[21] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural Join Order Selection for XML Query Optimization. In *ICDE*, pages 443–454, 2003.

[22] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical Learning Techniques for Costing XML Queries. In *VLDB*, pages 289–300, 2005.