VAMANA - A Scalable Cost-Driven XPath Engine

Venkatesh Raghavan, Kurt Deschler and Elke A. Rundensteiner
Computer Science Department, Worcester Polytechnic Institute, Worcester, MA 01609
Email: (venky, desch, rundenst)@cs.wpi.edu, Tel: (508) 831–5815, Fax: (508) 831–5776

Abstract— Several systems have recently been proposed for the evaluation of XPath expressions. However, none of these systems have demonstrated both scalability with large document sizes and robust support for the XPath language. Many of the scalability problems can be attributed to inadequate use of indexing during query evaluation. While poor support for the XPath language is often a consequence of an architecture overly optimized for certain queries. Finally, the proposed systems fail to adequately address costing with respect to query optimizations.

We present VAMANA as a solution for a cost driven and scalable evaluation of ad-hoc XPath expressions. VAMANA's index-oriented query plans allow queries to be evaluated while reading only a fraction of the data. VAMANA's pipelined query framework minimizes the cost of intermediate query processing while providing cost-based transformations to further improve performance. Our experimental study confirms that VAMANA's cost-driven optimization approach for optimizing queries achieves a substantial performance improvement with negligible optimization overhead compared to non-optimized queries. Our study comparing VAMANA against several leading XML query engines demonstrates that VAMANA's query engine is significantly faster than these existing solutions in all considered cases.

Index Terms— XML, XPath, query optimization, cost estimation, indexing.

I. Introduction

Many applications are already working with or beginning to work with data in the XML format. We anticipate that the need for queries over XML data will emerge that emphasize the structural semantics of XPath and XQuery languages for querying XML data. This brings the need for an efficient query engine tailored for XML data — similar to current commercial database engines offering query support for relational data. Early on, several DOM-based query engines had been proposed for XPath evaluation [1], [2], [3]. DOM-based engines load the entire document into main-memory before query execution. The maximum document size is bounded by the amount of physical main memory [4], hence scalability is clearly an issue here.

Significant research has been focused on developing efficient storage and query strategies for XML data. These efforts can be broadly divided into two categories, namely, those storing XML documents in relational tables and those employing a native storage solution for XML documents. The relational solution is based on shredding the XML document into relational tables [5], [6]. The relational storage solution must cope with the mismatch between the relational and the semi-structural XML data model. The relational solution requires a mapping algorithm to translate the user XML query into system-specific SQL subqueries to be executed on the underlying relational database. This often is not efficient, and in fact certain XML queries cannot be easily translated into SQL [5].

Alternatively, there has been activity into developing specially tuned system support for managing XML natively in an XML repository [7], [8], [9]. Such systems must support optimized evaluation of XML queries. For this, some of these systems have began to develop special query-processing algorithms, most notably the popular path join algorithms. They do not however yet fully take advantage of the opportunities offered by native XML index support. Most importantly, thus far these systems fail to demonstrate scalability with large document sizes [7], [8] – along the line we are expected to achieve in database systems for other data models. Cost-based query optimization, which is a now cornerstone of relational database technology, has not yet been given adequate attention in the context of XML databases. While some work on histograms for XML statistics capture has been undertaken in Timber [9], complete cost models for cost-based query rewriting have not yet been explored in the literature. Also, the existing costing based on histograms would require the system to continuously maintain the statistics under updates, which may prove expensive in environments that experience rather frequent updates. A robust support to all 13 XPath axes, predicate condition like value, range and position have not yet been provided by most of the existing engines [7], [8], [9].

A. VAMANA Approach

VAMANA is our solution for a cost driven, high performance engine for XPath queries. VAMANA is built around an XML repository called Multi-Axis Storage Structure (MASS) [10], which is an efficient system for storing and indexing XML documents many gigabytes in size. VAMANA provides comprehensive support for XPath expression evaluation for all 13 XPath axes.

VAMANA employs a cost model that is independent of the schema. Query costs are obtained from the actual indexed data rather than a data dictionary and thus are always up to date and accurate. This guarantees that cost accuracy is not affected by updates, inserts and deletes that may occur in the XML data. VAMANA's costing approach has the further advantage of exact counts that can be obtained for both location steps and arbitrary text values. This degree of accuracy allows VAMANA to

decide which query transformations are likely to lead to an efficient query plan. Our costing algorithm has the capability to calculate the cost over the entire database that may contain many XML documents or can be specific to a particular XML document or even a specific point within one XML document.

Currently most of the recent XML systems [11], [7], [9] make use of structural-path join type algorithms to evaluate XML queries. In contrast VAMANA makes extensive use of indexes for query evaluation by considering index-only plans. To the best of our knowledge VAMANA is the only XML query engine that focusses on index-only plan approaches for large XML documents. Also most prevalent XPath engines [2], [3], [12], [7] only deal with ancestor, descendant or child axes, ignoring the other axes supported by the XPath standard.

VAMANA's main contributions can be summarized as follows:

1) We define a physical algebra that supports index-based style of execution for any given XPath expression, including all 13 XPath axes as well as predicates such as value, range and position predicates.
2) We propose a novel cost estimation model for accurately estimating the cost of a query plan expressed in our physical algebra. This model is supplemented by our method of efficiently gathering accurate statistics about the XML data from the underlying storage structure MASS, directly.
3) We have developed an optimizer based on a library of a large set of rewrite rules that have been adapted for our physical algebra. This is complimented by a cost-driven heuristic search strategy that constructs optimized physical plans with minimal overhead. As per our knowledge, existing XPath engines do not provide the facility to change the query plan guided by actual statistics collected directly from the stored data at run time.
4) We have successfully implemented the above ideas in the VAMANA query engine, which employs an iterative and indexed-based execution strategy.
5) We describe experiments that demonstrate the effectiveness of our cost-driven, rule-based optimizer. Query transformation is guided by the selectivity factor of operators in the query plan. This heuristic is guaranteed to always produce a query plan that has better execution time than the original query plan.
6) Furthermore, we report on our experimental study comparing VAMANA with alternative engines [2], [3], [7], [13], illustrating the practicality and effectiveness of our proposed query processing techniques. Our experiments highlight the effectiveness of the cost model in identifying operators that are expensive and should thus be optimized.

II. Related Work

In recent years many DOM-based XQuery [2], [3], [13] and XPath engines [12] have been proposed. DOM based query engines are very main memory intensive. This poses a limitation on the size of the XML document [12].

Galax [2] is a popular XML query engine developed by Bell and AT&T labs. Based on our experiments described in Section VIII, Galax does not support all 13 XPath axes and performs poorly against large XML documents. The query optimization is at the logical level and does not utilize any statistics. Jaxen [13] is an open source XPath library for Java and makes use of the conventional top-down tree traversal approach for query processing. Jaxen does not support large XML documents of sizes $\geq$ 10Mb.

TIMBER [9] is a native XML database that can store and query XML documents. The query execution in TIMBER heavily depends on structural joins. Join operations can be expensive as the query becomes more complex. Query optimization involves estimating costs of all promising sets of evaluation plans. The physical algebra for a complex query can have many nodes thus exponentially increasing the number of possible evaluation plans. The criteria for selecting a promising plan is not specified. TIMBER makes use of a two dimensional histogram called a *position histogram* to estimate cost. Maintaining such a histogram can become expensive under frequent updates of the XML document.

eXist [7] is the closest to our effort, also being a native XML database system that provides index-based query processing for XPath expressions. eXist indexes elements or attributes based on their corresponding names. This index structure facilitates the path-join algorithm used in eXist to evaluate XPath expressions. To evaluate predicate expressions that contain value comparisons, eXist requires switching back to conventional memory-based tree traversal. An XML data store is used to facilitate storage of the DOM structure. This feature only indexes top-level elements. Hence predicate expressions involving attributes, text or low-level elements will involve more than just one look-up, while in VAMANA the index structure supports value-based comparisons in one look-up. eXist currently fails to execute all XPath axes like following-sibling, previous-sibling, etc. The Xindice system [8] is another native XML database management system that creates user-defined pattern indexes for small to medium size documents $<$ 5Mb.

Natix [11] is a native XML storage structure that clusters subtrees of XML documents into small XML segments. The XML data tree is partitioned into small subtrees and each subtree is stored into a data page. To facilitate the storage of large documents, Natix makes use of *proxy objects* that maintain the record identifier for the subtrees. Natix utilizes an inverted index to efficiently support query evaluation. The Natix query engine makes use of a path join algorithm for query execution. Natix does not address cost estimation and query optimization phases in query processing.

Many XML query estimation techniques [9], [14], [15] have been proposed in recent years. Some of them extend the existing traditional database histograms for statistics gathering. In a histogram approach, the domain

for an attribute *attr* in a relation R is partitioned into *buckets* considering a uniform distribution of the data in the relation. StatiX [14] is an XML query result estimator that makes use of histograms to summarize the XML schema structure and gather statistics. Histograms need to be maintained to keep them accurate over time. This could prove expensive for frequent updates.

The system described in [15] makes use of *correlated sub − path trees* (CST), which gather statistics only of frequently occurring sub-paths or *twiglets* in the data tree. While efficient for those frequent and thus indexed paths, it may prove inefficient for applications with many ad-hoc queries.

III. Running Example

As running example, we use the XML document *auction.xml* generated by the XMark [16] benchmark. Figure 1.a shows an instance of a person element in an XML document. An XPath expression [17] is composed of a series of *location steps*. Each *location step* has three parts namely an $axis\ specifier$, a *node test* and an optional *predicate*. An $axis\ specifier$ defines the direction of the specified navigation in the XML document tree structure. In this paper we use the XPath expressions Q1 and Q2 shown in Figure 1.b as our running examples.

The context node for a location step is defined as an XML node that is currently being processed [17]. In our example, the context node of the first location step in $Q1$ and in $Q2$ is the root of the XML document. The context for the following location steps and predicate expressions are provided by their corresponding parent. For example, in $Q1$ the context for the location parent::* is provided by the XML nodes returned by $descendant :: name$.

The $axis\ specifier$ defines the relationship between the context node and the selected XML nodes. The *node test* specifies the type of elements to filter from the given axis. *Predicate* is a filter on the nodes produced by the combination of $axis\ specifier$ and *node test*. In Q2, the predicate $text() =' YungFlach'$ filters the elements returned by the location step $//name$.

a.
```
<person id="person144">
    <name>Yung Flach</name>
    <emailaddress>Flach@auth.gr</emailaddress>
     <address>
            <street>92 Pfisterer St</street>
            <city>Monroe</city>
    <country>United States</country>
            <zipcode>12</zipcode>
    </address>
    <watches>
            <watch open_auction="open_auction108"/>
            <watch open_auction="open_auction94"/>
            <watch open_auction="open_auction110"/>
    </watches>
</person>
```

b.
```
Q1: descendant::name/parent::*/self::person/address
Q2: // name[ text() = Yung Flach ]/following-sibling::emailaddress
```

Fig. 1. a. XML Document. b XPath Expressions

IV. VAMANA System Overview

VAMANA is comprised of XPath Compiler, Optimizer, Cost Estimator and Query Execution Engine components as illustrated in Figure 2.

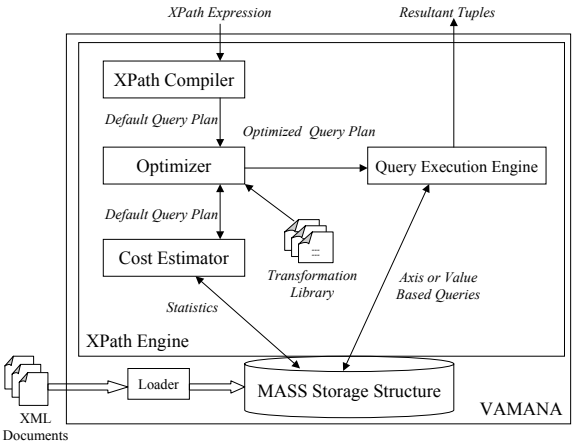

Fig. 2. VAMANA Architecture Overview

A. XPath Compiler

All XPath expressions can be logically represented as an algebraic tree structure. Each location step is translated into a parse tree node with relevant attributes like axis, node test and predicate information. The parse tree is built bottom up. Hence as the nodes are being created they are attached to its parent. The default parse trees for the expressions Q1 and Q2 are shown in Figure 3. Once the parse tree is generated we map each node to exactly one VAMANA operator to produce the physical plan.

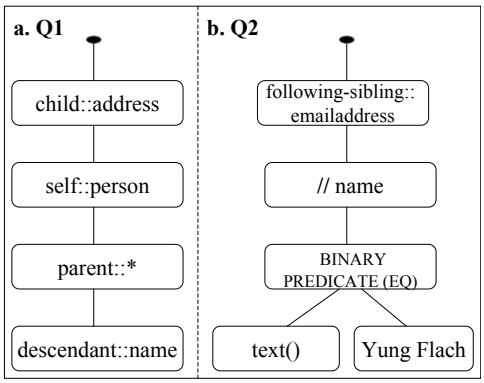

Fig. 3. Default Parse Tree for Q1 and Q2

B. VAMANA Storage Structure

VAMANA uses the MASS [10] indexing structure for all document storage and access. MASS simplifies query processing by facilitating efficient index-based access for all XPath location steps and value-based lookups. The combination of VAMANA's pipelined query operators and MASS efficient indexing allows for efficient query evaluation with minimal system resources.

MASS facilitates efficient evaluation of XPath axes, node tests, and range position predicates with use of its clustered indexes. MASS [10] uses *Fast Lexicographical Keys* (FLEX) for the structural encoding of XML nodes in the document. MASS node clustering allows efficient sequential traversal over node sets with minimal I/O and key comparisons. MASS can also count node set size for both axis-based and value-based lookups without fetching the data. MASS's efficient index lookups facilitate index-based query plans that outperform join-based plans in many cases. And the efficient counting allows VAMANA to quickly and accurately cost query plans. VAMANA shares the same node representation as MASS, which eliminates the cost of translating between node representations. Furthermore, document nodes do not need to be materialized from the persistent storage unless they are actually used in query processing. This is accomplished by passing the FLEX keys in place of the corresponding tuples.

MASS provides statistical information like number of tuples per page, number of pages, etc. used in cost estimation. The count of the number of tuples that satisfy a particular nodetest is used extensively. Thus is not expensive to calculate, because MASS's index structure facilitates count calculation from the FLEX key of the first and last node that satisfy the node test. Thus we can avoid scans by computing count on the index level without going to data.

V. Physical Algebra

A. VAMANA Physical Plan

A VAMANA $default\ query\ plan\ \mathcal{P}$ is an execution tree generated by replacing each node of the parse tree with its equivalent VAMANA algebra operator. A VAMANA operator is denoted as op_{id}^{cond}, where op is the symbol of the operator type, $cond$ represents a set of conditions applied by the operator, id is an identifier that uniquely identifies each operator, with $1 \leq id \leq m$, where m is the number of operators for a given plan $\mathcal{P}$.

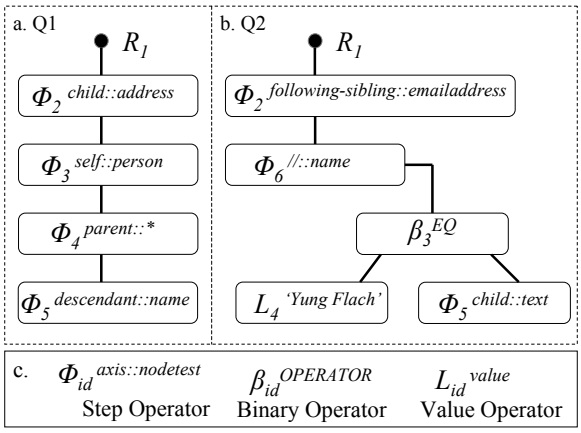

Fig. 4. VAMANA Query Plan for Q1 and Q2

We extend the idea of a context node defined in XPath to a reference to an XML node in the underlying indexed structure. In a VAMANA query plan each operator returns tuples which have structural encoding. This is used to define the context node of its corresponding parent operator. Consider the XPath expression $Q2$ (Figure 4) in which $\phi_6^{//::name}$ is defined as the *contextchild* of $\phi_2^{following-sibling::emailaddress}$.

The context node of any given VAMANA operator op_{id}^{cond} defines uniquely the position of an XML node in the index structure. The position is obtained by the structural path information encoded in the context node.

Predicate operators are used to represent XPath predicate filters [17], [18]. The predicate operator is made of a predicate condition and has one or two *predicate children*. The context node for processing a predicate operator is provided by its parent operator on which the predicate condition is evaluated. The predicate operator in return provides the context node for its leaf operators. In Figure 4.b, the operator $\phi_6^{//::name}$ being the leaf operator has no context children but one predicate child β_3^{EQ}.

Since the leaf operator has no context children, the context has to be set by the *query execution engine* before executing the query plan. In our example, the context of the leaf operator ($\phi_5^{descendant::name}$ in Q1 and $\phi_6^{//::name}$ in Q2) is set to the root of the XML document. Alternatively, in an XQuery expression [19] the leaf operator could receive context nodes from another expression.

Next, we introduce the concept of context path and predicate path which are required for dynamic context setting (Section V-B) and cost estimation (Section VI-B). A *context path* represents the path in the query plan from which the context is iteratively obtained. It is a path of operators in which each operator is the context child of the previous one. For example, the context path of the root node R_1 in Figure 4.b is context-path(R_1) = $< \phi_2^{following-sibling::emailaddress}, \phi_6^{//::name} >$. A *predicate tree* represents a sub-tree of operators starting from predicate children of a predicate operator to its leaves.

B. Dynamic Setting of Context

In VAMANA's index-based execution strategy, to start execution every operator requires a context node that uniquely refers to a particular XML node in the underlying index structure.

Leaf operators in the context path of the XPath expression are initially set by the *query execution engine* to the root of the XML document. When the leaf operator is first requested to provide tuples, it fetches the first XML node in the index structure that satisfies the conditions described in the operator. As the leaf operator is repeatedly requested for tuples, the context is dynamically moved over the index until all XML nodes in the index structure that satisfy the condition are exhausted.

The leaf operators on the predicate paths have their context set by the tuples generated by the operator on which the predicate condition filters on. For every tuple generated by the parent operator, the context of leaf operators in the predicate tree is set and then the predicate condition is evaluated.

The non-leaf operator starts execution from the context node whose information is extracted from the tuple generated by its context child. As the current XML node is processed the context node is dynamically changed to the next XML node in the index structure. When the current non-leaf operator reaches an XML node that does not satisfy its condition(s), it stops further advancement and requests the next tuple from its context child. The operator finishes execution when it has exhausted all the tuples provided by its context child.

C. VAMANA Operators

The default query plans for XPath expressions Q1 and Q2 are shown in Figure 4. VAMANA operators used for XPath specific operations are given below.

1) *Root Operator* R_1. The root operator identifies the starting point of the query plan. It has at most one context child and no predicate children. It returns all the tuples obtained from its context child.
2) *Step Operator* $\phi_{id}^{axis::nodetest}$. Each location step in an XPath expression is identified by a step operator. A step operator has at most one context child and at most one predicate operator. Each *step operator* fetches tuples from the index structure that satisfies a particular *nodetest* at a given *axis* with respect to a given context node.
3) *Literal Operator* L_{id}^{v}. A literal operator represents a literal of a particular value v [1]. A literal operator has no context child and no predicate children.
4) *Exist Predicate Operator*ξ_{id}. It denotes an exists predicate for an XPath expression. An exists predicate has one predicate child. For each tuple obtained from its parent operator, the operator applies the predicate expression as a filter condition. If the tuples condition is satisfied the parent operator is signalled to pass it to its corresponding parent. If the tuple doesn't satisfy the condition, it requests the next tuple from its parent.
5) *Binary Predicate Operator* β_{id}^{cond}. A binary predicate operator is denoted as β_{id}^{cond}, where *cond* is a logical connector operation like AND, OR, etc.. A binary predicate has two predicate children and a predicate condition. The predicate condition is applied to each of the tuples fetched by the parent operator. The binary condition is evaluated after executing both the sides of the predicate expression.
6) *Join Operator* J_{id}^{cond}. The join operator has a join condition and two context children. Tuples are fetched from both the context children and the join is applied.

VI. Optimizer

Optimization process in VAMANA is comprised of three phases, namely expression clean up, cost gathering and re-writing. Iterations of these phases are performed until the cost function of the query plan has been optimized. During optimization the query plan is transformed into an intermediate query plan by applying equivalence rules [20] from the transformation library. VAMANA optimization aims to transform the query plan such that each operator in the query plan is executed in the most optimized fashion. The query plan is incrementally optimized until the costs of all operators have been considered for optimization within the constraints of the VAMANA cost model.

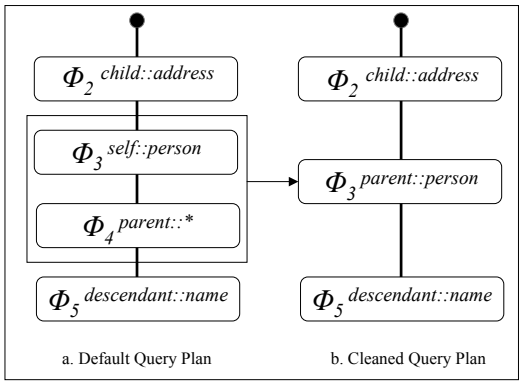

Fig. 5. Clean Up of XPath Expression $Q1$

A. Query Clean-Up

During each iteration before estimating the cost of each operator in the query plan, the *optimizer* does a clean up that targets all self axis nodes. Figure 5.a is the default query plan for $Q1$: $(descendant :: name/parent :: */self :: person/address)$. The clean up phase merges nodes $\phi_3^{self::person}$ and $\phi_4^{parent::*}$ into a single operator $\phi_3^{parent::person}$. The resultant query plan (Figure 5.b) is equivalent to the default query plan.

B. Cost Estimation

Since VAMANA uses a bottom-up execution strategy the cost estimation starts from the leaf operators of a given query plan and is propagated upwards. Consider the XPath expression $Q1$ in Figure 5.b.

At each operator op_i the statistics gathered are:

1) $COUNT(op_i)$: This statistics is only calculated for *step operators*. It represents the count of the number of XML nodes in the underlying index that satisfy the node test of the *step operator* ($\phi_i^{axis::nodetest}$). MASS provides an API to efficiently gather the count of a particular node test in its storage structure [10].
2) $TC(op_i)$: For a *literal operator* L_i^v, text count ($TC(op_i)$) is the number of occurrences of a particular literal value (v) in the index structure.
3) $IN(op_i)$: Maximum number of tuples operator op_i will receive in total from its *context child*.
   Case 1: For a leaf step operator on the context path of the query plan, the total number of tuples received is equal to the number of tuples available in the underlying index, i.e. $IN(op_i) = COUNT(op_i)$.

[1] In $Q2$ the literal operator has a value Yung Flach

TABLE I
Cost Table

AXIS	CASE	$OUTPUT(op_i)$
child, descendant, descendant-or-self	$COUNT(op_i) > IN(op_i)$	$COUNT(op_i)$
	$COUNT(op_i) \leq IN(op_i)$	
parent, ancestor, ancestor-or-self, following, following-sibling, preceding, preceding-sibling	$COUNT(op_i) > IN(op_i)$	$IN(op_i)$
	$COUNT(op_i) \leq IN(op_i)$	
self	$COUNT(op_i) > IN(op_i)$	$COUNT(op_i)$
	$COUNT(op_i) \leq IN(op_i)$	$IN(op_i)$

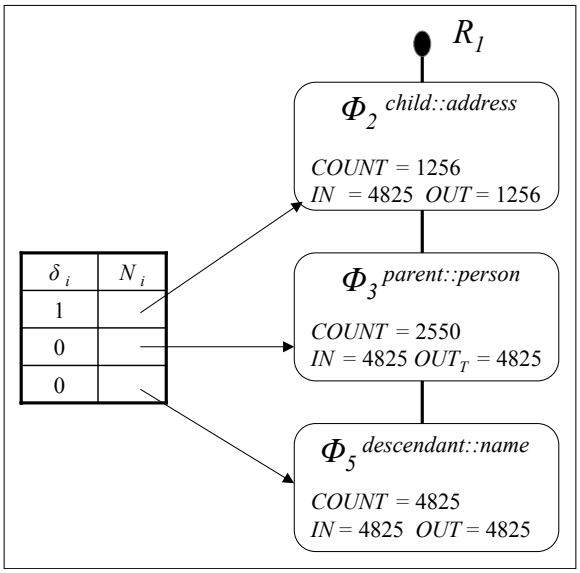

Fig. 6. Cost Estimation of Plan in Figure 5.b

Case 2: For all non-leaf operator(s), $IN(op_i) = OUT(op_j)$, where op_j is the context child of op_i.
Case 3: For all leaf step operator(s) on the predicate path of query plan, the total number of tuples received is equal to the number of tuples received by its predicate operator.

4) $OUT(op_i)$: The maximum number of tuples that the current operator op_i returns.
Case 1: A leaf step operator on the context path of the query plan returns all the tuples that occur in the index with respect to the context of the leaf operator, i.e., $OUT(op_i) = COUNT(op_i)$. For example, in Figure 6 the leaf operator $\phi_5^{//::name}$ returns all tuples satisfying the node test name starting from the root of the XML document.
Case 2: A literal operator(s) returns the same values every time a request for tuples is received. To facilitate the optimization of literal operators using a value-index, we define output as $OUT(op_i) = TC(op_i)$, where $TC(op_i)$ corresponds to the number of times a literal value occurs in the index.
Case 3: For all non-leaf step operator(s) (both context and predicate paths), $OUT(op_i)$ is calculated using the cost table shown in Table I. Consider operator $\phi_3^{parent::person}$ in Figure 6. It receives 4825 tuples from its context child $\phi_5^{//::name}$, while there are only 2550 instances of person in the XML document. This implies that the operator $\phi_3^{parent::person}$ can return at most 2550 tuples. Table I summarizes the upper bound of tuples that can be produced by a given step operator for each axis type.
Case 4: For leaf step operators on the predicate path, OUT is calculated by the cost table (Table I).
Case 5: For binary predicate operators that have a value-based equivalence, $OUT(op_i)$ is calculated as the minimum of number of tuples from the parent operator and the text count (TC) of the literal value.
Case 6: For all other predicate operators, $OUT(op_i)$ is equal to the maximum number of tuples generated by the parent operator on which the predicate expression is applied.

5) *Selectivity Ratio*: Defined as $\delta(op_i) = I_i/O_i$. After calculating the selectivity factor for all the nodes it is scaled to a ratio between the bounds of 0 and 1.

After cost estimation the optimizer generates an ordered list $\mathcal{L}$ of all operators sorted by their selectivity factor. The ordered list $\mathcal{L}(\mathcal{P})$ for a query plan $\mathcal{P}$ with m operators is defined to be an ordered array $<< op_j, \delta(op_j) > |$ where $op_j \epsilon N$ (N is the set of valid operators for the given query plan) and the pairs are sorted on the selectivity ratio $\delta(op_j) >$.

1) Running Example: The cost estimation of $Q1$ starts from the leaf operator $\phi_5^{descendant::name}$ (Figure 6). The leaf operator fetches the count of the number of XML nodes that satisfy the node test *name* in the index structure (COUNT(ϕ_5) = 4825). Based on the cost model for a leaf operator described in Section VI-B, IN(ϕ_5) = OUT(ϕ_5)= COUNT(ϕ_5) = 4825.

This cost is reflected in its parent $\phi_3^{parent::person}$ as IN(ϕ_3) = OUT(ϕ_5) = 4825. The count of the step operator is gathered in the same way as its context child. Based on the cost table described in Table I we calculate OUT(ϕ_3) = IN(ϕ_3) = OUT(ϕ_5) = 4825.

To illustrate the logic in the cost table described in Table I, consider the estimation of the operator $\phi_2^{child::address}$. IN and COUNT are gathered in the same fashion as its context child. Operator ϕ_2 has COUNT(ϕ_2) = 2550 and receives as input from $\phi_3^{parent::person}$ 4825 tuples. Since there is a smaller number of *address* than *person* and the axis is *child*, the upper bound of the number of output tuples is determined by ϕ_2. In a similar fashion the cost

estimation can be done for query $Q2$ (Figure 7).

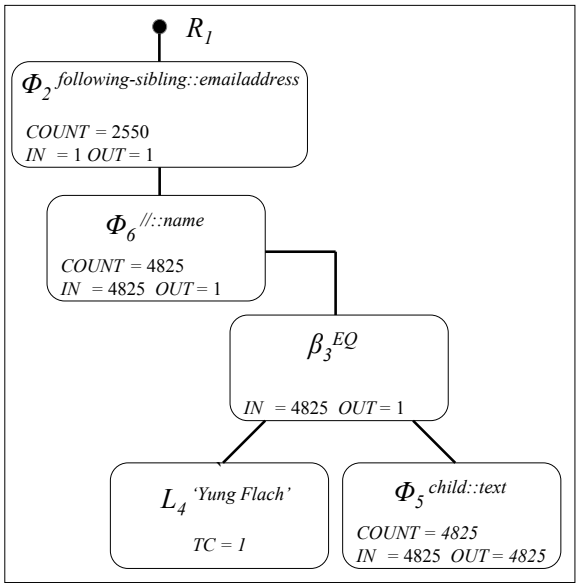

Fig. 7. Cost Estimation of XPath Expression $Q2$

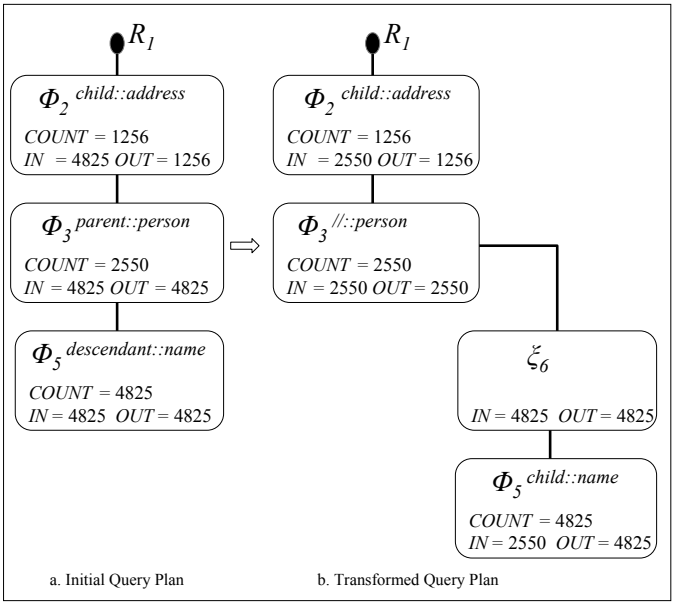

Fig. 8. Optimization of XPath Expression $Q1$

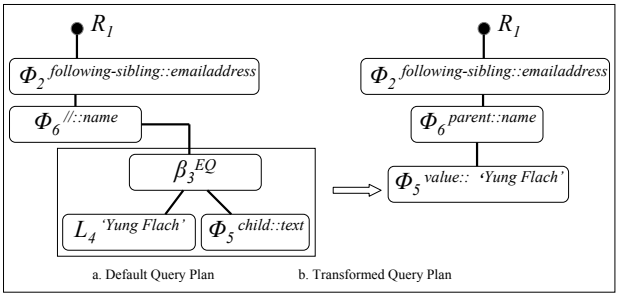

Fig. 9. Optimization of XPath Expression $Q2$

C. Optimization

Starting from the *operator* with the highest selectivity ratio, the *optimizer* examines each operator for its optimization potential. Selective operators are pushed down by applying transformation rules. The applicable transformation rule is determined by verifying the new cost that will incur if the transformation was done. If the transformation increases the cost of execution of the current operator, then that transformation rule is not considered. Increase in cost means that the transformed operator filters a lesser number of tuples. This cost estimation is done dynamically during the optimization phase. The cost involves the estimation of the transformed operator or sub-query that replaces the operator. This is inexpensive compared to costing the entire query plan. When a particular operator has been transformed, the *optimizer* repeats the process of costing and transformation.

1) Running Example Q1: Optimization of $Q1$ starts with the most selective operator $\phi_2^{child::address}$ in the ordered list $\mathcal{L}(\mathcal{P})$. Since VAMANA transformation library does not have any equivalent rules for this operator it moves on to operator $\phi_3^{parent::person}$. The optimizer finds an equivalence rule and performs the corresponding transformation (Figure 8). We then repeat the process of estimation and transformation on the modified query plan. The transformed query plan now facilitates the push-down of the most selective operator $\phi_2^{child::address}$ by applying the transformation rule to produce the query plan shown in Figure 11. Since no further valid transformation rule are available to optimize the query plan, it is considered optimal and passed for execution.

2) Running Example Q2: The initial costing for the default query plan of Q2 is shown in Figure 7. VAMANA's exploits the index by translating value-based queries into a location step. VAMANA facilitates the calculation of the text count TC for the literal operator ($L_4^{'YungFlach'}$). The XML document has only one occurrence of the value $YungFlach$. Hence β_3^{EQ} can at most return one $person$ that can satisfy the predicate condition, out of the 4825 tuples generated by its parent operator $\phi_2^{child::address}$ (Figure 9).

VII. Query Execution Engine

The execution of a query plan $\mathcal{P}$ begins by setting the context of the leaf step operators on the context path of the query plan. In the running examples it is set to be the root of the XML document. For an XQuery expression that typically contains multiple XPath expressions, the context node could be provided from another XPath expression.

A VAMANA operator during execution can be one of the following three states: INITIAL, FETCHING or OUT_OF _TUPLES. An operator is said to be in the INITIAL state when it has not yet been requested for a tuple. An operator goes into the FETCHING state either when it is fetching tuples from the underlying index structure, or when it is waiting for either its context child to fetch the next tuple to be processed, or for the predicate children to finish processing.

Algorithm 1 Execute() - Step Operator

```
Input: Step Operator φ_current;
Output: Resultant Tuple.
  while φ_current.state() != OUT_OF_TUPLES do
    if φ_current.getState() = INITIAL then
      if φ_current is a leaf node then
        φ_current.setState(FETCHING)
        return φ_current.fetchNextTuple()
        // Fetches the next node from the MASS index.
      else
        φ_current.setNextContext() // See Algorithm 2
      end if
    else
      if φ_current.getState() = FETCHING then
        T = φ_current.fetchNextTuple()
        if φ_current is a leaf node then
          return T
        else
          if T != null then
            return T
          else
            φ_current.setNextContext()
          end if
        end if
        if φ_current has a predicate child then
          if φ_current.evaluatePredicate() then
            return T;
          end if
        end if
      end if
    end if
  end while
```

An operator is in OUT_OF _TUPLES state when both of the conditions below are true.

Case 1: When all tuples that satisfy the specified condition have been extracted from the index.

Case 2: Context child (if any) has no more tuples.

Algorithm 1 explains in detail the execution

	FLEX KEYS
<site>	a
…	
<person id="person144">	a.d.y
<name>Yung Flach</name>	a.d.y.a
<emailaddress>Flach@auth.gr</emailaddress>	a.d.y.b
<address>	a.d.y.c
<street>92 Pfisterer St</street>	a.d.y.c.a
<city>Monroe</city>	a.d.y.c.b
<country>United States</country>	a.d.y.c.d
<zipcode>12</zipcode>	a.d.y.c.e
</address>	
<watches>	a.d.y.d
<watch open_auction="open_auction108"/>	a.d.y.d.a
<watch open_auction="open_auction94"/>	a.d.y.d.b
<watch open_auction="open_auction110"/>	a.d.y.d.c
</watches>	
</person>	
<person id="person145">	a.d.z
…	

Fig. 10. XML Document (Figure 1) with FLEX key

To illustrate the execution process, consider the optimized query plan for $Q1$ (Figure 11). Figure 10 represents the element person with its corresponding FLEX keys. To begin execution the *query execution engine* sets the context of the leaf step operator $\phi_2^{//::address}$ to the root [a] of the XML document . The root node R_1 goes into FETCHING state and requests its context child ($\phi_2^{//::address}$) to fetch context.

When operator $\phi_2^{//::address}$ is requested for tuples it changes its state to FETCHING and extracts the first *address* [a.d.y.c] in the index. To execute the predicate

Algorithm 2 GetNextContext() - Gets the next context from the context child

```
  child = Child()
  newContext = child.execute()
  if newContext != null then
    φ_current.resetContext(newContext)
    φ_current.setState(FETCHING)
  else
    φ_current.setState(OUT_OF_TUPLES)
  end if
```

expression ξ_7, its context node must be set to the tuple having FLEX key [a.d.y.c].

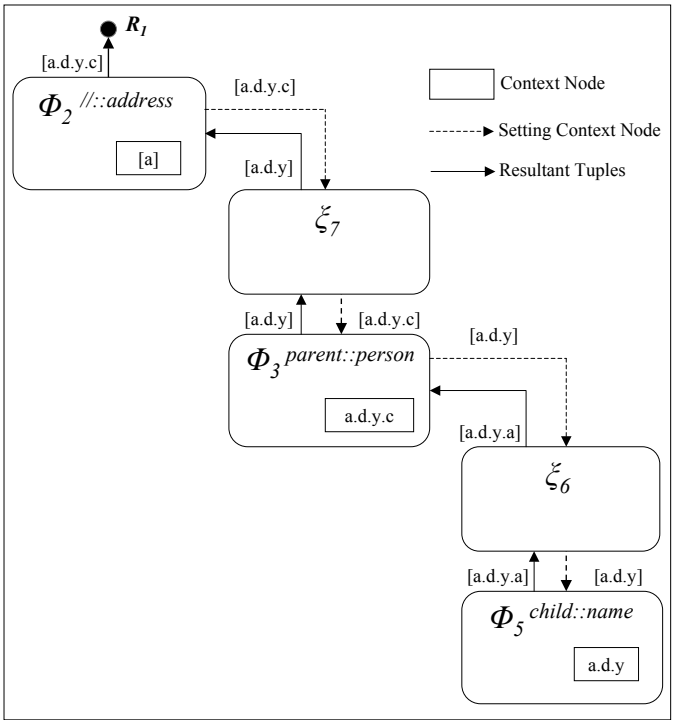

Fig. 11. Execution of a Query Plan

Once the context node of the predicate is set, the expression is evaluated. The *exist predicate operator* ξ_7 passes a request for tuples to its context child $\phi_3^{//::person}$. This operator in turn fetches the first *person* [a.d.y] who satisfies the condition axis *parent* with respect to the context node [a.d.y.c] in the index. For each *person* tuple generated, the second predicate expression (ξ_6)is executed in a similar fashion. If the predicate condition is satisfied then $\phi_3^{//::person}$ returns the tuple to its parent ξ_7 which in turn passes it to $\phi_2^{//::address}$. This signifies that the XML node having a FLEX key [a.d.y.c] satisfies the predicate condition *parent* :: *person*[*child* :: *name*]. It is then returned to R_1 to be outputted. This process is recursively done for all the tuples returned by the leaf operator $\phi_5^{//::address}$.

VIII. Experimental Studies

In this section we present our experimental evaluation of the VAMANA XPath engine using the data generated by the XMark benchmark [16]. The experiments were

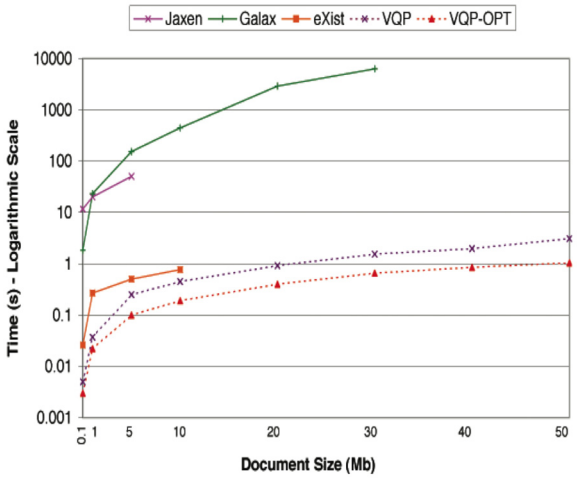

Fig. 12. Execution Time of $Q1$ in Seconds

performed on a Intel Celeron PC with 512Mb of RAM running SUSE Linux 9.0. Current XML query benchmark does not cover a wide range of XPath queries. Hence we choose the following XPath queries to cover major forward and reverse axes and predicate expressions.

- Q1 //person/address
- Q2 //watches/watch/ancestor::person
- Q3 /descendant::name/parent::*/self::person/address
- Q4 //itemref/following-sibling::price/parent::*
- Q5 //province[text()="Vermont"]/ancestor::person

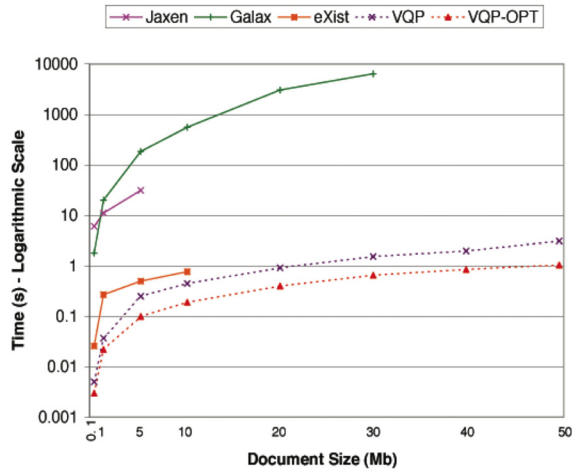

Fig. 13. Execution Time of $Q2$ in Seconds

We compare VAMANA against Galax [2] Jaxen [13] and eXist [7]. At the time of this testing, leading index-based query engines like TIMBER [9] and Natix [11] did not have a release to test our engine against. Thus eXist is the only native solution we have been able to compare against. The execution time recorded represents the total CPU elapsed time used for query execution. All query evaluations that failed to complete in two hours have no corresponding data points on the charts. Galax does not support certain axes like $following-sibling$. eXist is unable store large complex documents having sizes $\geq$ 20Mb. "VQP" represents the execution of default VAMANA query plan without optimization while "VQP-OPT" specifies the run time of optimized VAMANA query plan.

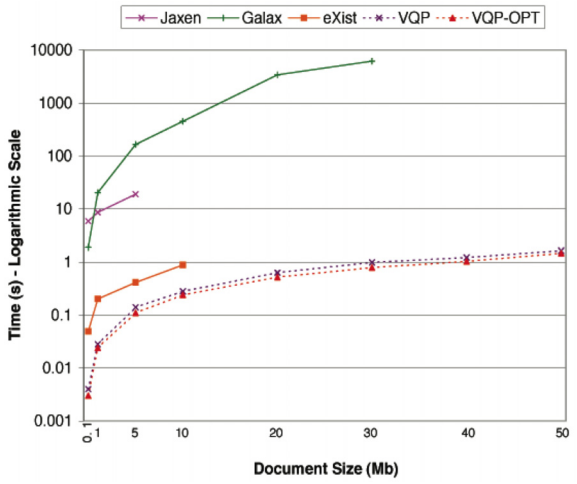

Fig. 14. Execution Time of $Q3$ in Seconds

Figures 12 and 13 show the results of running queries Q1 and Q2 on different engines. The execution of the query Q1 //person/address involves physically fetching for every *person* XML node a corresponding XML node that satisfies the condition $child :: address$. For instance, consider the XML document of size 10Mb, the total number of XML nodes with the nodetest *person* is 2550. While there exist only 1256 *address* XML nodes, thus causing twice as many fetch operations. On the other hand, the optimized query //address[parent::person] reduces the number of fetches and also exploits the capability of MASS in finding the parent XML node for any particular XML node. Now, only a check to see if the parent has a node-test *person* has to be made and thus reducing cost by at least 40%.

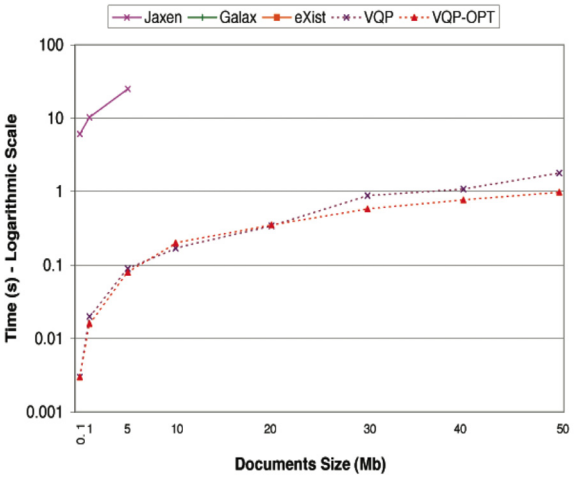

Fig. 15. Execution Time of $Q4$ in Seconds

In Q2, VAMANA optimizer reduces the execution by removing duplicates. The optimizer translates $//watches/watch/ancestor :: person$ into

$//watches[watch]/ancestor :: person$. This optimization is done only when duplicate elimination is desired. VAMANA's storage structure and execution structure facilitates evaluation of predicate condition. In comparison with eXist for query $Q5$, VAMANA performs nearly 100% faster. This is because eXist has to switch back to a tree traversal algorithm for predicate evaluation. Figure 14 experimentally confirms that the VAMANA optimizer each time generates an optimized query plan that runs faster than the default plan. The VAMANA optimizer chooses to transform a particular operator such that the number of output tuples of the current operation is reduced. This guarantees to produce a query plan that has the same or better execution time as the previous query plan.

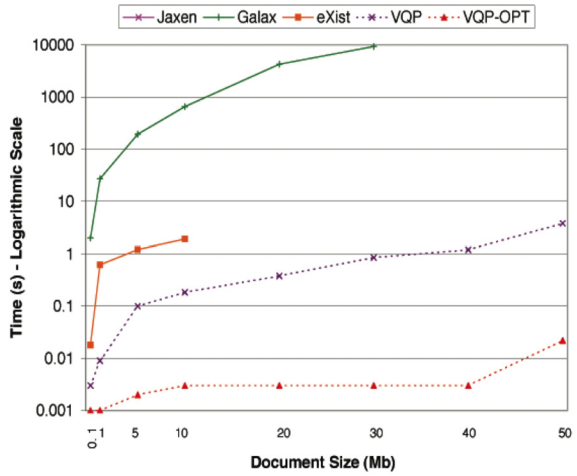

Fig. 16. Execution Time of $Q5$ in Seconds

To summarize, many of the prevalent XPath engines [2], [3], [7] only support a subset of the XPath axes. The experiments shown above illustrate that VAMANA supports all XPath axes. VAMANA exploits the large storage capacity of MASS (up to several Gbs) and can query large XML documents. Galax can only produce results in a reasonable time frame (less than two hours) for XML documents of sizes $\leq$ 30Mb. Jaxen and Xindices can handle small XML documents up to 10Mb and 5Mb respectively. We conclude that the optimizer always generates a query plan having the same or faster performance (CPU time) with respect to the default query submitted by the user. VAMANA's cost model efficiently captures the selectivity of the operators, thus aiding in transformations.

IX. Conclusion

We present VAMANA, an efficient, cost-driven XML query engine. VAMANA's index-only pipelined execution is novel to XML query evaluation as most of the existing engines use structural joins for query evaluation. The VAMANA cost model has been shown to be effective for identifying highly selective nodes. The cost-based heuristics successfully pushes selective operators down to increase execution speed. The dynamic cost estimation support makes costing up-to-date in environments even with frequent XML updates. The model is well supported by a set of transformation rules used for operator optimization. Our experiments have shown that VAMANA outperforms many existing XML query engines in both execution speed and document size.

References

[1] "Kweelt." [Online]. Available: http://db.bell-labs.com/galax/
[2] "Galax System." [Online]. Available: http://db.bell-labs.com/galax/
[3] "IPSI - XQ." [Online]. Available: http://www.ipsi.fraunhofer.de
[4] A. Marian and J. Simeon, "Projecting XML Documents," in VLDB, 2003, pp. 213–224.
[5] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian, "XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents," in VLDB, 2000, pp. 646–648.
[6] X. Zhang, M. Mulchandani, S. Christ, B. Murphy, and E. A. Rundensteiner, "Rainbow: Mapping-Driven Xquery Processing System," in SIGMOD, 2002, p. 614.
[7] W. Meier, "eXist: An Open Source Native XML Database," in Web, Web-Service and Database Systems Workshop, 2002, pp. 169–183.
[8] "Apache Xindices Project." [Online]. Available: http://xml.apache.org/xindice
[9] H. Jagadish, S. Al-Khalifa, L. Lakshmanan, A. Chapman, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, Y. Wu, and C. Yu, "TIMBER: A Native System for Querying XML," in SIGMOD, 2003, p. 672.
[10] K. Deschler and E. A. Rundensteiner, "MASS- multi Axis Storage Structure," in CIKM, 2003, pp. 520–523.
[11] C. Kanne and G. Moerkotte, "Efficient Storage of XML Data," in ICDE, 2000, p. 198.
[12] "Pathan." [Online]. Available: http://software.decisionsoft.com
[13] "Jaxen: Universal Xpath Engine." [Online]. Available: http://jaxen.org
[14] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Simeon, "Statix: Making XML Count," in SIGMOD, 2002, pp. 181–191.
[15] 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, pp. 595–604.
[16] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse, "XMark: A Benchmark for XML Data Management," in VLDB, 2002, pp. 974–985.
[17] J. Clark and S. DeRose, "XML Path Language (XPath) 1.0," Nov 1999.
[18] L. Wood, A. L. Hors, V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, G. Nicol, J. Robie, R. Sutor, and C. Wilson, "Document Object Model (DOM) 1.0," Sept 2000.
[19] S. Boag, D. Chamberlin, M.Fernandez, D. Florescu, J. Robie, and J. Simeon, "An XML Query Language (XQuery) 1.0," Oct 2004.
[20] D. Olteanu, H. Meuss, T. Furche, and F. Bry, "XPath looking forward," in EDBT Workshop on XML Data Management, 2002, pp. 109–127.