VAMANA : A High Performance, Scalable and Cost Driven XPath Engine

by

Venkatesh Raghavan

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2004

APPROVED:

Professor Elke Rundensteiner, Thesis Advisor

Professor Micha Hofri, Thesis Reader

Professor Michael Gennert, Head of Department

Abstract

Many applications are migrating or beginning to make use native XML data. We anticipate that queries will emerge that emphasize the structural semantics of XML query languages like XPath and XQuery. This brings a need for an efficient query engine and database management system tailored for XML data similar to traditional relational engines. While mapping large XML documents into relational database systems while possible, poses difficulty in mapping XML queries to the less powerful relational query language SQL and creates a data model mismatch between relational tables and semi-structured XML data. Hence native solutions to efficiently store and query XML data are being developed recently. However, most of these systems thus far fail to demonstrate scalability with large document sizes, to provide robust support for the XPath query language nor to adequately address costing with respect to query optimization.

In this thesis, we propose a novel cost-driven XPath engine to support the scalable evaluation of ad-hoc XPath expressions called VAMANA. VAMANA makes use of an efficient XML repository for storing and indexing large XML documents called the Multi-Axis Storage Structure (MASS) developed at WPI. VAMANA extensively uses indexes for query evaluation by considering index-only plans. To the best of our knowledge, it is the only XML query engine that supports an index plan approach for large XML documents. Our index-oriented query plans allow queries to be evaluated while reading only a fraction of the data, as all tuples for a particular context node are clustered together. The pipelined query framework minimizes the cost of handing intermediate data during query processing. Unlike other native solutions, VAMANA provides support for all 13 XPath axes. Our schema independent cost model provides dynamically calculated statistics that are then used for intelligent cost-based transformations, further improving performance. Our optimization strategy for increasing execution time performance is affirmed through our experimental studies on XMark benchmark data. VAMANA query execution is significantly faster than leading available XML query engines.

Acknowledgements

First, I would like to express my sincere appreciation and gratitude to my advisor Prof. Elke Rundensteiner for her guidance and valuable inputs. Her prompt feedback, ideas, suggestions and motivation made me successfully achieve my thesis goals. I would also like to thank her for guiding me as my graduate advisor throughout the M.S. (Computer Science) program at WPI. I also thank her for directing me in finding research opportunities on a NIST - WPI project EDaFS.

I thank my reader, Prof. Micha Hofri, for his time and valuable feedback.

I also would like to sincerely thank Kurt Deschler for helping me at every step of understanding the MASS system which was implemented by him. He closely worked with me by helping in designing the VAMANA architecture valuable to my thesis.

I'm thankful to all DSRG members for their encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Problem Definition

The increase in the number of applications dealing with data in the XML [1] format has brought demand for a language to extract result sets with ease from a large collection of stored XML documents. Over the recent years, several XML Query languages like XPath [2,3] and XQuery [24] have been proposed for that purpose. XPath is a World Wide Web Consortium (W3C) standard language that enables one to write expressions to identify certain parts of the XML document. XQuery on the other hand is the proposed query language by W3C focusing on XML document querying and restructuring. Hence XPath is a subset of XQuery.

This raises the need to have query engines that can efficiently process a wide variety of queries and data sets exposed in these new powerful query languages. The Document Object Model (DOM) [28] is a language independent API from W3C which is used to access the various parts of the document. Several DOM-based query engines [8,9,10] have been proposed for XPath evaluation. DOM-based engines load the entire document into main-memory before query execution. Thus they are very resource intensive. The maximum document size is bounded by the amount of physical main memory. Furthermore, the DOM structure requires complex recursive traversal for evaluation even for simple XPath expressions. The detailed comparison presented in [12] demonstrates both the document size limitations and lack of XPath support for the current DOM based systems.

In order to overcome the limitations of DOM based processing, several storage and indexing techniques have been proposed that facilitate query evaluation over persistent XML data. These systems can be divided along the line of relational mapped systems [6,7] and native XML systems [4]. The common trend for both the native and relational-backed systems to date has been to have excellent performance for certain classes of queries and limited support for the rest of the XPath language. The fundamental problem is that without adequate indexing and index-sensitive query processing, there are many queries the cannot be evaluated efficiently.

The most recent native query engines [4,13,14,15,26] have begun to address a wider variety of XPath expressions. The eXist system [13] uses four indexes and a path-join algorithm to facilitate evaluation of all XPath axes over multiple XML documents. Predicate execution is supported, but requires a less efficient tree traversal strategy. Furthermore, eXist's clustered storage comes at a cost since XML nodes from all documents must be filtered to locate nodes from a single document.

1.2 VAMANA Approach

VAMANA is our solution for robust, high performance processing for XPath queries. VAMANA is built around an XML repository called Multi-Axis Storage Structure (MASS) [4], which is an efficient system for storing and indexing XML documents many gigabytes in size.

VAMANA employs a cost model that is independent of the schema. Query costs are obtained from the actual data rather than a data dictionary and thus are always up to date and accurate. Furthermore, this guarantees that cost accuracy is not affected by updates, inserts and deletes that may occur in the XML data. Our cost model does not suffer the overhead of having to parse the entire document would be the case for histogram-based costing. VAMANA's costing system has the further advantage in that exact counts 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. VAMANA provides comprehensive support for XPath expression evaluation for all 13 XPath axes.

Currently all index-based [23,13,15] XML storage makes use of a structural path join algorithm to evaluate XML queries. In contrast VAMANA makes extensive use of indexes for query evaluation by considering index-only plans. VAMANA according to our knowledge is the only XML query engine that supports an index plan approach for large XML documents. Also most prevalent XPath engines [9,10,11,13] only deal with ancestor, descendant or child axes ignoring the other axes supported by the XPath standard. While VAMANA on the other hand supports all 13 XPath axes.

Contributions: VAMANA's main contributions can be summarized as follows:

1. We define a physical algebra that supports index-based execution for any given valid XPath expression, including all 13 axes, value predicates.

2. We describe a novel cost estimation model and describe our method for efficiently gathering accurate statistics from the underlying storage structure MASS. Our costing algorithm has the option 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 to a specific point in the XML document.

3. We illustrate empirically the effectiveness of the cost model to identify operators that are expensive and should this be optimized.

4. We present a set of transformation rules for our physical algebra based on the XPath equivalence rules stated in [4]. Our proposed transformation rules exploit properties of the underlying storage structure (MASS) to optimize value-based XPath expressions.

5. We develop an execution strategy that employs an iterative, bottom-up, indexed-based plan. While we illustrate that our execution is able to make effective use of our underlying MASS XML index statistics and its capabilities, in principle the execution algorithm is independent of our storage structure.

6. We describe experiments that demonstrate the effectiveness of our cost-driven rule based optimizer, resulting constantly improvement in execution time.

7. We compare the VAMANA execution engine with leading DOM based [8,9,10] which performs poorly against our engine and 50% faster than native XML query engines [13].

1.3 Outline

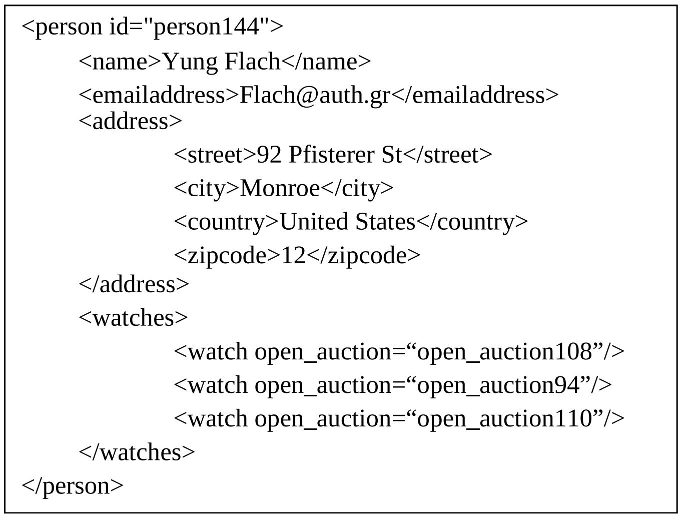
The next chapter reviews related research in query processing and cost estimation. The overall architecture of our VAMANA system is discussed in Chapter 3. Chapter 4 illustrates the steps involved in the generation of the default VAMANA query plan. In Chapter 5 we describe the VAMANA cost estimation strategy which is used in the optimization of the input XPath expression. Chapter 6 presents the VAMANA iterative index-based execution strategy. Our experimental evaluation is presented in Chapter 7 and conclusions is given in Chapter 8.

Chapter 2

Related Work

2.1 Running Example

As running example, we use the XML document *auction.xml* generated by the XMark [17] benchmark. Figure 2.1 shows an instance of a person in an XML document.



```
<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>
```

Figure 2.1: XML Document

XPath [2,3] is a W3C language that enables one to write expressions to address certain parts of the XML document. XPath is made up 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 2.2, as our running examples.

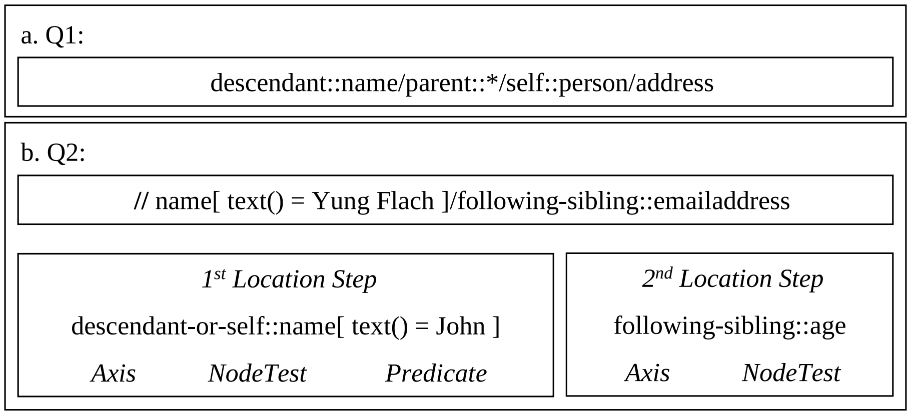

a. Q1:
descendant::name/parent::*/self::person/address
b. Q2:
// name[text() = Yung Flach]/following-sibling::emailaddress
1st Location Step
descendant-or-self::name[text() = John]
Axis
NodeTest
Predicate
2nd Location Step
following-sibling::age
Axis
NodeTest

Figure 2.2: XPath Illustration Example

The context node for an execution step in a query is defined by the XPath language as an XML node (XPath Data model [2,3]) that is currently being processed. 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 is 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* is the type name of the XML nodes from the given context node and axis. *Predicate* is a filter on the nodes produced by the combination of *axis specifier* and *node test*. In the XPath expression Q2, the predicate *text() = 'Yung Flach'* filters the elements returned by the location step *//name*.

2.2 XML Storage Structures

Significant research has been done in developing efficient storage and query strategies for XML data. They can be broadly divided into two categories those storing XML documents in relational tables and those employing a native storage for XML documents.

2.2.1 XML to Relational Mapping

Relational databases have matured over the years providing stable database management services like query processing, transaction management, concurrency control, crash recovery, etc. These facilitate the management of XML data. The relational solution is based on shredding the XML document into relational tables [6,7,24]. Many loading algorithms have been proposed [6,7,26], for example, [24] discusses many algorithms to efficiently store XML data based on a work load query.

The relational solutions require a mapping algorithm to translate the user XML query to SQL subqueries for the underlying relational data. We note that there are certain XML queries that cannot be translated into SQL [8]. The relation storage solution must cope with the mismatch between the relational and the semi-structural XML data model. Also, the relational model has no built in support for structure encoding nor for ordering. Long XPath navigations may require the execution of many joins, a rather expensive process even in a relational engine. Hence, there is a need for a native solution to efficiently store and query XML data.

2.2.2 DOM-based Solution

In recent years many DOM-based XQuery [9,10,22] and XPath evaluation engines [11] have been proposed. DOM based query engines are very main memory intensive which poses a limitation on the size of the XML document that can be processed in a reasonable time. A detailed discussion of the maximum size of the XML document handled by a DOM-based engine is found in [12].

Galax [9] is a popular XML query engine developed by Bell and AT&T labs. Based on our experiments described in Chapter 7, Galax does not support all XPath axes and performs poorly against large XML documents. The query optimization is only at the logical level and does not utilize any statistics that can be gathered from the XML document.

Jaxen [22] is an open source XPath library for Java that supports various XML API's like DOM, JDOM, dom4j and ElectricXML. Jaxen makes use of the conventional top-down tree traversal approach for query processing. Jaxen does not support large XML documents of sizes $\geq$ 10Mb.

2.2.3 XML Indices

ToX [26], developed at University of Toronto, is a repository for XML data and meta-data. ToX storage engine stores the XML documents in either a relational database or an object-oriented database. The relational solution is on the same line as discussed in Section 2.2.1. We find scanning of complete or partial documents to be expensive for query evaluation. ToX also supports storing and indexing of the DOM structure corresponding to an XML document. ToX query processor makes use of a *Path Index* and a *Value Index* to handle XPath queries.

TIMBER [15] is a native XML database that can store and query XML documents. TAX, the tree-based query algebra used in TIMBER, makes use of pattern trees to represent a query expression. The query execution in TIMBER heavily depends on structural joins. Join operations can be very expensive as the query becomes more complex. Query optimization estimates in TIMBER involve 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 plan. The criteria for selecting a promising plan is not specified. TAX operator takes as input one or more sets of trees and produces one output tree that preserves order. TIMBER makes use of a two dimensional histogram called *position histogram* to estimate cost. It becomes expensive to maintain the histogram if there are frequent updates of the XML document. It is unclear how the histogram is used to estimate the cost for XPath axes like following-sibling, previous-sibling, etc.

eXist [13] is a native XML database system that provides an index-based query processing for XPath expressions. eXist assigns a unique node identifier for all the XML elements and attributes in the XML document(s). eXist indexes elements or attributes based on its corresponding name. This index structure facilitates the path-join algorithm used in eXist to evaluate XPath expressions. Since all XML documents in a collection are indexed together, it leads to passing through an array of nodes that have the same name but belong to different documents. If the collection has large XML documents, many comparisons are required to arrive at the required element or attribute. To evaluate predicate expressions that contain value comparisons, eXist requires to switch back to conventional memory-based tree traversal. An XML data store is used to facilitate storage of the DOM [4] 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 [14] is another native XML database management system that creates user-defined pattern indexes. Xindices is developed to store and index small to medium size documents less than 5Mb. While Xindice offer more complete support for XPath, query execution and optimization is not defined.

Natix [23] 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 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.

Multi-Axis Storage Structure (MASS) [14] developed at WPI provides an efficient storage and access structure for XML documents. MASS provides an interface to retrieve node-sets from all 13 XPath axes. It also provides some statistics like number of tuples per page, number of pages, etc. In VAMANA we use MASS as the underlying storage structure (See Section 3.1).

2.3 Cost Estimation

Many XML query estimation techniques [15,16,27] have been proposed in recent years. Some of them extend the existing traditional databases 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 [16] is an XML query result estimator that makes use of histograms to summarize the XML schema structure and gathers statistics. Histograms need to be maintained to make them accurate in predicting cost. This could prove expensive if there are frequent updates to the XML documents.

[27] makes use of *correlated sub-path tree* (CST), which gathers statistics of only frequently occurring sub-path or *twiglets* occurring in the data tree. While efficient for those frequent and indexed paths this would prove not efficient for applications with many ad-hoc XPath expressions.

Chapter 3

System Architecture

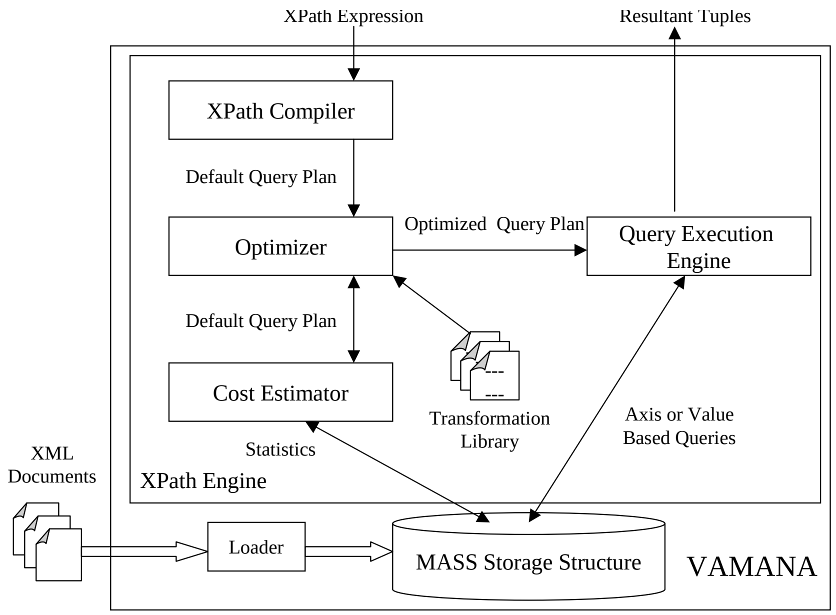


Figure 3.1: VAMANA Architecture Overview

The VAMANA system as shown in Figure 3.1 is comprised of the MASS storage structure, XPath Compiler, Optimizer, Cost Estimator and Query Execution Engine.

3.1 XML Storage Structure

VAMANA uses the MASS [4] indexing structure for all document storage and access. MASS simplifies query processing by facilitating index-based plans for all XPath location steps and value-based lookups. MASS also provides temporary storage and buffer management for VAMANA's query operators. 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 using its clustered indexes. This is true for all 13 XPath axes and both specific and wildcard ("*") node tests. 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 efficient index lookups facilitates index-based query plans that outperform join-based plans in many cases while 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 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 and is not expensive to dynamically calculate. 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.

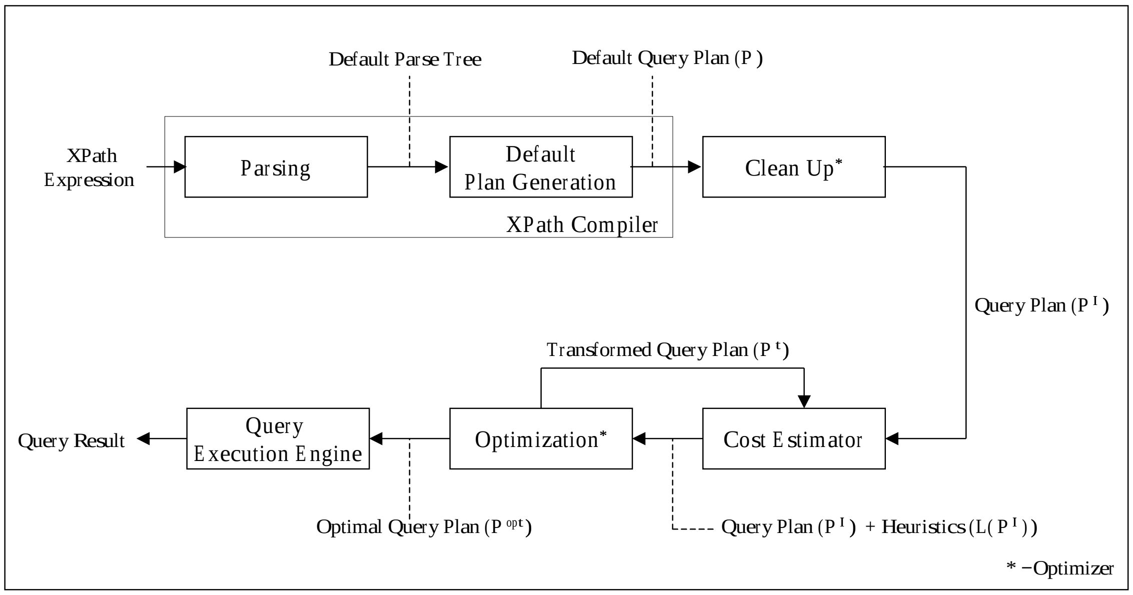


Figure 3.2: Query Processing in VAMANA

3.2 XPath Compiler

All XPath expressions can be logically represented as an algebraic tree structure. Our *XPath Compiler*[25] mimics the grammar given by the XPath Language [2,3] to transform the input XPath expression into a default parse tree. 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, so as the nodes are being created they are attached to its parent. A dummy root is attached to the generated parse tree. The default parse trees for the XPath expressions Q1 and Q2 are shown in Figure 3.3. Once the parse tree is generated we map each node to exactly one VAMANA operator to produce the physical plan. The VAMANA physical algebra is discussed in detail in Chapter 4.

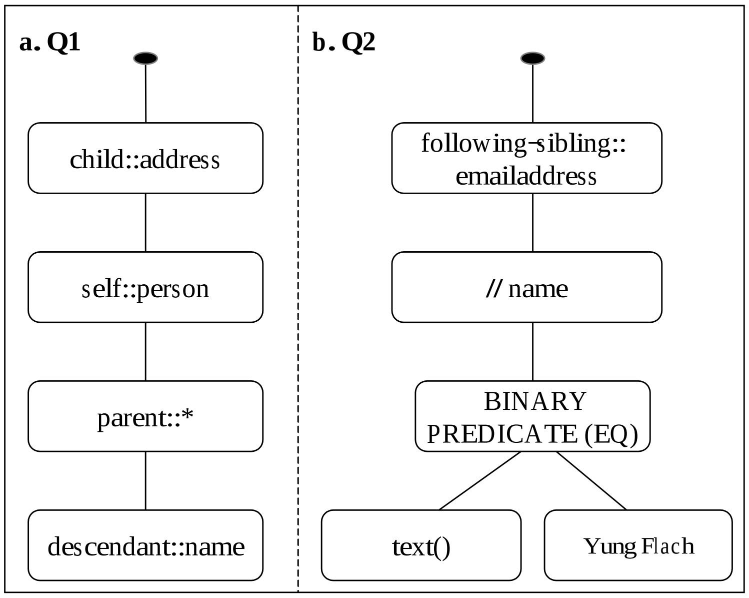


Figure 3.3: Default Parse Tree for Q1 and Q2

3.3 Cost Estimator

For each operator in the given physical query plan, the VAMANA cost estimator gathers some statistics with the help of the underlying MASS index structure and then applies computations to propagate estimations up to the tree to compute the total cost. As VAMANA follows a pipeline style of execution the cost estimation of a given query plan is started from the leaf operators to the root operator. Section 5.2 describes in depth the cost model and estimation process of VAMANA.

3.4 Optimizer

VAMANA *optimizer* (Chapter 5) consists of three subcomponents namely query clean-up, cost estimation and optimization. During query clean-up all self axis operators are eliminated. The plan is then sent to the cost estimator to gather heuristics. Based on the selectivity ratio, determined by the cost estimator, the

VAMANA optimizer tries to push selective operators down so as to reduce intermediate tuples and increase run time efficiency. This is achieved by applying XPath equivalence rules available in the transformation library. The transformed query plan undergoes several such iterations to explore for further optimization.

3.5 Query Execution Engine

VAMANA like many commercial database systems executes the query plan in a pipelined-iterative fashion to avoid temporary copies of intermediate results whenever possible. This execution strategy (See Chapter 6) facilitates the reduction of I/O operations as all tuples for a particular context node are clustered together. VAMANA adopts a *data flow* style of query execution, where the control flows downwards from the root to the leaves of the query plan and the resultant tuples are returned upwards to the root.

Chapter 4

Physical Algebra

VAMANA physical plan is designed to effectively support index-based query execution. In other XML query processing engines [6,7,13,15] each operator of their physical algebra outputs a set of tuples which are then passed to its parent for further processing. In VAMANA, on the other hand each operator in the query plan iteratively outputs tuples that are used by its parent operator in a pipelined fashion.

4.1 Notation

A VAMANA **default query plan** $\mathcal{P}$ is an execution tree generated by replacing each node of the parse tree with its equivalent VAMANA operator.

Definition 4.1.1 *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 $\mathcal{P}$.*

In an XPath language a context node is defined as the current XML node being processed. We extend the idea of a context node as a reference to an XML node in

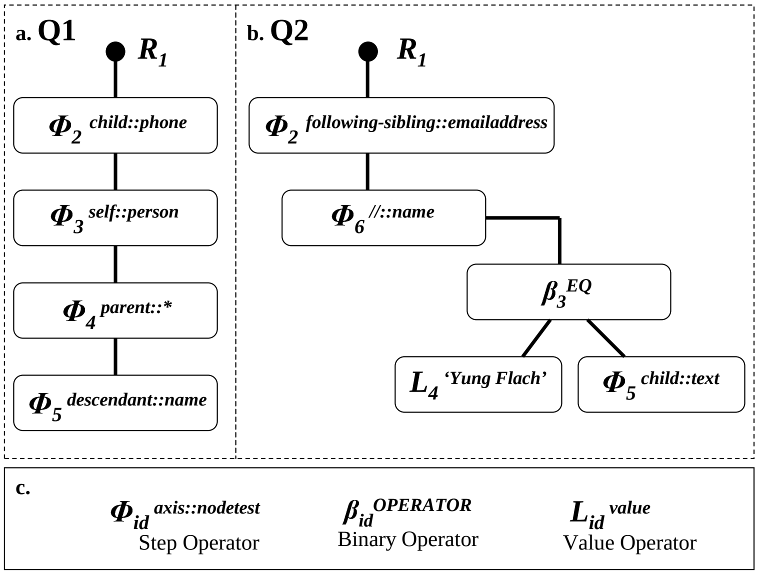


Figure 4.1: VAMANA Query Plan for Q1 and Q2

the underlying indexed structure. In a VAMANA query plan each operator returns tuples which have structural encoding. This information can be used to define the context node of its corresponding parent operator. MASS [4] uses *Fast Lexicographical Keys* (FLEX) for the structural encoding of XML nodes in the document. To illustrate consider XPath expression $Q2$ (Figure 4.1) in which $\phi_6^{//::name}$ is defined as the **context child** of $\phi_2^{following-sibling::emailaddress}$.

Definition 4.1.2 *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 [2,3]. The predicate operator can have one or two **predicate children** and a predicate condition. 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 4.1.b, the operator $\phi_6^{//::name}$ is the leaf operator and 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 [21] 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 4.2) and cost estimation discussed later in Section 5.2. A **context path** represents the path in the query plan from which the context is obtained iteratively. It is a path of operators such that each operator is the context child of the previous one. For example, the context path of the root node R_1 in Figure 4.1.b is denoted as 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.

4.2 Dynamic setting of context

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

Leaf operators in 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 operators on which the predicate conditions filter 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.

4.3 VAMANA Operators

The default query plans for XPath expressions Q1 and Q2 are shown in Figure 4.1. VAMANA operators used to perform XPath specific operations are given below.

4.3.1 Root Operator R_1

The root operator identifies the starting point of the query plan. The root operator has at most one context child and no predicate children. It returns all the tuples obtained from its context child. A root operator with no context children represents a null expression.

4.3.2 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. A step operator which is at the leaf has no context child. 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.

4.3.3 Literal Operator L_{id}^{υ}

A literal operator represents a literal of a particular value υ, in $Q2$ the literal operator has a value *Yung Flach*. A literal operator has no context child and no predicate children. A *literal operator* can only occur as a leaf operator in the predicate path of a query plan. A *literal operator* takes no input and always returns its value υ when asked for tuples.

4.3.4 Exist Predicate Operator ξ_{id}

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 satisfy the condition signal the parent operator to pass it to its corresponding parent. If the tuple doesn't satisfy the condition, it requests the next tuple from its parent.

4.3.5 Binary Predicate Operator β_{id}^{cond}

A binary predicate operator is denoted as β_{id}^{cond}, where *cond* is an operation like AND, OR, etc., that represents a logical connector. A binary predicate has two predicate children and a predicate condition. The execution of a binary predicate operator is similar to that of an exists predicate operator. 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.

4.3.6 Join Operator J_{id}^{cond}

A VAMANA join operator has *cond* as its join condition. The join operator has two context children and a join condition. Tuples are fetched from both the context children and the join is applied.

Chapter 5

Optimizer

The optimization of a query plan is comprised of three phases, namely expression clean up, cost gathering and optimization. Iterations of these phases are performed until the cost function of the query plan has been optimized for VAMANA execution model. During optimization the query plan is transformed into an intermediate query plan by applying equivalence rules [5] from the transformation library (see Section 5.3). 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 the operators have been considered for optimization within the constraints of the VAMANA cost model.

5.1 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 5.1.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 5.1.b) is equivalent to the default

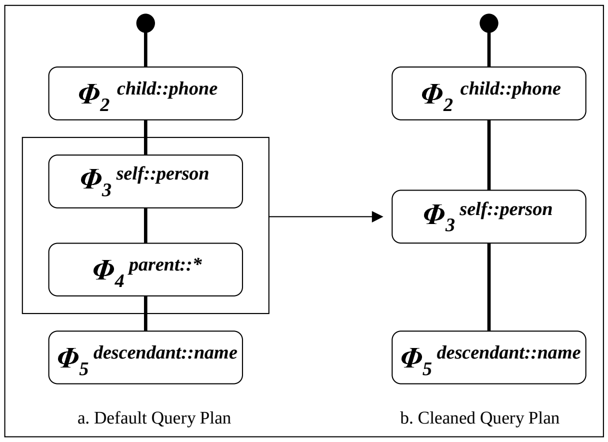


Figure 5.1: Clean Up of XPath Expression $Q1$

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	$COUNT(op_i) > IN(op_i)$	$COUNT(op_i)$
	$COUNT(op_i) \leq IN(op_i)$	$IN(op_i)$

Table 5.1: Cost Table

query plan.

5.2 Cost Estimation

Since VAMANA uses a bottom-up execution strategy the cost estimation starts from the leaf operator of a given query plan and is propagated upwards. To illustrate the cost estimation process, consider the XPath expression $Q1$ in Figure 5.1.b. where the cost estimation is started from $(\phi_5^{//::name})$.

At each operator op_i the following statistics are gathered:

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 structure that satisfy the node test of the *step operator* $(\phi_i^{axis::nodetest})$. MASS provides an

API to efficiently gather count of a particular node test in its storage structure [4].

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.

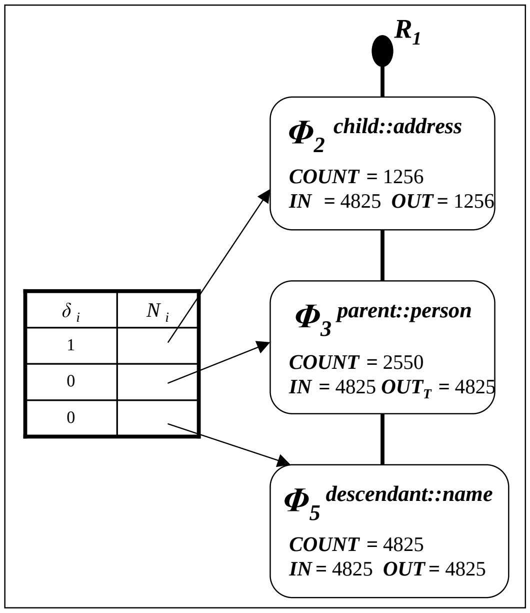


Figure 5.2: Cost Estimation of Query Plan in Figure 5.1.b

3. *IN(op_i)* : The maximum number of tuples that the 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 structure, i.e. $IN(op_i) = COUNT(op_i)$.

   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 underlying index structure with respect to the context of the leaf operator, i.e., $OUT(op_i) = COUNT(op_i)$. For example, in Figure 5.2 the leaf operator $\phi_5^{//::name}$ returns all tuples satisfying the node test *name* starting from the root of the corresponding 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 5.1.

   Consider operator $\phi_3^{parent::person}$ in Figure 5.2. 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 5.1 summarizes the upper bound of the number of tuples that can be produced by a given step operator for each particular axis type.

   Case 4: For all leaf step operators on the predicate path, $OUT(op_i)$ is calculated by the cost table shown in Table 5.1.

   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 $<< op_j, \delta(op_j) > \mid$ 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 $\delta(op_j) >$.

5.2.1 Running Example

The cost estimation of query Q1 starts from the leaf operator $\phi_5^{descendant::name}$ (Figure 5.2). The leaf operator fetches the count of the number of XML nodes that satisfy the node test *name* in the MASS index structure (COUNT(ϕ_5) = 4825). Based on the cost model for a leaf operator described in Section 5.2, 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 (COUNT(ϕ_3) = 2550). Based on the cost table described in Table 5.1 we calculate OUT(ϕ_3) = IN(ϕ_3) = OUT(ϕ_5) = 4825.

To illustrate the logic in the cost table described in Table 5.1, 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 are lesser 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 5.3).

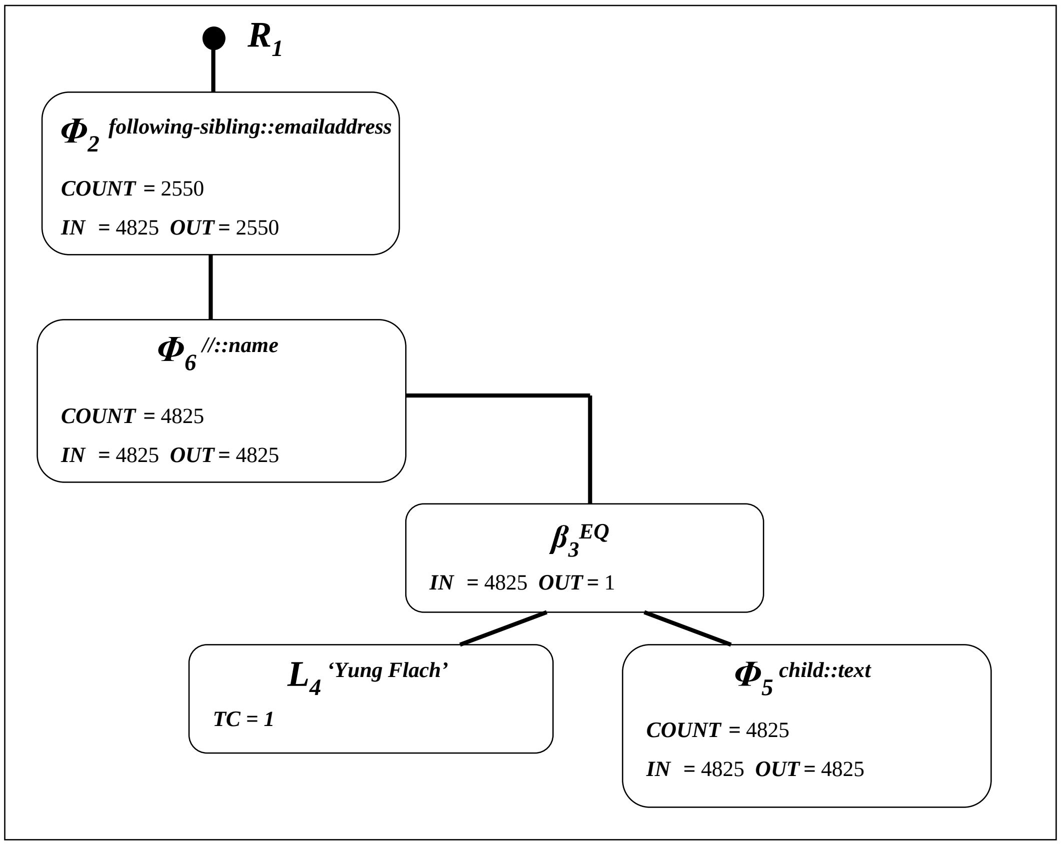


Figure 5.3: Cost Estimation of XPath Expression $Q2$

5.3 Transformation Rules

The VAMANA transformation library has a list of transformation rules that include extensions of the equivalence rules for XPath expressions stated in [5]. Some of transformation rules that are supported by the current VAMANA structure is as shown below.

- *Rule 1 : /child::m/parent::n* $\equiv$ */self::n[child::m]*
- *Rule 2 : /child::n[parent::m]* $\equiv$ */self::m/child::n*

- *Rule 3 : /descendant::n/parent::m/..* $\equiv$ *descendant-or-self::m[child::n]/..*
- *Rule 4 : /descendant-or-self::n/child::m/..* $\equiv$ *descendant-or-self::m[parent::n]/..*
- *Rule 5 : p/following-sibling::n/parent::m* $\equiv$ *p[following-sibling::n]parent::m*
- *Rule 6 : /descendant::n/parent::m* $\equiv$ *descendant-or-self::n/child::m*
- *Rule 7 : p/following-sibling::m/parent::m* $\equiv$ *p[following-sibling::m]parent::m*
- *Rule 8 : p/following-sibling::n[parent::m]* $\equiv$ *p[parent::m]/following-sibling::n*
- *Rule 9 : /descendant::n[ancestor::m]* $\equiv$ *descendant-or-self::m/descendant::n*
- *Rule 10 : p/following-sibling::n[ancestor::m]* $\equiv$ *p[ancestor::m]/following-sibling::n*
- *Rule 11 : /child::m/preceding-sibling::n* $\equiv$ *descendant::n[following-sibling::n]*
- *Rule 12 : /descendant::m/preceding-sibling::m* $\equiv$ *descendant::n[following-sibling::m]*
- *Rule 13 : /descendant::n[preceding-sibling::m]* $\equiv$ *descendant::m/following-sibling::n*
- *Rule 14 : /descendant::n[preceding::m]* $\equiv$ *descendant::m/following::n*
- *Rule 15 : /descendant::m/preceding::n* $\equiv$ */descendant::n[following::m]*

5.4 Optimization

Starting from the *operator* with the highest selectivity ratio, the *optimizer* examines each operator for its optimization potential. Expensive operators (based on selectivity ratio) are transformed into equivalent and less expensive operators by applying transformation rules discussed in Section 5.3.

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

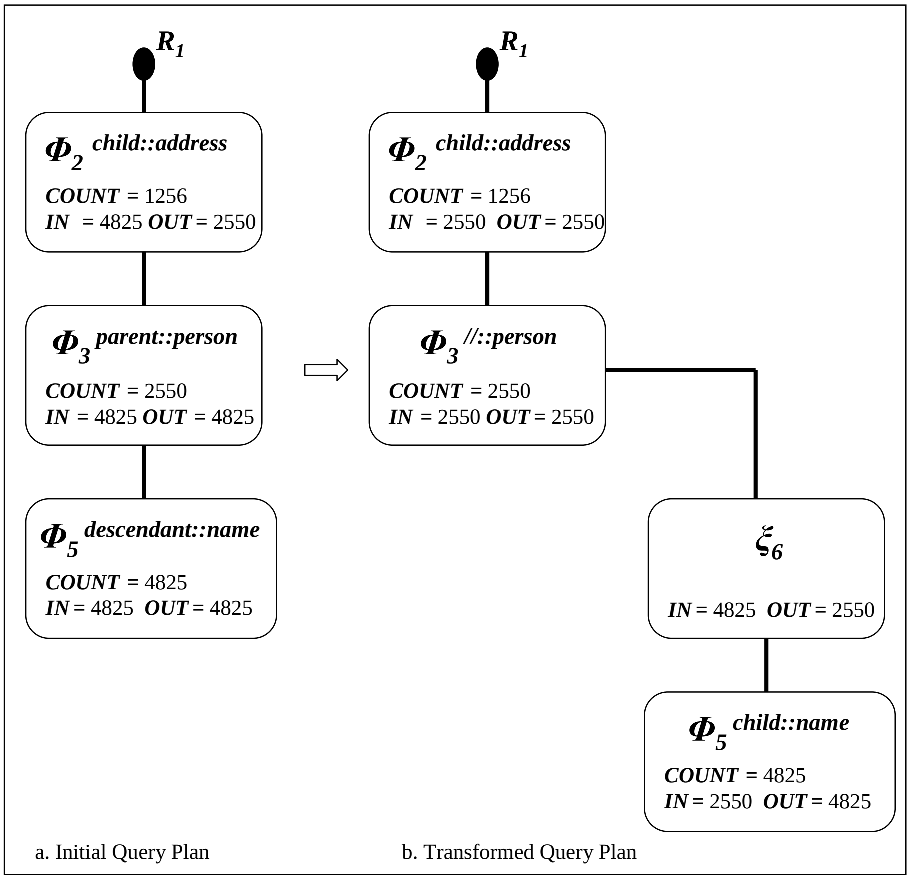


Figure 5.4: Optimization of XPath Expression $Q1$

execution of the current operator, then that transformation rule is not considered. The increase in cost is identified if the transformed operator filters a lesser number of tuples, when our aim instead is to increase its filtering capacity. 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. It is very inexpensive as compared to costing the entire query plan. The cost model of VAMANA is so designed to check the selectivity of a particular operator. When a particular operator has been transformed, the *optimizer* repeats the entire process of costing and transformation.

Algorithm 1 Optimize()

```
Input:$\mathcal{P}$.
Output: Optimal Execution Tree $\mathcal{P}^{opt}$
  CostEstimate() //populates $\mathcal{L}(\mathcal{P})$
  for all operator $op_i$ in $\mathcal{L}(\mathcal{P})$ do
    if ruleExist($op_i$) then
      reOptimize = OptimizeNode($op_i$)
      //Returns ‘true’ if any transformation done
    end if
    if reOptimize then
      Optimize()
    end if
  end for
```

5.4.1 Problem of Termination

For any given VAMANA query plan $\mathcal{P}$ and given XML document(s) we have to proof that the optimization phase is terminating.

Proof

Let the VAMANA query plan be denoted as $\mathcal{P}$. The cost estimation is based on the given XML document or set of XML documents indexed in the MASS storage structure.

During optimization, the plan $\mathcal{P}$ is transformed into $\mathcal{P}'$ if and only if the selectivity the optimized operator (in $\mathcal{P}$) increases. Selectivity of the operator directly affects number of output tuples or the total number of intermediate tuples which would need to be processed during the execution of the query plan. When the operator produces lesser number of tuples this reduction is propagated to its parent operator, thus reducing overall output tuple count. In some cases operator may not be able to reduce the overall output tuple count of the plan. But since it reduces the number of tuples it affects the total number of intermediate tuples being processed by the plan.

We can thus conclude that a plan $\mathcal{P}$ is transformed into $\mathcal{P}'$ if and only if the number of output tuples that the plan generates is reduced or the number of intermediate tuples are decreased.

From lexicographic measure, we have m for a given plan P and XML document(s) as $m(\mathcal{P}) = < m_1(\mathcal{P}), m_2(\mathcal{P}) >$, where $m_1(\mathcal{P}) =$ Total number of output tuples and $m_2(\mathcal{P}) =$ Total number of intermediate tuples.

A VAMANA query plan $\mathcal{P}$ can be transformed in to $\mathcal{P}^{'}$ if and only if.

1. $m_1(\mathcal{P}) > m_1(\mathcal{P}^{'})$ or

2. $m_1(\mathcal{P}) = m_1(\mathcal{P}^{'})$ and $m_2(\mathcal{P}) > m_2(\mathcal{P}^{'})$

Lemma 1, If $\mathcal{P} \rightarrow \mathcal{P}^{'}$ by applying a transformation then $m_1(\mathcal{P}) > m_1(\mathcal{P}^{'})$

The number of elements in the given document is fixed and is a positive integer value. Hence the total number of output tuples cannot be decreasing for ∞ iterations. Similarly the total number of intermediate tuples cannot be decreasing infinitely.

Corollary, there does not exist ∞ sequence $\mathcal{P} \rightarrow \mathcal{P}^{'} \rightarrow \mathcal{P}^{''}$...

We thus prove that the optimization strategy used in VAMANA is terminating.

5.4.2 Running Example Q1

The optimizer starts optimization of query $Q1$ from 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 (*Rule: 3*) and performs the corresponding transformation as shown in Figure 5.4.

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 (*Rule: 4*) to produce the query plan shown in Figure 5.5. Since no further valid transformation

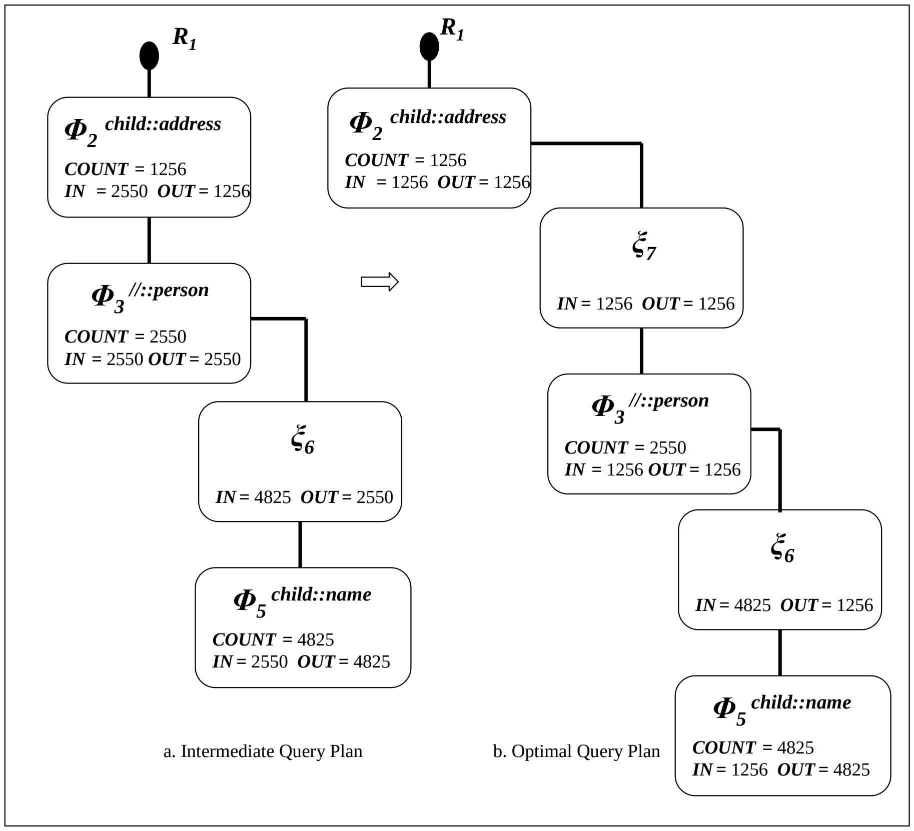


Figure 5.5: Optimization of XPath Expression $Q1$

rules is available to optimize the query plan, it is considered it optimal and passed to VAMANA *query execution phase* for evaluation.

5.4.3 Running Example Q2

The initial costing of the default query plan for the XPath expression Q1 is shown in Figure 5.3. VAMANA's optimization strategy exploits the underlying index structure by translating value-based queries into a location step based on a value. To illustrate, 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 $\phi_2^{child::address}$.

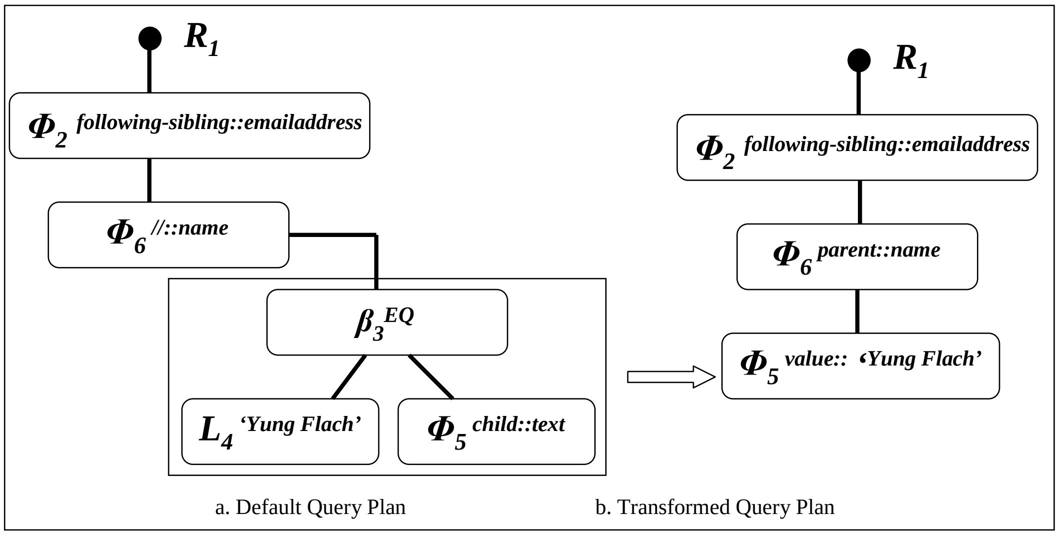


Figure 5.6: Optimization of XPath Expression $Q2$

Figure 5.6 describes the transformation of binary predicate operator β_3^{EQ} into a location step $\phi_6^{//::name}$.

Chapter 6

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 for example the context of the leaf step operator 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.

```
                                                                 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
…
```

Figure 6.1: XML Document (Figure 2.1) with FLEX key

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

Algorithm 2 Execute() - Step Operator

```
Input: Step Operator $\phi_{current}$;
Output: Resultant Tuple.
  while $\phi_{current}.state()$ != OUT_OF_TUPLES do
    if $\phi_{current}.getState() = INITIAL$ then
      if $\phi_{current}$ is a leaf node then
        $\phi_{current}.setState(FETCHING)$
        return $\phi_{current}.fetchNextTuple()$
        // Fetches the next node from the MASS index.
      else
        $\phi_{current}.setNextContext()$ // See Algorithm 3
      end if
    else
      if $\phi_{current}.getState() = FETCHING$ then
        T = $\phi_{current}.fetchNextTuple()$
        if $\phi_{current}$ is a leaf node then
          return T
        else
          if T != null then
            return T
          else
            $\phi_{current}.setNextContext()$
          end if
        end if
        if $\phi_{current}$ has a predicate child then
          if $\phi_{current}.evaluatePredicate()$ then
            return T;
          end if
        end if
      end if
    end if
  end while
```

An operator goes into the **OUT_OF_TUPLES** state when both of the conditions below are true.

Case 1: When all possible tuples have been extracted from storage with respect to a particular context and satisfying the specified condition.

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

Algorithm 2 explains in detail the execution of a step operator.

Algorithm 3 GetNextContext() - Gets the next context from the context child

```
child = $Child()$
newContext = child.execute()
**if** newContext != null **then**
    $\phi_{current}.resetContext(newContext)$
    $\phi_{current}.setState(FETCHING)$
**else**
    $\phi_{current}.setState(OUT\_OF\_TUPLES)$
**end if**
```

To illustrate the execution process, consider the optimized query plan for $Q1$ generated by VAMANA *optimizer*. Figure 6.1 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 (having FLEX key [a]) of the XML document (as shown in Figure 6.2). 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 storage structure. In the query plan, the operator $\phi_2^{//::address}$ has a predicate filter, ξ_7. To execute the predicate expression, its context node must be set to tuple having FLEX key [**a.d.y.c**] generated by $\phi_2^{//::address}$.

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 is 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

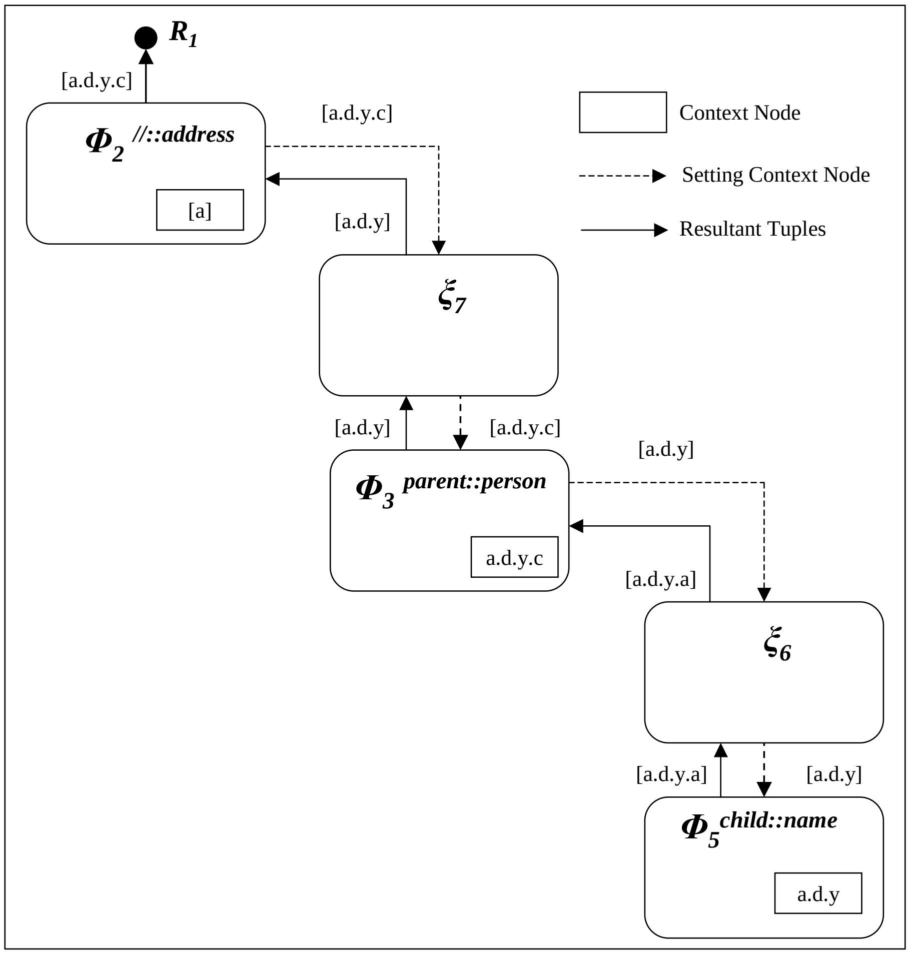


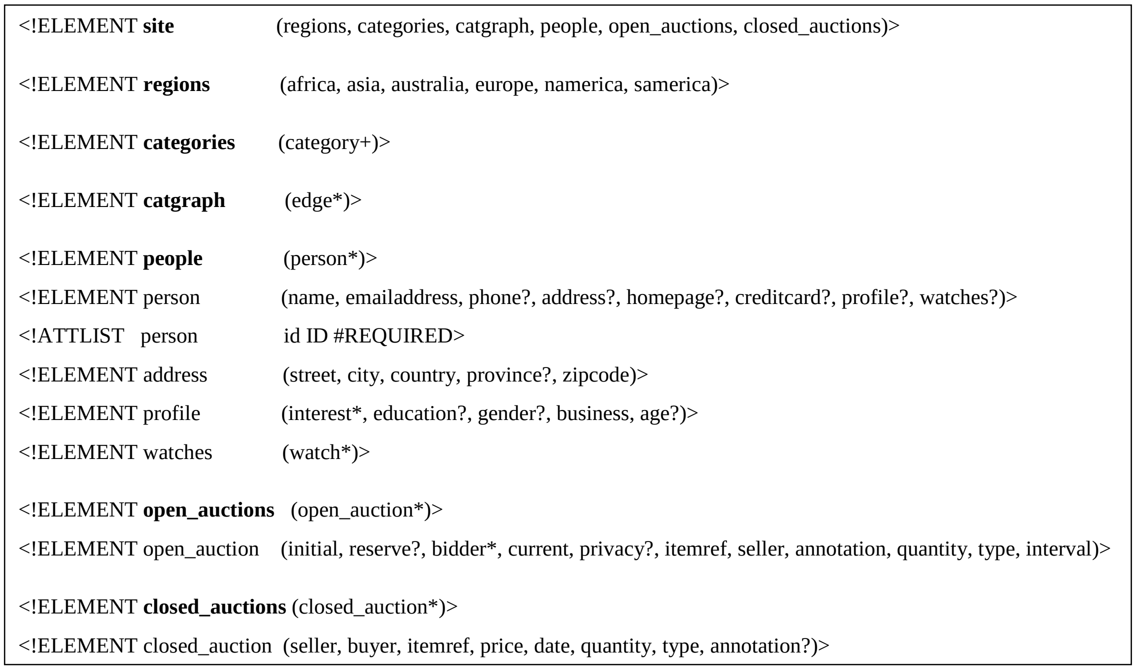
Figure 6.2: Execution of a Query Plan

recursively done for all the tuples returned by the leaf operator $\phi_5^{//::address}$.

Chapter 7

Experimental Studies

7.1 Setup



```
<!ELEMENT site              (regions, categories, catgraph, people, open_auctions, closed_auctions)>

<!ELEMENT regions           (africa, asia, australia, europe, namerica, samerica)>

<!ELEMENT categories        (category+)>

<!ELEMENT catgraph          (edge*)>

<!ELEMENT people            (person*)>
<!ELEMENT person            (name, emailaddress, phone?, address?, homepage?, creditcard?, profile?, watches?)>
<!ATTLIST   person          id ID #REQUIRED>
<!ELEMENT address           (street, city, country, province?, zipcode)>
<!ELEMENT profile           (interest*, education?, gender?, business, age?)>
<!ELEMENT watches           (watch*)>

<!ELEMENT open_auctions   (open_auction*)>
<!ELEMENT open_auction    (initial, reserve?, bidder*, current, privacy?, itemref, seller, annotation, quantity, type, interval)>

<!ELEMENT closed_auctions (closed_auction*)>
<!ELEMENT closed_auction  (seller, buyer, itemref, price, date, quantity, type, annotation?)>
```

Figure 7.1: Partial Schema Representation of XMark Auction Database

In this section we present our experimental evaluation of VAMANA XPath engine using the data generated by the XMark [17] benchmark. The schema of the

XMark auction database is partially described in the DTD shown in Figure 7.1. We generated XML documents using the XMark benchmark for eight scaling factors, 0.001 (113Kb), 0.01 (1.11Mb), 0.05 (5.6Mb), 0.1 (11.3Mb), 0.2 (22.8Mb), 0.3 (34Mb), 0.4 (45.3Mb) and 0.5 (56.2Mb). The experiments where performed on a Intel Celeron PC with 512MB of RAM running SUSE Linux 9.0. Currently there does not exist a well documented benchmark for XPath queries. We hence choose our XPath queries based on the following factors.

- Cover major forward and reverse XPath axes. Also cover predicate expressions like position, range and equivalence.
- Experimentally demonstrate the correctness of the execution strategy.
- Empirically verify that optimization does not make the query more expensive

The queries considered for our experiments are stated as follows.

- 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

We compare VAMANA against IPSI [10], Galax [9] Jaxen [22] and eXist [13]. Out of which IPSI does not support many of the axes (Q2, Q4 and Q5) and performs very poorly for queries Q1 and Q3 in comparison to other engines. The leading index-based query engines like TIMBER [15] and Natix [23] do not have a release so as to test our engine against. We compare against eXist which is a native solution

	XMark scaling factor							
Engine	0.001	0.01	0.05	0.1	0.2	0.3	0.4	0.5
IPSI	11.82	18.49	DNF-MO	DNF-MO	DNF-MO	DNF-MO	DNF-MO	DNF-MO
Jaxen	11.70	20.14	50.59	DNF-MO	DNF-MO	DNF-MO	DNF-MO	DNF-MO
Galax	1.85	23.36	154.08	446.29	2918.02	6331.33	DNF-2hrs	DNF-2hrs
eXist	0.009	0.145	0.227	0.403	X-Map	X-Map	X-Map	X-Map
VQP	0.002	0.021	0.12	0.23	0.49	0.89	1.08	1.55
VQP-OPT	0.002	0.009	0.10	0.049	0.21	0.35	0.59	0.64

Table 7.1: Execution Time of Q1 in Seconds

for XML storage. The execution time recorded represents the total CPU elapsed time (user and system) used for execution of the query. All query evaluations that failed to complete in two hours of CPU time are represented as "DNF-2hrs" ("Did Not Finish - With in two hours") and queries that require more memory than available are denoted as "DNF-MO"("Did Not Finish - Memory Overflow"). Galax does not support certain axes like *following-sibling* denoted in our charts as "NS" ("Not Supported"). eXist is unable to map into their storage structure large complex documents having sizes $\geq$ 20Mb and we denote it in our experiment as "X-Map". "VQP" represents the execution of default VAMANA query plan without optimization while "VQP-OPT" specifies the run time of optimized VAMANA query plan over the underlying MASS structure.

7.2 Correctness of Execution Strategy

We verify the correctness of the execution strategy by comparing the output of each query over different query engines. We physically compared the results of the queries over smaller XML document (100Kb,1Mb). For the rest of the documents we compared the counts of the result size, which was observed to be consistent over all the engines.

	XMark scaling factor							
Engine	0.001	0.01	0.05	0.1	0.2	0.3	0.4	0.5
IPSI	NS	NS	NS	NS	NS	NS	NS	NS
Jaxen	6.04	11.065	31.205	DNF-MO	DNF-MO	DNF-MO	DNF-MO	DNF-MO
Galax	1.80	19.95	184.98	557.31	3072	6419.77	DNF-2hrs	DNF-2hrs
eXist	0.026	0.269	0.501	0.770	X-Map	X-Map	X-Map	X-Map
VQP	0.005	0.037	0.25	0.45	0.92	1.54	1.98	3.10
VQP-OPT	0.003	0.022	0.10	0.19	0.40	0.657	0.8531	1.045

Table 7.2: Execution Time of Q2 in Seconds

7.3 Improved Performance

Tables 7.1 and 7.2 show the results of running query Q1 and Q2 over 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 2550 *address* XML nodes, thus causing twice as much 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. The path information of any XML node in the MASS index structure contains in it the path information of its parent which can be extracted with ease. Now, only a check to see if the parent has a node-test *person* has to be make and thus reducing cost by at least 40%.

In query 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 by the user.

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

	XMark scaling factor							
Engine	0.001	0.01	0.05	0.1	0.2	0.3	0.4	0.5
IPSI	11.10	18.96	DNF-MO	DNF-MO	DNF-MO	DNF-MO	DNF-MO	DNF-MO
Jaxen	5.87	8.597	18.91	DNF-MO	DNF-MO	DNF-MO	DNF-MO	DNF-MO
Galax	1.88	20.30	165.09	450.70	3413.01	6176.26	DNF-2hrs	DNF-2hrs
eXist	0.049	0.201	0.412	0.885	X-Map	X-Map	X-Map	X-Map
VQP	0.004	0.028	0.14	0.28	0.63	0.99	1.22	1.64
VQP-OPT	0.003	0.024	0.11	0.24	0.52	0.79	1.03	1.46

Table 7.3: Execution Time of Q3 in Seconds

	XMark scaling factor							
Engine	0.001	0.01	0.05	0.1	0.2	0.3	0.4	0.5
IPSI	NS	NS	NS	NS	NS	NS	NS	NS
Jaxen	6.08	10.30	25.01	DNF-MO	DNF-MO	DNF-MO	DNF-MO	DNF-MO
Galax	NS	NS	NS	NS	NS	NS	NS	NS
eXist	DNF	DNF	DNF	DNF	DNF	X-Map	X-Map	X-Map
VQP	0.003	0.020	0.09	0.17	0.34	0.88	1.08	1.79
VQP-OPT	0.003	0.016	0.08	0.20	0.35	0.58	0.77	0.98

Table 7.4: Execution Time of Q4 in Seconds

algorithm for predicate evaluation.

7.4 Correctness of Optimization Strategy

Table 7.3 experimentally confirms that the VAMANA optimizer each time generates an optimized query plan that runs faster 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.

7.5 Summary of Experimental Evaluation

Many of the prevalent XPath engines [9,10,13] only support a subset of the XPath axes. Galax currently does not support the *following-sibling* axes. The experiments shown above illustrate that VAMANA supports all XPath axes.

	XMark scaling factor							
Engine	0.001	0.01	0.05	0.1	0.2	0.3	0.4	0.5
IPSI	NS	NS	NS	NS	NS	NS	NS	NS
Jaxen	NS	NS	NS	NS	NS	NS	NS	NS
Galax	1.99	27.09	195	659.18	4282.57	9342.31	DNF-2hrs	DNF-2hrs
eXist	0.018	0.620	1.206	1.92	X-Map	X-Map	X-Map	X-Map
VQP	0.003	0.009	0.10	0.184	0.38	0.85	1.194	3.82
VQP-OPT	0.001	0.001	0.002	0.003	0.003	0.003	0.003	0.022

Table 7.5: Execution Time of Q5 in Seconds

VAMANA exploits the large storage capacity of MASS (up to several Gbs) and can query large XML documents. While Galax can 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 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.

Chapter 8

Conclusion and Future Work

8.1 Summary

We present VAMANA, an efficient, cost-driven XML query execution engine (XPath). The VAMANA physical algebra effectively supports execution of all 13 XPath axes and predicate expressions like value, position and range conditions. 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 index-only plan execution facilitate non-materialization of intermediate results.

The dynamic cost estimation support makes costing up-to-date in environments which have frequent XML updates. The costing can be performed over the entire database or for a particular XML document or a specific point in the document. The VAMANA cost model is simple but very effective for identifying highly selective nodes. And hence can be pushed down to increase execution speed. The model is well supported by a set of transformation rules used for operator optimization. VAMANA exploit properties of the underlying storage structure (MASS) to optimize value-based XPath expressions. Our experiments have shown that VAMANA outperforms currently available DOM-based and index-based XML query engines in both execution speed and document size.

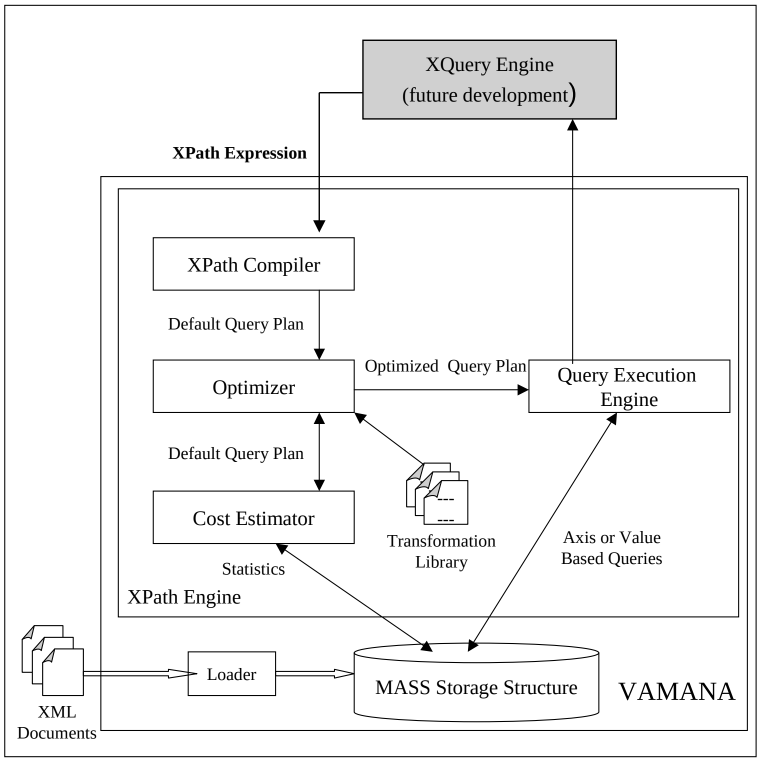


Figure 8.1: Future Research in VAMANA

8.2 Future Research

Unlike other XML query processing that start from the logical level of query execution to physical execution, we started our design and implementation ground up. We here built a robust XPath engine called VAMANA around the already existing storage structure MASS.

The work done in this thesis forms a building block for the next phase of developing a full-fledged XQuery Engine. VAMANA currently implements only a subset of the XPath equivalence rules stated in [5]. VAMANA transformation library can extended to support all the transformation rules. The VAMANA cost model currently does not include estimation of join operations, leaving scope for future research into proving that index-based plans are on par with structural join algorithms. The current cost model can be exploited and extended to support for XQuery optimization.

Chapter 9

References

1. T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler and F. Yergeau. *Extensible Markup Language (XML) 1.0 W3C Recommendation.* Available at http://www.w3.org/TR/2004/REC-xml-20040204/, Feb 2004.

2. J. Clark and S. DeRose. *XML Path Language (XPath) 1.0, W3C Recommendation.* Available at http://www.w3.org/TR/xpath, 2002.

3. A. Berglund, S. Boag, D. Chamberlin. M. F. Fernández, M. Kay, J. Robie and J. Siméon, *XML Path Language (XPath) 2.0, W3C Working Draft.* Available at http://www.w3.org/TR/xpath20/, Nov 2003.

4. K. Deschler and E. Rundensteiner. *MASS- Multi Axis Storage Structure*, CIKM, New Orleans, USA, Nov 2003.

5. D. Olteanu, H. Meuss, T. Furche, and F. Bry. *Symmetry in XPath.* Proceedings of Seminar on Rule Markup Techniques, no. 02061, Schloss Dagstuhl, Germany, Feb 2002.

6. M. Carey, J. Kiernan, J. Shanmugasundaram, E. Shekita and S. N. Subramanian. *XPERANTO: Middleware for Publishing Object-Relational Data as*

XML Documents, The VLDB Journal, pages 646-648, 2000.

7. X. Zhang, M. Mulchandani, S. Christ, B. Murphy and E. Rundensteiner. *Rainbow: Mapping-Driven XQuery Processing System*, SIGMOD, page 614, 2002.
8. Kweelt. From http://kweelt.sourceforge.net.
9. Galax system. Available at http://db.bell-labs.com/galax/.
10. IPSI-XQ - XQuery demonstrator. Available at http://www.ipsi.fraunhofer.de
11. Pathan, Available at http://software.decisionsoft.com
12. A. Marian and J. Siméon. *Projecting XML Documents*, VLDB'2003. Berlin, Germany, September 2003
13. W. Meier. eXist: An Open Source Native XML Database. Web and Database-Related Workshops, Erfurt, Germany, October 2002.
14. Apache Xindice. *Available at: http://xml.apache.org/xindice/*
15. H. Jagadish, S. Al-Khalifa, A. Chapman, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, N. Wiwatwattanan, Y. Wu and C. Yu. TIMBER: A native XML database, The VLDB Journal, 11(4): 274-291, 2002.
16. J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Siméon. *Statix: Making XML count.* In Proc. of SIGMOD, pages 181-191, 2002.
17. XMark - The XML Benchmark project. www.xml-benchmark.org
18. R. Goldman and J. Widom. *DataGuides: Enabling Query Formulation and Optimization in Semi-structured Databases*, the 23rd Int. Conf. on Very Large Databases (VLDB), Athens, Greece, pages 436-445, 1997.

19. T. Milo and D. Suciu. *Index structure for path expression*, In Proceedings of 7th International Conference on Database Theory, pages 277-295, 1999.

20. Q. Li and B. Moon.*Indexing and Querying XML Data for Regular Path Expressions*, Proceedings of 27th International Conference on Very Large Database (VLDB'2001), Rome, Italy, pages 361-370, September 2001.

21. S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, J. Siméon, *An XML Query Language (XQuery) 1.0, W3C Working Draft.* Available at http://www.w3.org/TR/xquery/, Nov 2003.

22. B. McWhirter and J. Strachnan. *Jaxen: Universal XPath Engine.* Available at http://jaxen.org/.

23. C.C. Kanne and G. Moerkotte, *Efficient storage of XML data.* In Proc. ICDE Conf., poster abstract, page 198, San Diego, USA, 2000.

24. P. Bohannon and J. Freire and P. Roy and J. Simeon, *From (XML) Schema to Relations: A Cost-based Approach to XML Storage*, ICDE, 2002

25. T. Worthington and E. Rundenstiner, *XPath Processor*, WPI MQP Report, April 2002.

26. Flavio Rizzolo, Alberto Mendelzon. *Indexing XML Data with ToXin*, WebDB, pages 49-54, Santa Barbara, USA, 2001.

27. Z. Chen, H. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. Ng and D. Srivastava, *Counting Twig Matches in a Tree*, ICDE, pages 595-604, 2001.

28. Document Object Model (DOM), Available at http://www.w3.org/DOM/.