Algorithms for XML Stream Processing: Massive Data, External Memory and Scalable Performance

Muath Alrammal

UNIVERSITÉ PARIS-EST

ÉCOLE DOCTORALE

MSTIC: Mathématiques et Sciences et Technologies de l'Information et de la Communication

Thèse de doctorat

Informatique

Muath ALRAMMAL

Algorithms for XML Stream Processing: Massive Data, External Memory and Scalable Performance.

Thèse dirigée par: Professeur Gaétan HAINS

Soutenance le 16 mai, 2011

Composition du jury:

Rapporteurs:			
	Rada CHIRKOVA	North Carolina State University	USA
	Véronique BENZAKEN	Université Paris-Sud 11	France
Examinateurs:			
	Mohamed ZERGAOUI	Innovimax SARL	France
	Mostafa BAMHA	Université d'Orléans	France
Directeur:			
	Gaétan HAINS	Université Paris-Est, Créteil	France

Acknowledgments

I would like to express my gratitude, appreciation and sincere thanks to my supervisor Pr. Gaétan HAINS for his excellent guidance, helpful and useful discussions, and continuous encouragement which made this work possible. He always helped me with pleasure in the problems that I faced during this work.

I also express my thanks to the CIO of the Innovimax company, Mr. Mohamed ZERGAOUI. He provided valuable XML expertise and technical scrutiny at many points of our project.

I am deeply indebted to all members of my research laboratory LACL for their support. I am especially grateful to Régine LALEAU, Frédéric GAVA, Flore TSILA, Tristan CROLARD, Frank POMMEREAU, Serghei VERLAN, Marie Duflot for their support both in terms of resources and encouragement.

Deep thanks to my fellow doctoral students from Palestine, Jordan, Tunisia, Morocco, Algeria, France, Italy, China, India, Greece. I have always benefited from them through discussion both technically and socially.

I cannot forget the constant encouragement and support of my whole family including my sisters.

Last but certainly not the least, I am proud to acknowledge the generous and enduring support of my wife who supported me through all the tough times.
I dedicate this work to the spirits of my beloved parents and my father in law whose constant support and prayers were gospel of encouragement for me to keep struggling for ambitions.

Résumé

Plusieurs applications modernes nécessitent un traitement de flux massifs de données XML, et cela crée des défis techniques. Parmi ces derniers, il y a la conception et la mise en ouvre d'outils pour optimiser le traitement des requêtes XPath. Il s'agit alors de fournir une estimation précise des coûts de ces requêtes traitées sur un flux massif de données XML.

Dans cette thèse, nous proposons un nouveau modèle de prévision de performance qui estime à priori le coût (en terme d'espace utilisé et de temps écoulé) pour les requêtes structurelles du fragment de langage Forward XPath.
Ce faisant, nous réalisons une étude expérimentale pour confirmer la relation linéaire entre le traitement de flux et les ressources d'accès aux données. Ce qui nous permet de construire un modèle mathématique (utilisant des régressions linéaires) pour prévoir le coût d'une requête XPath.
En outre, nous présentons une technique nouvelle d'estimation de la *sélectivité*. Elle constituée de deux éléments. Le premier est le résumé *path tree* ou arbre des chemins: une présentation concise et précise de la structure d'un document XML. Le second est l'algorithme d'estimation de sélectivité: un algorithme efficace de flux pour traverser l'arbre des chemins afin d'estimer les valeurs des paramètres de coût. Ces paramètres sont utilisés par le modèle mathématique pour déterminer le coût d'une requête XPath.
Nous comparons les performances de notre modèle avec les approches existantes.
De plus, nous présentons un cas d'utilisation de celui-ci dans un système en ligne de traitement en flux des requêtes. Le système utilise notre modèle de prévision de performance pour estimer le coût (en terme de temps / mémoire) d'une requête XPath. En outre, il fournit une estimation précise à l'auteur de la requête relativement au coût et au volume de sa requête. Ce cas d'utilisation illustre les avantages pratiques de la gestion de performance avec nos techniques.

Mots clés: Traitement de flux, données XML, requêtes XPath, estimation de sélectivité, Modèle de performance, optimisation de requêtes.

Summary

Many modern applications require processing of massive streams of XML data, creating difficult technical challenges. Among these, there is the design and implementation of applications to optimize the processing of XPath queries and to provide an accurate cost estimation for these queries processed on a massive steam of XML data.
In this thesis, we propose a novel performance prediction model which a priori estimates the cost (in terms of space used and time spent) for any structural query belonging to Forward XPath.
In doing so, we perform an experimental study to confirm the linear relationship between stream-processing and data-access resources. Therefore, we introduce a mathematical model (linear regression functions) to predict the cost for a given XPath query. Moreover, we introduce a new selectivity estimation technique. It consists of two elements. The first one is the path tree structure synopsis: a concise, accurate, and convenient summary of the structure of an XML document. The second one is the selectivity estimation algorithm: an efficient stream-querying algorithm to traverse the path tree synopsis for estimating the values of cost-parameters. Those parameters are used by the mathematical model to determine the cost of a given XPath query.
We compare the performance of our model with existing approaches.
Furthermore, we present a use case for an online stream-querying system. The system uses our performance predicate model to estimate the cost for a given XPath query in terms of time/memory. Moreover, it provides an accurate answer for the query's sender. This use case illustrates the practical advantages of performance management with our techniques.

Keywords: Streaming processing, XML data, XPath queries, query optimization, selectivity estimation, performance prediction model.

Contents

CHAPTER 1

Introduction

Contents

1.1 Introduction

Extensible markup language (XML) [Bray 2008] is a simple, very flexible text format derived from SGML, the standard generalized markup language (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.

The most salient difference between HTML [Hickson 2011] and XML is that HTML describes presentation and XML describes content. An HTML document rendered in a web browser is human readable. XML is aimed toward being both human and machine readable.

XML has gone from the latest buzzword to an entrenched e-business technology in record time. XML is currently being heavily pushed by the industry and community as the *lingua franca* for data representation and exchange on the Internet. The popularity of XML has created several important applications like information dissemination, processing of the scientific data, and real time news. Query languages like XPath [Berglund 2010] and XQuery [Boag 2010] have been proposed for accessing XML data. They provide a syntax for specifying which elements and attributes are sought to retrieve specific pieces of a document.

Despite a logically clean structure, the computational complexity of XPath, or XQuery queries can vary dramatically [TenCate 2009] [Gottlob 2005] and the unconstrained use of XPath leads to unpredictable space and time costs.

Furthermore, often, data sets are too large to fit into limited internal memory and/or need to be processed in real time during a single forward sequential scan. In addition, sometimes query results should be output as soon as they are found.

One proposed approach to combine the simplicity of XML data, the declarative nature of XPath queries and reasonable performance on large data sets is to impose their processing by purely streaming algorithms. The result is that queries must be restricted to a fragment of XPath but on the other hand processing space can be limited and very large documents can be accessed efficiently. This is the approach we describe and investigate in this thesis.

A stream of XML data is the depth-first, left-to-right traversal of an XML document [Bray 2008]. In the streaming model queries must be known before any data arrives, so queries can be preprocessed by building machines for query evaluation. Figure 1.1 illustrates a data stream processor.

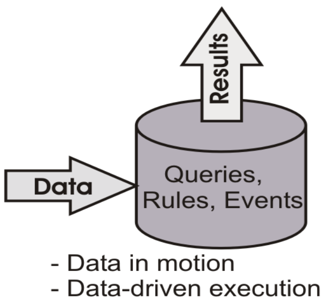


Figure 1.1: Data Stream Processor.

Query evaluation of a stream of XML data raises many challenges compared to a non streaming environment: the recursive nature of XML document, the single sequential forward scan of a stream of XML data, also the presence of descendant axes and predicates in the XPath query. An explanation for some of these challenges follows:

• During the evaluation process of XPath queries which include predicates, we may encounter potential answers nodes (solutions) before we reach the required data and evaluate the predicates to decide their satisfaction. Based on that, we need to record information about the potential answer nodes, as well as, their associated pattern matches to the query until the relevant data is encountered, so we can determine the predicate satisfaction. In the worst case, the size of the buffer (the size of the potentials answers nodes) reaches the document's size.

• The descendant axis traversal in a query and the recursive structure of the XML document may cause an exponential number of pattern matches of sub-queries from a single initial node.

The author of a query may have no immediate idea of what to expect in memory consumption and delay before collecting all the resulting sub-documents. This unpredictability can diminish the practical use of XPath stream-processing.

To alleviate this situation we need an accurate performance prediction (cost) model for stream-processing of XPath queries.

The remainder of the chapter is structured as follows: section 1.1.1 presents some terminology used in the chapters of this thesis. Section 1.2 summarizes some of the challenges which should be considered by any performance prediction model developed for query optimization in streaming mode. Section 1.3 describes the main contributions of this thesis. Finally, section 1.4 illustrates the organisation of the thesis and explains the relations and dependencies between the chapters of this thesis.

1.1.1 Preliminaries

In this section, we present and define some terminology used in this thesis.

1.1.1.1 Data Model of XML Document

An XML document is modeled as a rooted, ordered, labelled tree, where each node corresponds to an element, attribute or a value, and the edges represent (direct) element-subelement or element-value relationships. An XML document, when passed through a SAX [Brownell 2002] parser, will generate a sequence of events. A streaming algorithm processes the SAX events, which are: $startElement(X,l)$ and $endElement(X)$. They are produced respectively when the opening or closing tag of a element is encountered and accept the name of the element X as input parameter. When a text value is encountered, the event $Text(value)$ is activated. The list l for $StartElement(X,l)$ represents the list of attributes for the element name X.

Our stream-querying algorithm presented in chapter 4 processes attributes. While our selectivity estimation algorithm presented in the same chapter does not treat explicitly the attribute $'@'$, since it can be handled in a way similar to the child.

1.1.1.2 XPath

XPath [Berglund 2010] is a language that describes how to locate specific elements (and attributes, processing instructions, etc.) in a document. It operates on the abstract, logical structure of an XML document, rather than its surface syntax. This logical structure is known as the data model (that we defined in 1.1.1.1). XPath has a particular importance for XML applications since it is a core component of many XML processing standards such as XSLT [Kay 2007] or XQuery [Boag 2010].
XPath can be classified based on its fragment as follows:

- XPath 2.0: it is the largest fragment of XPath, for precise information about its grammar see [Berglund 2010].
- XPath 1.0: it is a sub fragment of XPath 2.0, for precise information about its grammar see [Clark 1999].

In this thesis, we define Forward XPath as below:

- Forward XPath: *a sub fragment of XPath 1.0 consisting of queries that have: child, descendant axis. NodeTest which is either element, wildchard, $'text()'$. Predicate with ($'or'$, $'not'$, $'and'$), and arithmetic operations.*

For a precise understanding of Forward XPath, we illustrate its grammar in figure 1.2. A location path is a structural pattern composed of sub expressions called steps. Each step consists of an *axis* (defines the tree-relationship between the selected nodes and the current node), a *node-test* (identifies a node within an axis), and *zero or more predicates* (to further refine the selected node-set). An absolute location path starts with a $'/'$ or $'//'$ and a relative location path starts with a $'./'$ or $'.//'$.

Our restriction to the downward axes in our XPath fragment is not absolute, we could cover more general axes than $'/'$, $'//'$ by using rewrite rules as shown in [Olteanu 2002] to reduce more general axis operations to forward ones when possible.

A *Simple path* is any query which belongs to the fragment Forward XPath, but does not contain predicates. A *Twig path* is any query which belongs to the fragment Forward XPath and which contains predicates

```
Path := GenericPath | AttributeStep
GenericPath := GenericStep | GenericStep GenericPath | GenericStep1
GenericStep := Axis NodeTest | Axis NodeTest '[' Predicate ']'
AttributeStep := '@' NodeTest  | '@' NodeTest '[' Predicate ']'
GenericStep1 := Axis 'text()'
Axis := '/' | '//'
NodeTest := name | '*'
Predicate := PredicatePath | PredicatePath CompOp constant
                           | Predicate 'and' Predicate
                           | Predicate 'or' Predicate
                           | 'not(' Predicate ')'
CompOp := '=' | '!=' | '>' | '>=' | '<' | '<='
PredicatePath := '.' GenericPath | AttributeStep
```

Figure 1.2: Grammar of Forward XPath

1.1.1.3 Recursion in XML Document

So-called *recursion* occurs frequently in XML data [Choi 2002]: some elements with the same node-labels are nested on the same path in the data tree. In [Bar-Yossef 2004], the authors define the recursion depth of an XML data tree D with respect to the query node q in Q, denoted by r_q as: the length of the longest sequence of nodes $e_1, \ldots, e_r$ in D, such that 1) all the nodes lie on the same path (root-to-leaf), and 2) all the nodes match structurally the sub-pattern q. To facilitate and clarify the meaning of recursion, figure 1.3 illustrates the recursion depth of document D with respect to the query Q: $//A//B[.//C]/K$.

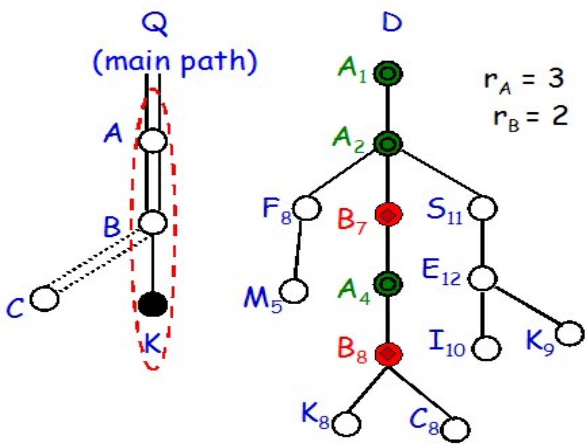


Figure 1.3: Recursion depth of D w.r.t Q.

The single line edges represents child $('/')$, the double line edges represents descendant$('//')$, single dashed line represents $[./node()]$, double dashed line rep-

Matching Nbr.	Nodes of Structural Matching			
1	A_1	B_7	C_8	K_8
2	A_2	B_7	C_8	K_8
3	A_4	B_8	C_8	K_8

Table 1.1: Nodes of Structural Matching of Q in D

resents $[.//node()]$ and the result node which is in this example the shaded node K. It is obvious from figure 1.3 that node C is not on the main path, this is why we do not consider it as a r_C. If we have a look at table 1.1, both nodes A and B are applied to the definition of r_q. Actually, $r_A = 3$ is represented by (A_1, A_2, A_4), while $r_B = 2$ is represented by (B_7, B_8).

1.1.1.4 Document Depth

Let d_D be the length of the longest root to leaf path in the tree. In our example in figure 1.3, document depth is the length of the path from root node A_1 to the leaf node K_8.

1.1.1.5 Stream-querying Process

The process of stream-querying consists in outputting all nodes in an XML data set D (answer nodes) that satisfy a specified XPath query Q at its result node.

1.1.1.6 Stream-filtering Process

The process of stream-querying consists in determining whether there exists at least one match of a query Q in an XML data set D.

1.1.1.7 Synopsis

A synopsis data structure is a succinct description of the original data set with low computational overhead and high accuracy for processing tasks like selectivity estimation and query answer approximation.

1.1.1.8 Selectivity Estimation Technique

Selectivity estimation is an estimate of the number of matches for a query Q evaluated on an XML document D. It is desirable in interactive and internet applications. With it, the system could warn the end user that his/her query is so coarse that the amount of results will be overwhelming.

However, this selectivity does not measure the size of these matches. Furthermore, it measures neither the total amount of memory allocated by the program to find these matches (space used) nor the processor time used by the program to find the matches (time spent). As a result, selectivity estimation appears necessary but incomplete as a technique for managing queries on large documents accessed as streams and it is not sufficient to model the query cost.

1.1.1.9 Performance Prediction Model

The performance prediction model is a mathematical model which estimates the cost (in terms of space used and time spent) of an XPath query before actually executing it. A precise performance prediction model requires an efficient selectivity estimation technique, but selectivity alone is not sufficient to model the cost for a given query as we explain in more details in chapter 5.

1.2 Challenges

Developing performance prediction models for query optimization is significantly harder for XML queries than for traditional relational queries. The reason is that XPath query operators are more complex than relational operators such as table scans and joins. Moreover, the query evaluation process of a stream of XML data raises extra challenges compared to non-streaming environments: the recursive nature of XML documents, the single sequential forward scan of a stream of XML data, also the presence of descendant axes and the predicates in the XPath query.

The basic idea is to identify the parameters that determine the cost for a given query on an XML document, such as:

1. *NumberOfMatches*: is the number of answer elements found during processing of the query Q on the XML document D.

2. *Cache*: is the number of elements cached in the run-time stacks during processing of the query Q on the XML document D. They correspond to the axis nodes of Q.

3. *Buffer*: is the number of potential answer elements buffered during processing of the query Q on the XML document D.

4. *OutputSize*: is the total size in MiB of the number of answer elements found during processing of the query Q on the XML document D.

5. *WorkingSpace*: is the total size in MiB for the number of elements cached in the run-time stacks and the number of potential answer elements buffered during processing of the query Q on the XML document D.

6. *NumberOfPredEvaluation*: is the number of times the query's predicates are evaluated (their values are changed or passed from an element to another).

The above mentioned parameters are interrelated (we can not ignore any of them) and necessary to estimate the cost for a given twig. Below, we explain the need for this set of parameters:

the value of *NumberOfMatches* is insufficient for estimating the cost for a given query Q evaluated on an XML document D. For example: the cost for a given query Q_1 with *NumberOfMatches* =5 might be higher than the cost for a given query Q_2 with *NumberOfMatches* =7, because the value of *OutputSize* for Q_1 is larger than the same value for Q_2. Therefore, we need to know the size of the answers (*i.e. OutputSize*. Increasing the value of *OutputSize* increases the cost of Q.

However, the values of *NumberOfMatches* and *OutputSize* are still insufficient to determine the cost of Q precisely. Because during the processing of Q, we may need to buffer some potential answers elements (*i.e. Buffer*) and to cache the intermediate answers (*i.e. Cache*). The values of *Cache* and *Buffer* affect the cost of Q. Increasing their values increases the cost of Q, because it increases the time needed for buffering and caching. But still they are insufficient to determine the cost of Q precisely. For example: the cost for a given query Q_1 with *Buffer* =5 and *Cache*) = 8 might be higher than the cost for a given query Q_2 with *Buffer* =7 and *Cache*) =10, because the value of *WorkingSpace* for Q_1 is large than the value of *WorkingSpace* for Q_2.

During the processing of Q, there is another parameter which affects it cost that is *NumberOfPredEvaluation*. Increasing the value of *NumberOfPredEvaluation* increases the cost of Q, because it increases the evaluation time of Q.

The performance prediction model should estimate accurately these parameters in order to estimate the cost of Q.

Below, we summarize some of the challenges which should be considered by any performance prediction model developed for query optimization in streaming mode.

1.2.1 The Expressiveness of XPath

The existence of the descendant axis $'//'$, the wildcard node $('*')$, predicates and the same node-labels in the XPath query evaluated on a deep and recursive XML document increase the buffering, caching sizes and the processing time enormously. This is because the stream-querying algorithm will be forced to buffer and cache a large number of nodes. An example of a complex XPath query is $//A[./B//*]//*[./A]/K$ (see figure 1.4).

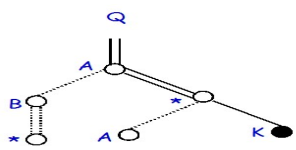


Figure 1.4: Complex XPath Query.

The standard query language for XML [Bray 2008] namely XPath

	XMark	Book	TreeBank
Structure	Wide and Shallow	Semi deep+ and recursive	narrow deep and recursive
Data Size	113MiB	12MiB	82MiB
Max./Avg Depth	12/5.5	22/19.4	36/7.6

Table 1.2: Different Data set Structures.

[Berglund 2010] is a very rich and expressive language, therefore, the performance prediction model should be strongly capable to model the cost for simple and complex queries. In this thesis, we are interested in modeling and estimating the cost for any structural query which belongs to the fragment of Forward XPath.

1.2.2 Structure of XML Data Set

Data sets may have varied structures, for instance, shallow XML data sets (wide) that do not include recursive elements (figure 1.5). In this case the caching space costs of the stream-querying algorithm might be negligible. An example for this type of XML data set is XMark [Schmidt 2001] which is a well known benchmark data set that allows users and developers to gain insights into the characteristics of their XML repositories. Table 5.11 indicates that the depth of this data set reaches 12 (not deep), and has a size of 113MiB[1].

Other data sets are semi-deep and recursive e.g. the Book data set [Diaz 1999], actually a synthetic data set generated using IBM's XML generator, based on a real DTD from an W3C XQuery use case. As we can see from table 5.11, it has a size of 12 MiB which is not enormous and a maximum depth that reaches 22 which is quite deep compared to its size. It includes only one recursive element named section. In fact, different sections node can be nested on the same path in the data set, therefore this kind of data set (semi-deep and recursive) increases the buffering space, caching space and processing time.

We can find also data sets with a narrow deep structure, e.g. the TreeBank data set [Suciu 1992]. Here one can recognize the structure of the data set from it maximum depth in table 5.11 that is 36, moreover, its average depth is 7.8. The existence of these properties in the data set is strongly related to the algorithmic complexity of stream-processing.

The performance prediction model should be able to model the cost for a given query on any data set (structure/size).

[1]The mebibyte (MiB) is a multiple of the unit byte for digital information. One mebibyte (MiB) is 2^{20} (*i.e.,* 1024 x 1024) bytes [ICE 2007]

1.2.3 Query Evaluation Strategy

The strategy used to evaluate the XPath query may affect the size of the buffering space B and the processing time. B might reach document size $|D|$ in the worst case. For example, let us consider that we have the document D and the query Q: $//A[./F]/C$ as it is shown in figure 1.5. In the so-called lazy approach, $B = n$ or in other words $B = |D|$ since the predicate of A is not evaluated until $< /A_i >$ arrives. In this case all nodes starting from C_1 to C_n have to be buffered, which will increase the buffering size remarkably. In the so-called eager approach $B = 0$ because the predicates of A is evaluated to be true the moment element $< F >$ arrives. Thus, each $< C_i >$ can be flushed as a query result the moment it arrives and does not need to be buffered at all. Obviously this will improve the buffering space performance.

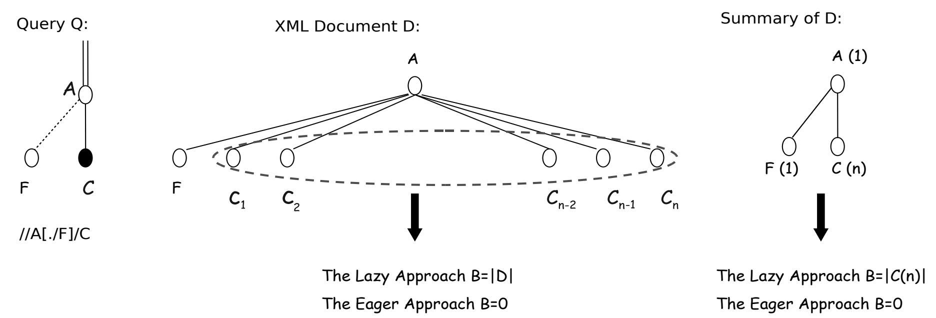


Figure 1.5: Lazy and Eager Approaches.

Note that figure 1.5 is an example of a wide data set.

The performance prediction model should be time and space efficient no matter what query evaluation technique is used. In the example of figure 1.5, a possible solution to reduce the buffering space of the lazy approach is to evaluate Q on the summary of D. The number in the bracket to the right of the each element (in the summary of D) represents its frequency. Therefore, by applying the the lazy approach we buffer only one element C with it frequency (n) and we get the number of matches that is the value of n.

1.2.4 Evolution and Data Set Updating

When the underlying XML document (data set) D is updated, *i.e.* some elements are added or deleted, the performance prediction model should be able to model precisely the change in the cost for a given query Q on D (without the need to rebuild its "tables").

All the above challenges to performance prediction will be addressed in later chapters.

1.3 Contributions

We first present two surveys on selectivity estimation techniques for XML queries and on stream-processing for XML data. Following this, the technical developments are described, with main contributions as follows:

- We study in detail the path tree, a synopsis structure for XML documents that is used for accurate selectivity estimates. We formally define it and we introduce two algorithms to construct it. To the best of our knowledge, the path tree was not formally defined in the literature, but was used before in more limited ways.

- We extend and optimize the lazy stream-querying algorithm LQ which was introduced by [Gou 2007]. The current version of the algorithm processes any query belonging to the fragment of Forward XPath (that is explained in section 1.1.1.2).

- We present a new selectivity estimation algorithm which was inspired from our extended stream-querying algorithm LQ. Our estimation algorithm is efficient for traversing the path tree structure synopsis to calculate the estimates. The algorithm is well suited to be embedded in a cost-based optimizer.

- We present a study we performed to confirm the linear relationship between the stream-processing and the data-access. As we will see later (section 5.3 and section 5.4), this linear relationship has an important role in our performance prediction models.

- We present the performance prediction model - simple path, an accurate model for stream-processing of simple path queries. Our model collects static information about the XML document and predicts *a priori* the memory consumption of a query to within a few percent. This allows a user to either modify the query if predicted consumption is too high, or to allow the algorithm to execute normally. The model is portable: its prediction is also correct to within a small error on a different machine. Moreover, the error rate is stable from small documents to documents on the order of 1GiB and nothing prevents application to much larger documents.

- We present the performance prediction model - twig path, an accurate model for stream-processing of any structural query which belongs to the fragment of Forward XPath. The model is able to estimate the cost for a given query in terms of time spent /memory used.

- We present a use case called an on-line stream-querying system. The system of the use case uses the performance predicate model - twig path to estimate the cost for a given query in term of time/memory. Moreover, it provides an accurate answer for the query's sender.

1.4 Thesis Organisation

This thesis is organized as follows: after the brief introduction in this chapter, chapter 2 reviews the state of the art and pinpoints two critical areas where we develop our research work: (1) selectivity estimation techniques for XML queries, where we justify the need of a new selectivity estimation technique that is based on the streaming technique. (2) stream-processing for XML data, where we explain the raison of choosing the lazy stream-querying algorithm LQ in our work.

Chapter 3 defines the path tree structure synopsis that is used for accurate selectivity estimates. The path tree is used by our selectivity estimation technique that is introduced in chapter 4. Moreover, this chapter introduces two techniques to built the path tree, they are: (1) the automata-based algorithm and (2) the streaming -based algorithm with examples on the incremental updates for the path tree.

Chapter 4 presents our selectivity estimation technique. This chapter starts by explaining the extension process for the stream-querying algorithm which was introduced by [Gou 2007]. After that, it presents the selectivity estimation algorithm which was inspired from our extend stream-querying algorithm LQ. Our estimation algorithm is efficient for traversing the path tree structure synopsis to calculate the estimates. The algorithm is well suited to be embedded in a cost-based optimizer. The path tree and the selectivity estimation technique represent our selectivity estimation algorithm. Then, the chapter continues by presenting several examples on the selectivity estimation for path/twig expressions. Finally, it ends by showing the accuracy of our selectivity estimation technique.

Chapter 5 presents the performance prediction model. It starts by a preliminary study we performed to confirm the linear relationship between stream-processing and data-access. Then, it proceeds by explaining the main idea and the general structure of the performance prediction model. After that, it presents the "performance prediction model - simple path" where extensive experiments were performed to show the accuracy of the estimation and the portability of the model. The chapter continues by introducing the "performance prediction model - twig path" which estimates the cost for any given query belong to the fragment of Forward XPath. The model uses our selectivity estimation technique (the path tree synopsis plus the selectivity estimation algorithm) that is presented in chapter 4 to estimate the selectivity for a given query. Extensive experiments were performed. We considered the accuracy of the estimations, the types of queries and data sets that this synopsis can cover, the cost of the synopsis to be created and the estimated vs measured time/memory. The chapter ends by introducing an online stream-querying system (a use case). The system uses the "performance predicate model - twig path" to estimate the cost for a given query in term of time/memory. Moreover, it provides an accurate answer for the query's sender.

Chapter 6 concludes our work and presents our perspectives for future research.

1.4.1 The Dependency of Thesis's Chapters

Figure 1.6 illustrates the dependencies between thesis's chapters.

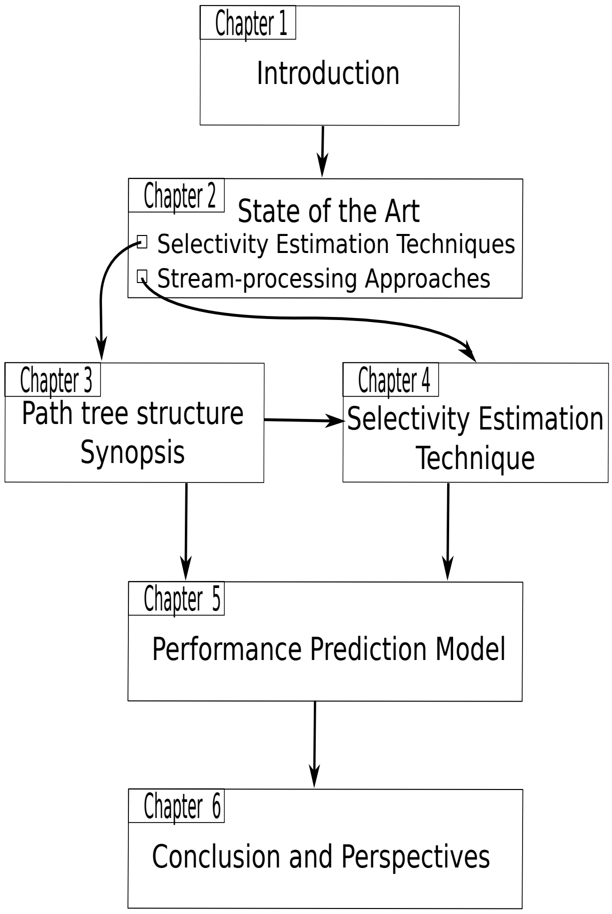


Figure 1.6: The Dependency of Thesis's Chapters

CHAPTER 2

State of the Art

Contents

2.1 Introduction

This chapter surveys the existing work on stream processing for XML data and on selectivity estimation techniques for XPath queries. We start by presenting the important properties needed for the selectivity estimation techniques before we explain them. After that, we categorize the stream processing approaches, then we explain several techniques or algorithms for each one.

2.2 Selectivity Estimation

In this section, we start by introducing some of the important properties of selectivity estimation techniques. After that, we give an overview of the literature related to this domain, as of 1Q 2011.

2.2.1 Properties of Selectivity Estimation Techniques

The design and the choice of a particular selectivity estimation technique depends on the problem being solved with it. Therefore, the technique needs to be constructed in a way related to the needs of the particular problem being solved [Aggarwal 2007].

In general, we would like to construct the synopsis structure in such a way that it has wide applicability across broad classes of problems. In our work, the applicability to streams of XML data makes the space and time efficiency issue of construction critical.

When looking for an efficient, capable (general enough) and accurate selectivity estimation technique for XPath queries, there are several issues that need to be addressed. Some of these issues can be summarized as follows:

- It must be practical: in general, one of the main usages of the selectivity estimation techniques is to accelerate the performance of the query evaluation process. Thus, while theoretical guarantees are important for any proposed approach, practical considerations are much more important. The performance characteristics of the selectivity estimation process are a crucial aspect of any approach. The selectivity estimation process of any query or sub-query must be much faster than the real evaluation process. In other words, the cost savings on the query evaluation process using the selectivity information must be higher than the cost of performing the selectivity estimation process. In addition, the required summary structure(s) for achieving the selectivity estimation process must be efficient in terms of memory consumption.

- It should support structural and data value queries: in principal, all XML query languages can involve structural conditions in addition to the value-based conditions. Therefore, any complete selectivity estimation system for the XML queries requires maintaining statistical summary information about both the structure and the data values of the underlying XML documents. A recommended way of doing this is to apply the XMill approach [Liefke 2000] in separating the structural part of the XML document from the data part and then group the related data values according to their path and data types into homogeneous sets. A suitable summary structure for each set can then be easily selected. For example, the most common approaches in summarizing the numerical data values is by using histograms or wavelets while several tree synopses could be used to summarize the structural part.

- One pass constraint: for streaming applications or techniques, the streams of XML data typically contain a large number of points, the contents of the stream cannot be examined more than once during the course of computation.

Therefore, all summary structure/data values construction algorithms should be designed under a one pass constraint.

- It should be strongly capable: the standard query languages for XML [Bray 2008] namely XPath [Berglund 2010] and XQuery [Boag 2010] are very rich languages. They provide rich sets of functions and features. These features include structure and content-based search, join, and aggregation operations. Thus, a good selectivity estimation approach should be able to provide accurate estimates for a wide range of these features. In addition, it should maintain a set of special summary information about the underlying source XML documents. For example: a universal assumption about a uniform distribution of the elements structure and the data values may lead to many potential estimation errors because of the irregular nature of many XML documents.

- It must be accurate: providing an accurate estimation for the query optimizer can effectively accelerate the evaluation process of any query. However, on the other hand, providing the query optimizer with incorrect selectivity information will lead it to incorrect decisions and consequently to inefficient execution plans.

- It must evolve and be incremental: when the underlying XML document is updated, *i.e.* some elements are added or deleted, the selectivity estimation technique should be updated (without the need of rebuilding it) as well to provide an accurate selectivity estimation for a given query.

- It should be independent: it is recommended that the selectivity estimation process be independent of the actual evaluation process which facilitates its use with different query engines that apply different evaluation mechanisms. This property is an advantage for software engineering of the corresponding module(s).

- Time and Space Efficiency: In many traditional synopsis methods on static data sets (such as histograms), the underlying dynamic programming methodologies require super-linear space and time. This is not acceptable for a data stream [Aggarwal 2007]. For the case of space efficiency, it is not desirable to have a complexity which is more than linear in the size of the stream.

2.2.2 Path/Twig Selectivity Estimation Techniques

In this section, we give an overview of the literature related to the selectivity estimation approaches in the XML domain. Estimation techniques can be classified in terms of the structure used for collecting the summary information into two main classes:

1. Synopsis-based estimation techniques: this class of the estimation techniques uses tree or graph structures for representing the summary information of the source XML documents.

2. Histogram-based estimation techniques: this class of the estimation techniques uses the statistical histograms for capturing the summary information of the source XML documents.

2.2.2.1 Synopsis-Based Estimation Techniques

[Aboulnaga 2001] have presented two different techniques for capturing the structure of the XML documents and for providing accurate cardinality estimations for the path expressions. The presented techniques only support the cardinality estimations of simple path expressions without predicates and so-called *recursive* axes (repeated node-labels in the expression). Moreover, the models cannot be applied to twigs.

The first technique presented in this paper is a summarizing tree structure called a path tree. A path tree is a tree containing each distinct rooted path in the database (or data set) where the nodes are labeled by the tag name of the nodes.

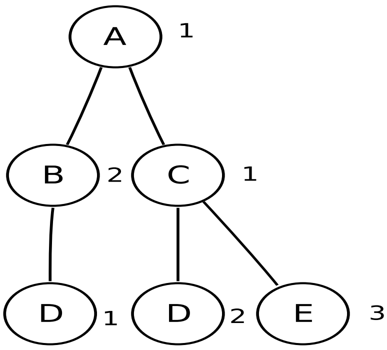
To estimate the selectivity of a given path expression p in the form of $s1/s2/.../sn$, the path tree is scanned by looking for all nodes with tags that match the first tag of the path expression. From every such node, downward navigation is done over the tree following child pointers and matching tags in the path expression with tags in the path tree. This will lead to a set of path tree nodes which all correspond to the query path expression. The selectivity of the query path expression is the total frequency of these nodes.

The problem is the size of the path tree constructed form a large XML document, it is larger than the available memory size for processing. To solve this problem, the authors described different summarization techniques based on the deletion of low frequency nodes, and on their replacement by means of $* - nodes$ $(star\ nodes)$. Each $* - node$, denoted by a special tag name "$*$", denotes a set of deleted nodes, and inherits their structural properties as well as their frequencies. Unfortunately, the path tree is not formally defined in this work, and to the best of our knowledge, it is not defined in the literature.

The second technique presented in this paper is a statistical structure called Markov table (MT). This table, implemented as an ordinary hash table, contains any distinct path of a length up to m and its selectivity. Thus, the frequency of a path of length n can be directly retrieved from the table if $n \leq m$, or it can be computed by using a formula that correlates the frequency of a tag to the frequencies of its $m-1$ predecessors if $n > m$. Since the size of a Markov table may exceed the total amount of available main memory, the authors present different summarization techniques which work as in the case of a path tree and delete low frequency paths and replace them with $* - paths$.

```
<A>
    <B> </B>
    <B>
        <D> </D>
    </B>
    <C>
        <D> </D>
        <E> </E>
        <D> </D>
        <E> </E>
        <E> </E>
    </C>
</A>
```

(a) XML document D



(b) path tree of D

Path	Freq.
A	1
B	2
C	1
D	3
E	3
AB	2
AC	1
BD	1
CD	2
CE	3

(c) MT of D

Figure 2.1: An XML document D and its both path tree and markov table

Figure 2.1(a) illustrates an example of XML document D and the representation of its both corresponding path tree 2.1(b) and Markov table 2.1(c).

XPATHLEARNER [Lim 2002] is an on-line learning method for estimating the selectivity of XML path expressions by means of statistical summaries used to build a Markov histogram on path selectivities gathered from the target XML engine. It employs the same summarization and estimation techniques as presented in [Aboulnaga 2001].

The novelty of XPATHLEARNER is represented by the fact that it collects the required statistics from the query feedback in an on-line manner, without accessing and scanning the original XML data, which is in general resource-consuming.

These statistics are used to learn both tag and value distributions of input queries, and, when needed, to change the actual configuration of the underlying Markov histogram in order to improve the accuracy of approximate answers. From this point of view, XPATHLEARNER can be intended as a workload-aware method.

An important difference between XPATHLEARNER and the MT of [Aboulnaga 2001] is that the XPATHLEARNER supports the handling of predicates (to further refine the selected node-set) by storing statistical information for each distinct tag-value pair in the source XML document.

[Zhang 2005] have presented a similar technique called Comet (Cost Modeling

Evolution by Training) for cost modeling of complex XML operators. It exploits a set of system catalogue statistics that summarizes the XML data, the set of simple path statistics and a statistical learning technique called transform regression instead of detailed analytical models to estimate the selectivity of path expressions. The technique used to store the statistics is the path tree of [Aboulnaga 2001].
This work is more oriented toward XML repositories consisting of a large corpus of relatively small XML documents. Their initial focus is only on the CPU cost model. To do that, they developed a CPU cost model for XNAV operator which is an adaptation of TurboXPATH [Josifovski 2005]. Their idea was taking from previous works in which statistical learning method are used to develop cost models of complex user-defined functions [He 2004] and [Lee 2004].
The system can automatically adapt to changes over time in the query workload and in the system environment. The optimizer estimates the cost of each operator in the query plan (navigation operator, join operator) and then combines their costs using an appropriate formula.
The statistical model can be updated either at a periodic intervals or when the cost-estimation error exceeds a specified threshold. Updating a statistical model involves either re-computing the model from scratch or using an incremental update method.

The authors of [Chen 2001] proposed a correlated sub-path tree (CST), which is a pruned suffix tree (PST) with set hashing signatures that helps determine the correlation between branching paths when estimating the selectivity of twig queries. The CST method is off-line, handles twig queries, and supports substring queries on the leaf values. The CST is usually large in size and has been outperformed by [Aboulnaga 2001] for simple path expressions.

Described in [Polyzotis 2004b] the Twig-Xsketch is a complex synopsis data structure based on XSketch synopsis [Polyzotis 2002a] augmented with edge distribution information. It was shown in [Polyzotis 2004b] that Twig-Xsketch yields selectivity estimates with significantly smaller errors than correlated sub-path tree (CST). For the data set XMark [Schmidt 2001] the ratio of error for CST is 26% vs. 3% for Twig-Xsketch.
TreeSketch [Polyzotis 2004a] is found on a partitioned representation of nodes of the input graph-structured XML database. It extends the capabilities of XSketch [Polyzotis 2002a] and Twig-Xsketch [Polyzotis 2004b] . It introduces a novel concept of count-stability (C-stability) which is a refinement of the previous F-stability of [Polyzotis 2002a]. This refinement leads to a better performance in the compression of the input graph-structured XML database. TreeSketch builds its synopsis in two steps. First, it creates an intermediate count-stability (C-stability) synopsis that preserves all the information of the original XML data set in a compact format. After that, the Tree-Sketch synopsis is built on top of the C-stability synopsis by merging similar structures.

The construction time of TreeSktech for the complex data set TreeBank 86MiB (depth 36) took more than 4 days, this result was confirmed in [Luo 2009]. Moreover, the TreeSketch synopsis does support the recursion in the data set as it is explained in [Zhang 2006b].

[Zhang 2006b] have addressed the problem of deriving cardinality estimation (selectivity) of XPath expressions. In this work, the authors are mainly focusing on the handling of XPath expressions that involve only structural conditions. The main idea of their paper is to provide an efficient treatment of recursive XML documents and an accurate estimation of recursive queries. An XML document is said to be *recursive* if it contains an element directly or indirectly nested in an element with the same name. In other words, if it contains rooted paths which have multiple occurrences of the same element labels. A path expression is said to be recursive with respect to an XML document if an element in the document could be matched to more than one NodeTest in the expression (it contains same node-labels). Therefore, in order to derive an efficient and accurate estimation for a recursive path expression, the authors introduce a new notion named as recursion levels. Given a rooted path in the XML tree, they define the path recursion level (PRL) by the maximum number of occurrences of any label minus 1. The recursion level of a node in the XML tree is defined by the PRL of the path from root to this node. The document recursion level (DRL) is defined to be the maximum PRL over all rooted paths in the XML tree.

The authors define a summary structure for summarizing the source XML documents into a compact graph structure called XSeed. The XSeed structure is constructed by starting with a very small kernel which captures the basic structural information (the uniform information) as well as the recursion information of the source XML document. The kernel information is then incrementally updated through the feedback of queries.

The XSeed kernel is represented in the form of a label- split graph summary structure proposed by [Polyzotis 2002b]. In this graph, each edge $e = (u; v)$ is labeled with a vector of integer pairs $(p0 : c0, p1 : c1, ..., pn : cn)$. The ith integer pair $(pi : ci)$ indicates that at recursion level i, there are a total of pi elements mapped to the synopsis vertex u and ci elements mapped to the synopsis vertex v.

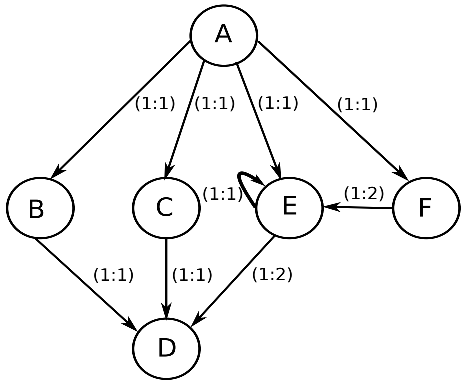
Figure 2.2 illustrates an example of XML document D and its corresponding XSeed kernel

The high compression ratio of the kernel can lead to a situation where information is lost. This loss of information results in the occurrence of significant errors in the estimation of some cases. To solve this problem, the authors introduce another layer of information, called hyper-edge table (HET), on top of the kernel. This HET captures the special cases that are not addressed by original assumptions made by the kernel (irregular information). For example, it may store the actual cardinalities of specific path expressions when there are large errors in their estimations.

Relying on the defined statistic graph structure and its supporting layer, the

```
<A>
    <B> </B>
        <D> </D>
    <C>
        <D> </D>
    </C>
    <E>
        <D> </D>
        <E> </E>
        <D> </D>
    </E>
    <F>
        <E> </E>
        <E> </E>
    </F>
</A>
```

(a) XML *D*



(b) XSeed Kernel of *D*

Figure 2.2: An XML Document *D* and its XSeed kernel

authors propose an algorithm for the cardinality estimation of the structural XPath expressions. The main contribution of this work is the novel and accurate way of dealing with recursive documents and recursive queries.

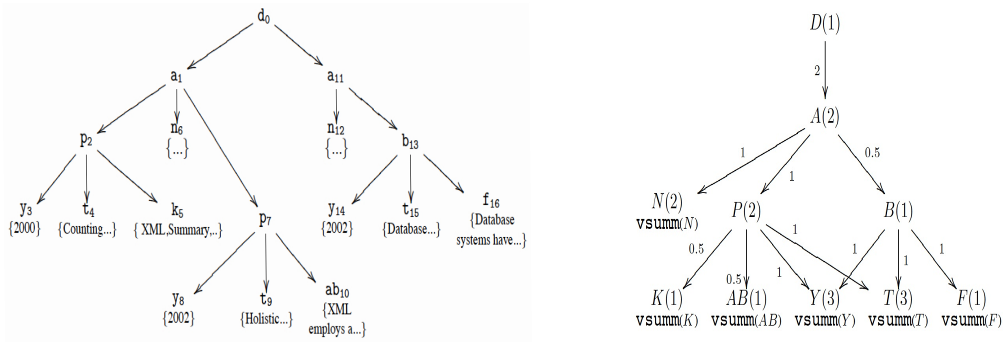
By treating the structural information in a multi-layer manner, the XSeed synopsis is simpler and more accurate than the TreeSketch synopsis. However, although the construction of XSeed is generally faster than that of TreeSketch, it is still time-consuming for complex datasets.

Paper [Polyzotis 2006] introduced XCLUSTER, which computes a synopsis for a given XML document by summarizing both the structure and the content of document. XCLUSTER is considered to be a generalized form of the XSketch tree synopses which is a previous work of the authors presented in [Polyzotis 2002b].

On the structure content side, an XCLUSTER tree synopsis is a node-labeled graph where each node represents a sub-set of elements with the same tag, and an edge connects two nodes if an element of the source node is the parent of elements of the target node. Each node in the graph records the count of elements that it represents while each edge records the average child count between source and target elements.

On the value content side, XCLUSTER has borrowed the idea of the XMill XML compressor [Liefke 2000] which is based on forming structure-value clusters which groups together data values into homogeneous and semantically related containers according to their path and data type. Then, it employs the well-known histogram techniques for numeric and string values [Chaudhuri 2004] [Poosala 1996] and introduces the class of end-biased term histograms for summarizing the distribution of unique terms within textual XML content.

The XCLUSTER estimation algorithm relies on the key concept of a query embedding, that is, a mapping from query nodes to synopsis nodes that satisfies the structural and value-based constraints specified in the query. To estimate the selectivity of an embedding, the XCluster algorithm employs the stored statistical information coupled with a generalized path-value independence assumption that essentially de-correlates path distribution from the distribution of value-content. This approach can support twig queries with predicates on numeric content, string content, and textual content.



(a) XML D (b) XCLUSTER of D

Figure 2.3: An XML document D and its XCLUSTER synopsis

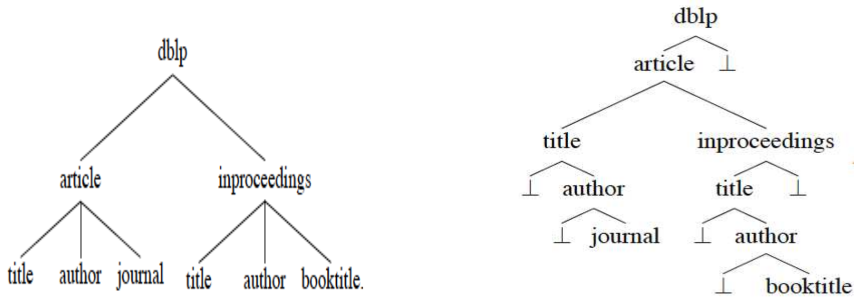
Figure 2.3 illustrates an example of XML document D and its corresponding XCLUSTER.

However, though XCLUSTER address the summarization problem for structured XML content, but its construction time is unknown. Furthermore, as it is mentioned in [Sakr 2010] it does not process a nested expressions (nested predicates).

[Fisher 2007] have proposed the SLT (Straight line tree) XML tree synopsis. The idea of this work is based on the fact that the repetitive nature of tags in the XML documents makes tag mark-ups re-appears many times in a document.

Hence, the authors use the well-known idea of removing repeated patterns in a tree by removing multiple occurrences of equal subtrees and replacing them by pointers to a single occurrence of the subtree. The synopsis is constructed by using a tree compression algorithm to generate the minimal unique directed acyclic graph (DAG) of the XML tree and then representing the resulting DAG structures using a special form of grammars called an straight line tree grammar (SLT grammar).

Figure 2.4 illustrates an example of an XML document D and the representation



(a) An XML D (b) The $bin(D)$

$$
\begin{aligned}
A_0 &\rightarrow \perp \\
A_1 &\rightarrow title(\perp, author(\perp, y_1)) \\
Start = A_2 &\rightarrow dblp(article(A_1(journal), \\
&\quad inproceedings(A_1(booktitle), \\
&\quad \perp)), \perp)
\end{aligned}
$$

(c) SLT grammar for $bin(D)$

Figure 2.4: An XML document D, the binary tree representation $bin(D)$ and SLT of $bin(D)$

of its corresponding binary tree $bin(D)$ and SLT grammar for $bin(D)$. Additionally, the size of this grammar is further reduced by removing and replacing certain parts of it, according to a statistical measure of multiplicity of tree patterns. This results in a new grammar which contains size and height information about the removed patterns.
The authors have described an algorithm for a tree automaton which is designed to run over the generated lossy SLT grammars to estimate the selectivity of queries containing all XPath axes, including the order-sensitive ones. This algorithm converts each XPath query into its equivalent tree automaton and describes how to evaluate this tree automaton over a document to test whether the query has at least one match in the document and returns the size of the result of the query on a document. The proposed synopsis of this work has the ability to support the estimation of all XPath axes using an efficient memory space, however, it unfortunately can deal only with structural XPath queries. Furthermore, as it is mentioned in [Sakr 2010], this approach does not support any form of predicate queries.

Some work also has been conducted to estimate the selectivity for XQuery. The design and implementation of a relational algebraic framework for estimating the selectivity of XQuery expressions was described in [Saker 2007], [Saker 2008], and [Teubner 2008]. In this approach (Relational algebraic), XML queries are translated into relational algebraic plans [Grust 2004]. A peephole-style analysis of these relational plans is performed to annotate each operator with a set of special properties [Grust 2005]. These annotations are produced during a single pass over the relational plan and use a set of light-weight inference rules which are local in the sense that they only depend on the operator's immediate plan inputs. Summary information about the structure and the data values of the underlying XML documents are kept separately. Then by using all these pieces together with a set of inference rules, the relational estimation approach is able to provide accurate selectivity estimations in the context of XML and XQuery domains. The estimation procedure is defined in terms of a set of inference rules for each operator which uses all of the available information to estimate the selectivity of not only the whole XQuery expression but also of each sub-expression (operator) as well as the selectivity of each iteration in the context of FLWOR expressions. The framework enjoys the flexibility of integrating any XPath or predicate selectivity estimation technique and supports the selectivity estimation of a large subset of the powerful XML query language XQuery.

Recently, [Luo 2009] proposed a sampling method named subtree sampling to build a representative sample of XML which preserves the tree structure and relationships of nodes.

They examine the number of data nodes for each tag name starting from the root level. If the number of data nodes for a tag is large enough, a desired fraction of the data nodes are randomly selected using simple random sampling without replacement and then the entire subtrees rooted at these selected data nodes are included, as sampling units, in the sample. They call each such set of subtrees to which random sampling is applied a subtree group. If a tag has too few data nodes at the level under study, then all the data nodes for that tag at that level are kept and they move down to check the next level in the tree.

The paths from the root to the selected subtrees are also included in the sample to preserve the relationships among the sample subtrees. This sampling scheme assumes that the sizes of the subtrees in the same subtree group are similar. This is because the root nodes of these subtrees have the same tag name, *i.e.* they are nodes of the same type. These root nodes reside in the same level. Consequently, subtrees in the same subtree group tend to have similar structures, thus similar sizes. Based on this observation, the sampling fraction of the subtree groups f_i', where f_i is the sampling fraction of the ith subtree group, can be simply set to f_t, which is the sampling fraction of the whole data set.

If the number of nodes n for a tag satisfies the minimum requirements $n * ft >= 1$, they consider it large enough. In figure 2.5 the subtree sampling is applied to the DBLP data set [Ley 2011]. In the second level (directly after the root), there are 10000 "*book*" nodes and 20000 "*article*" nodes. By assuming that the sampling

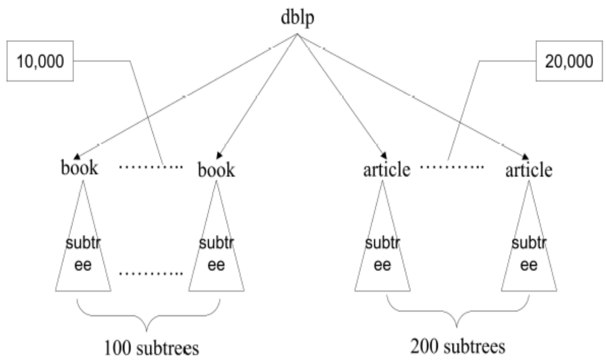


Figure 2.5: Subtree Sampling for DBLP

fraction is 1%, we conclude that both tags have sufficient large numbers of nodes 10000* 1% = 100 and 20000* 1% = 200. Based on that, they randomly select 100 "book" nodes and 200 "article" nodes from the second level and include their subtrees as the sample. Also, they include the paths from the root to the subtrees to preserve the hierarchy.

The sample trees are just a portion of the original XML data tree. These sample trees differ only in magnitude from the original XML data tree. Therefore, ordinary twig query evaluation methods, such as TwigStack [Bruno 2002] can be applied directly to the sample trees synopsis to derive approximate answers.

Though a subtree sampling synopsis can be applied to aggregations functions such as SUM, AVG, etc., this approach is based on an essential assumption that nodes of the same type and at the same level have similar subtrees. Moreover, it is shown in [Luo 2009] that XSeed [Zhang 2006b] outperforms subtree sampling for queries with Parent/Child or Ancestor/Descendant on simple data set e.g. XMarK [Schmidt 2001], while it is the inverse for *recursive* data sets.

2.2.2.2 Histogram-Based Estimation Techniques

As we mentioned before, this class of the estimation techniques uses the statistical histograms for capturing the summary information of the source XML documents. Below, we give a survey on the existing work.

[Freire 2002] have presented an XML Schema-based statistics collection technique called StatiX. This technique leverages the available information in the XML Schema to capture both structural and value statistics about the source XML documents. These structural and value statistics are collected in the form of histograms. The StatiX system is employed in LegoDB [Bohannon 2002]. LegoDB is a cost-based XML-to-relational storage mapping engine, which tries to generate efficient relational configurations for the XML documents.
The StatiX system consists of two main components. The first component is the

XML schema validator which simultaneously validates the document against its associated schema and gathers the associated statistics. It assigns globally unique identifiers (IDs) to all instances of the types defined in the schema. Using these assigned IDs, structural histograms are constructed to summarize information about the connected edges. Value histograms are constructed for types that are defined in terms of base types such as integers. The storage of the gathered statistics is done using equi-depth histograms (wherein the frequency assigned to each bucket of the histogram is the same). The second component is the XML schema transformer which enables statistics collection at different levels of granularity. Although, StatiX is used in the context of the LegoDB system and the presented experimental results indicate highly accurate query estimates, the technique can only be applied to documents described by XML schemas with no clear view as to how it can be extended to deal with schema-less documents. Moreover, the paper [Freire 2002] does not show a clear algorithm for estimating the cardinality of the XQuery expression and there is no clear definition of the supported features and expressions of the language.

[Wu 2002] have presented an approach for mapping XML data into 2D space and maintaining certain statistics for data which fall into each pre-defined grid over the workspace. In this approach, each node x in an XML document D is associated with a pair of numbers, $start(x)$ and $end(x)$, numeric labels representing the pre-order and post-order ranks of the node in the XML document D. Each descendant node has an interval that is strictly included in its ancestors interval. For each basic predicate P a two-dimensional histogram summary data structure is built and collectively named as position histograms. In the position histograms data structure, the $start$ values are represented by the $x-axis$ while the end values are represented the $y-axis$. Each grid cell in the histogram represents a range of start position values and a range of end position values. The histogram maintains a count of the number of nodes satisfying the conditions of predicate P and has start and end positions within the specified ranges of the grid cell. Since the start position and end position of a node always satisfies the formula $start <= end$, none of the nodes can fall into the area below the diagonal of the matrix. So, only the grid cells that reside on the upper left of the diagonal can have a count of more than zero.

Given a predicate P_1 associated with the position histograms H_1 and a predicate P_2 associated with the position histograms H_2, estimating the number of pair of nodes u,v where u satisfies P_1 and v satisfies P_2 and u is an ancestor of v is done either in an $ancestor-based$ fashion or in a $descendant-based$ fashion. The $ancestor-based$ estimation is done by finding the number of descendants that joins with each ancestor grid cell. The $descendant-based$ estimation is done by finding the number of ancestors that are joined with each descendant grid cell.

The authors presented another type of histograms named *coverage histogram* to increase the accuracy of the estimation in cases where the schema information is available. For a given predicate P, using the schema information it can be known if the two nodes satisfying the predicate do not have any ancestor-descendant relationship. To deal with this no-overlap situation, additional information is stored in the form of a *coverage histogram*. The *coverage histogram* for a predicate P $Cvgp[i][j][m][n]$ represents the number of nodes in cell (i, j) that are descendants of nodes in cell (m,n) satisfying P. Although, the model handles the estimation of twig queries well, it is very limited to the ancestor and descendant paths and has no clear way for extension to the other paths. Moreover, the number of constructed *position histograms* is proportional to the number of the interested predicates which is considered to be relatively high.

Follow-up work has improved on the ideas of interval histograms by leveraging adaptive sampling techniques [Wang 2003]. In this work, the proposed technique treats every element in a node set as an interval, when the node set acts as the ancestor set in the join or a point or when the node set acts as the descendant set. Two auxiliary tables are then constructed for each element set. One table records the coverage information when the element set acts as the ancestor set, while the other captures the start position information of each element when the element set acts as the descendant set. To improve the accuracy of the estimated results, sampling-based algorithms are used instead of the two-dimensional uniform distribution assumption as used in [Wu 2002].

[Wang 2004] have proposed a framework for XML path selectivity estimation in a dynamic context using a special histogram structure named *bloom histogram* (BH). *BH* keeps a count of the statistics for paths in XML data. Given an XML Document D, the path-count table $T(path, count)$ is constructed such that for each $path_i$ in D, there is a tuple t_i in T with $t_i.path = path_i$ and $t_i.count = count_i$ where $count_i$ is the number of occurrences of $path_i$. Using T, a bloom histogram H is constructed by sorting the frequency values and then grouping the paths with similar frequency values into buckets. Bloom filters are used to represent the set of paths in each bucket so that queried paths can be quickly located.

Path	Count
/a	10
/a/f	10
/a/e	499
/a/c	501
/a/b	999
/a/d	1001

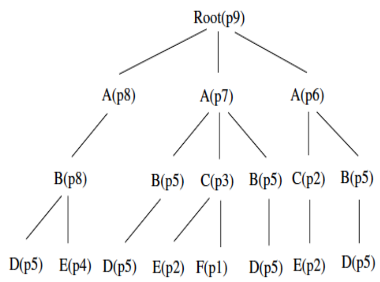
Bloom Filter	Count
BF (/a, /a/f)	10
BF (/a/c, /a/e)	500
BF (/a/d, /a/b)	1000

Figure 2.6: An example path-count table and its bloom histogram. $BF(P)$ is a bloom filter for a set of paths

Figure 2.6 illustrates an example of path-count table and its corresponding bloom histogram.

To deal with XML data updates and the dynamic context, the authors proposed a dynamic summary component which is an intermediate data structure from which the bloom histogram can be recomputed periodically. When data updates arrive, not only the XML data is updated but the updated paths are also extracted, grouped and propagated to the dynamic summaries. Although, the bloom histogram is designed to deal with data updates and the estimation error is theoretically bounded by its size, it is very limited as it deals only with simple path expressions of the form $/p1/p2/.../pn$ and $//p1/p2/.../pn$.

[Li 2006] have described a framework for estimating the selectivity of XPath expressions with a main focus on the order-based axes (following, preceding, following-sibling, and preceding-sibling). They used a path encoding scheme to aggregate the path and order information of XML data. The proposed encoding scheme uses an integer to encode each distinct root-to-leaf path in the source XML document and stores them in an encoding table. Each node in the source XML document is then associated with a path id that indicates the type of path where the node occurs. Additionally, they designed a PathId-Frequency table where each tuple represents a distinct element tag and aggregates all of its associated element tags with path ids and their frequency. To capture the order information, they used the Path-order table associated to each distinct element tag name to capture the sibling-order information based on the path ids. Figure 2.7 illustrates an example of XML document and its corresponding path encoding scheme.



(a) An XML *D*

Root–to–leaf	Encoding
Root/A/B/D	1
Root/A/B/E	2
Root/A/C/E	3
Root/A/C/F	4

(b) Encoding Table

Bit–Seq	Int
0001	p1
0010	p2
0011	p3
0100	p4
1000	p5
1010	p6
1011	p7
1100	p8
1111	p9

(c) Path Id Table

Figure 2.7: An XML document *D* and its path encoding scheme

For estimating the cardinality of XPath expressions, the authors introduced the Path Join algorithm. Given an XPath query Q, the path join retrieves a set of path ids and the corresponding frequencies for each element tag in Q from the PathId-Frequency table. For each pair of adjacent element tags in Q, they use a nested loop to determine the containment of the path ids in their sets. Path IDs that clearly do not contribute to the query result will be removed. The frequency values of the remaining path ids will be used to estimate the query size. The algorithm uses the information of the path-order table to compute the selectivity of the (following-sibling, preceding-sibling) axes that may occur in Q. The XPath expression which involves the preceding or following axes is converted into a set of XPath expressions involving only preceding-sibling or following-sibling axes according to the path ids of the nodes associated with preceding or following axes after the path ID join. Then, the estimation result is given by the selectivity sum of the set of path expressions. The authors introduced two compact summary structures called p-histogram and o-histogram, to summarize the path and order information of XML data respectively. A p-histogram is built for each distinct element tag to summarize the pathId-frequency information. In this histogram, each bucket contains a set of path ids and their average frequency value. Based on the observation that the path-order table is very sparse and the frequencies in the majority of the cells are zero, the o-histogram is designed to summarize the path-order information where only the cells with non-zero values are stored. Although, the proposed model is the first work to address the problem of cardinality estimation of XPath expression with order-based axes, it is unfortunately not clear how an extension can be introduced to support predicates.

2.2.3 Summary - The Choice of the Path tree Synopsis

Some of the main usages of selectivity estimation techniques are to accelerate the performance of the query evaluation process and to estimate the cost for a given query. Furthermore, a good technique should be able to provide accurate estimates for a large fragment of XPath. These techniques should also support structural and data value queries. The synopsis of the estimation technique should be constructed rapidly (one pass on the XML Data) for the different types (deep, large, recursive,..etc.) of XML data sets. In addition, the required summary structure(s) for achieving the selectivity estimation process must be efficient in terms of memory and space consumption.
Our main objective is to build an estimation selectivity technique for the fragment of Forward XPath (defined in 1.1.1.2) with the above mentioned features.

Below, we summary and compare the related work based on the following criteria:

- **The Fragment of XPath**: Some techniques estimate the selectivity for only path expressions and they do not support twigs, e.g., [Aboulnaga 2001], [Wang 2004], [Li 2006], and [Fisher 2007]. Others, support twigs with structural queries only, so can not support twigs with $text()$, e.g., [Zhang 2006b] and [Polyzotis 2004a].

  The XSeed technique [Zhang 2006b] can not process a nested expressions (nested predicates) [Sakr 2010]. While the TreeSktech technique

[Polyzotis 2004a] does not support queries with Ancestor-Descendant relationships neither queries with $'text()'$ [Luo 2009].

The XCLUSTER [Polyzotis 2006] addresses the summarization problem for structured XML content, but its construction time is unknown. Furthermore, as it is mentioned in [Sakr 2010] it does not process a nested expressions (nested predicates).

Some work has been conducted to estimate the selectivity for XQuery.
The design and implementation of a relational algebraic based framework for estimating the selectivity of XQuery expressions was described in [Saker 2007], [Saker 2008].

- **The construction time of the Synopsis**: few papers present the time needed to construct their synopses (summaries). The construction time of TreeSktech [Polyzotis 2004a] for the complex data set TreeBank 86MiB (depth 36) took more than 4 days, this result was confirmed in [Luo 2009]. XSeed treats the structural information in a multi-layer manner, the XSeed synopsis is simpler and more accurate than the TreeSketch synopsis. However, although the construction of XSeed is generally faster than that of TreeSketch, it is still time-consuming for complex datasets.

  Other techniques, do not present the construction time for their synopses (summaries), for example: XCLUSTER [Polyzotis 2006].

- **Recursion in the data set** : the XSeed and the XCLUSTER synopses are more general than the TreeSketch synopsis because the latter does not support the recursion in the data sets as it is explained in [Zhang 2006b].

- **Selectivity of structural queries and synopsis size**: several structure synopses, such as Correlated Suffix Trees [Chen 2001], Twig-Xsketch [Polyzotis 2004b], TreeSketch [Polyzotis 2004a], and XSeed [Zhang 2006b] store some form of compressed tree structures and simple statistics such as node counts, child node counts, etc. Due to the loss of information (in particularly the structure of the original data set), selectivity estimation heavily relies on the statistical assumptions of independence and uniformity. Consequently, they can suffer from poor accuracy when these assumptions are not valid. The above proposed structures synopses can not be evaluated by ordinary query evaluation algorithms, they require specialized estimation algorithms.

- **Incremental update**: minimal synopsis size seems desirable but won′t be the best because incremental maintenance would be difficult [Goldman 1997].

This is the case of many selectivity estimation techniques such as: Correlated Suffix Trees [Chen 2001], TreeSketch and XSeed.

The path tree structure synopsis was introduced by [Aboulnaga 2001] to estimate the selectivity for path expression only. This structure was used by [Zhang 2005]. But to the best our knowledge, this structure was not formally defined in the literature. It has overall advantages like complete structural information and the possibility of being evaluated by streaming algorithms. This synopsis captures the structure of the XML document and permits by using an efficient stream-querying algorithm to estimate efficiently the selectivity for any query belongs to the fragment of Forward XPath.

We propose to use a stream-querying algorithm and an adapted path tree synopsis to optimize and improve the space consumption of the selectivity estimation process.

In the next section 2.3, we present several stream-processing approaches, we then compare them to find the best approach that can be used to traverse the path tree structure synopsis and to estimate efficiently the selectivity for any query which belongs to the fragment of Forward XPath.
Then the next chapter 3, we formally define the path tree synopsis. Furthermore, we give different algorithms to construct and update it.

2.3 Stream-processing Approaches

Much research has been conducted to study the processing of XML documents in streaming fashion. The different approaches to evaluate XPath queries on streams of XML data can be categorized as follows (1) *stream-filtering*: determining whether there exists at least one match of the query Q in the XML document D, yielding a boolean output, for example XTrie [Chan 2002]. (2) *Stream-querying*: finding which parts of D match the query Q. This implies outputting all answer nodes in a XML document D *i.e.* nodes that satisfy a query Q. An example of stream-querying research is XSQ [Peng 2003].

Below, we present some existing algorithms for each category.

2.3.1 Stream-filtering Algorithms

A filtering system delivers documents to users based on their expressed interests (queries). Figure 2.8 shows the context in which a filtering system operates. There are two main sets of inputs to the system: user profiles (queries) and the stream of documents. User profiles describe the information preferences of individual users. These profiles may be created by the users themselves, e.g., by choosing items in a Graphical User Interface, or may be created automatically by the system using machine learning techniques. The user profiles are converted into a format that can be efficiently stored and evaluated by the filtering system. These profiles are effectively standing queries which are applied to all incoming documents. Hereafter, profiles and queries are used interchangeably.

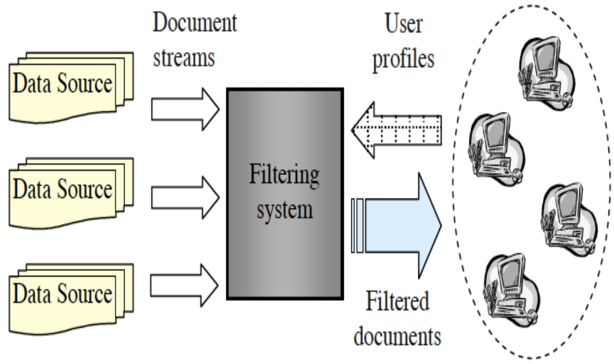


Figure 2.8: Overview of a filtering system

The other key inputs to a filtering system are the document streams containing continuously arriving documents from data sources. These documents are to be filtered and delivered to users or systems in a timely fashion. Filtering is performed by matching each arriving document against all of the user profiles to determine the set of interested users. The document is then delivered to this set of users. In our system, documents are processed one-at-a-time. That is, incoming documents are placed in a queue, a document is removed from the queue and processed in its entirety (*i.e.,* matched with all relevant queries) before processing is initiated for the next document. As filtering systems are deployed on the internet, the number of users for such systems can easily grow into the millions. A key challenge in such an environment is to efficiently and quickly search the huge set of user profiles to find those for which the document is relevant.

Various stream-filtering systems have been proposed. Below we explain some of them.

XFilter [Altinel 2002] is the first filtering system that addresses the processing of streaming XML data. It was proposed for for selective dissemination of information (SDI). For structure matching, XFilter adopts some form of Finite State Machine (FSM) to represent path expressions in which location steps of path expressions are mapped to machine states. Arriving XML documents are then parsed with an event-based parser, the events raised during parsing are used to drive the FSMs through their various transitions. A query is said to match a document if during parsing, an accepting state for that query is reached.

In the filtering context, large numbers of queries representing the interests of the user community are stored and must be checked upon the arrival of a new document. In order to process these queries efficiently, XFilter employs a dynamic index over the states of the query FSMs and includes optimizations that reduce the number of path expressions that must be checked for a given document. In large-scale systems there is likely to be significant commonality among user interests, which could result in redundant processing in XFilter.

YFilter [Diao 2002] is an XML filtering system aimed at providing efficient filtering for large numbers (e.g., $10's$ or $100's$ of thousands) of queries. The key innovation in YFilter is an Nondeterministic Finite Automaton (NFA)-based representation of path expressions which combines all queries into a single machine. Figure 2.9 illustrates an examples of this NFA, where all common prefixes of the paths are represented only once in the NFA.

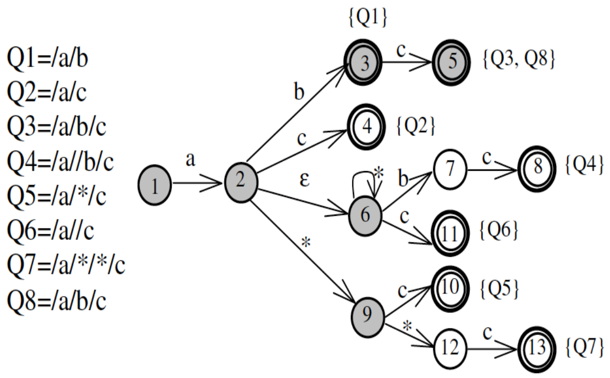


Figure 2.9: XPath queries and a corresponding NFA

A basic path matching engine of YFilter is designed to handle query that are written in a subset of XPath. YFilter focuses on two common axes: the parent-child axis $'/'$, and the ancestor-descendant axis $'//'$. It support node tests that are specified by either an element name or the wildcard $'*'$ (which matches any element name). Predicates can be applied to address contents of elements or to reference other elements in the document.

In [Böttcher 2007] a SAX Based approach is introduced to evaluate the XPath queries that support all axes of *Core XPath*. Each input query is translated into an automaton that consists of four different types of transitions. The small size of the generated automata allows for a fast evaluation of the input stream of XML data within a small amount of memory. The authors implemented a prototype called XPA. The query processor decomposes and normalizes each XPath query, such that the resulting path queries contain only three different types of axes, and then converts them into lean XPath automata for which a stack of active states is stored. The input SAX event stream is converted into a binary SAX event stream that serves as input of the XPath automata.
In [Böttcher 2007], it is shown that XPA consumes far less main memory than

YFilter [Diao 2002]. XPA consumes from 20% of the document size on average for simples XPath queries without predicate filters up to 50 % of the document size on average for paths with predicate filters.

The XTrie [Chan 2002] technique is built on top of the XFilter approach and claims 2-4 times improvement in speed over the XFilter [Altinel 2002] system. Its authors proposed a trie-based index structure, which decomposes the XPath expressions (XPEs) to substrings that only contain parent-child axis. As a result, the processing of these common substrings among XPath expressions (XPEs) can be shared.

The three key prominent features of XTrie can be summarized as follows: (1) it can filter based on complex and multiple path expressions, (2) it supports both ordered and un-ordered matching of XML documents, (3) since XTrie uses substrings, instead of elements names to index, the authors claim that XTrie can reduce both the number of unnecessary index probes and avoid redundant matching.
Figure 2.10 illustrates an example of XTrie. First, the XPath queries are decomposed into substrings, then, in the substring-table ST a row is created for each substring of each indexed XPE. Finally, the trie T is created. T is a rooted tree constructed from the set of distinct substrings S, where each edge in T is labeled with some element name. Each node N in T has two special pointers: (1) the substring pointer points to some row in ST (2) The Max-suffix pointer points to some internal node in T and its purpose is to ensure the correctness of the matching algorithm. The substring-table ST contains one row for each substring of each indexed XPE.

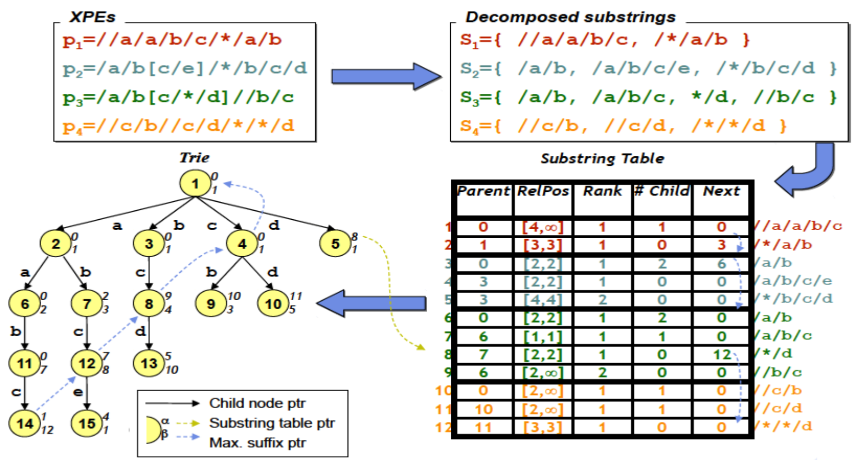


Figure 2.10: XTrie example.

XTrie is designed to support online filtering of streaming XML data and is based on the SAX event-based interface that reports parsing events, the search procedure for the XTrie, which accepts as input an XML document D and an

XTrie index (ST, T), processes the parsing events generated by D, and returns the identifiers of all the matching XPEs in the index. XTrie outperformed XFilter [Altinel 2002]. But YFilter [Diao 2002] has demonstrated a better performance than XTrie on certain workloads.

In [Bar-Yossef 2004] the authors initialized a systematic and theoretical study of lower bounds on the amount of memory required to evaluate XPath queries over streams of XML data. They present a general lower bound technique, which given a query, specifies the minimum amount of memory that any algorithm evaluating the query on a stream would need to incur.
The first memory lower bound is the *query frontier size*. When a query Q is represented as a tree, the frontier size at a node of this tree is the number of siblings of this nodes, and its ancestors$'$ siblings. The query frontier size of Q is the largest frontier over all nodes of Q. The second lower bound is the document *recursion depth*. The recursion depth of a tree t with respect to a query Q is the maximal number of nested nodes matching a same node in Q. The third lower bound is the logarithmic value $log(d)$, where d is the depth of the document t.
Based on these bounds a stream-filtering algorithm was proposed to optimize the space complexity, it deviates from some paradigms that use automata or transducers. The algorithm transforms the query into NFA and uses different arrays for matching the stream of XML data. For queries in the fragment of $Univariate\ XPath$, the space complexity of the algorithm is $O(|Q|.\ r.\ (log|Q| + log\ d + log\ r))$, where $|Q|$ is the query size, r is the document recursion depth, and d is the document depth. The time complexity is $O(|D|.|Q|.r)$, where $|D|$ is the document size.

XPush machine [Gupta 2003] was proposed to improve the performance of stream-filtering. It processes a large number of XPath expressions, each with many predicates, on a stream of XML data. The XPush machine is constructed lazily by creating an AFA (Alternating Finite Automaton) for each expression, and then transforming the set of AFAs into a single DPDA (Deterministic Pushdown Automaton). This is similar to the algorithm for converting an NFA to a DFA as described in the standard textbook on automata where stack automata are defined [Hopcroft 1979].
Existing systems (e.g. YFilter [Diao 2002]) can identify and eliminate common subexpressions in the structure navigation part of XPath queries. This technique focuses on eliminating redundant work in predicate evaluation part. For examples, given the following two path expression
$P1 = //a[./b/text() = 1\ and\ .//a[@c > 2]]\ and$
$P2 = //a[@c > 2\ and\ ./b/text() = 1]$, previous techniques cannot exploit the fact that the predicate $[./b/text() = 1]$ is common.

Since inherently the XPush machine cannot be partially updated, addition of a single expression necessitates recalculation (*i.e.,* reconstruction) of the XPush machine as a whole. In other words, the cost of updating an automaton depends on the total number of AFAs (or expression).To solve this problem [Takekawa 2007] proposed an integrated XPush machine, which enables incremental update by constructing the whole machine from a set of sub-XPush machines. The evaluation result positively demonstrates that efficient partial change of the AFAs is possible without significantly affecting all of the state transition tables.

Recently, SFilter [Nizar 2009b] was proposed. SFilter indexes the queries compactly using a query guide and uses simple integer stacks to efficiently process the stream of XML data.
A *query guide* G is an ordered tree representation of all the path expressions that exploits the prefix commonality between the path expressions such that (i) the root of G is the same as the dummy root $'r'$ of the path path expressions and (ii) the root-to-result node path of each path expression appears in G as a path that starts at node $'r'$ and ends at a descendant node and the path has the same node labels and edge constraints (*i.e.,* P-C or A-D edge) of the path expression.
The basic idea of this approach is to process the streaming XML data one tag at a time using the query guide representing the given path expressions. At any time during execution, the algorithm maintains a sequence of elements S in the stream whose open-tags have been seen but close-tags are yet to arrive. It maintains an integer stack at every query guide node to keep track of the current sequence of tags S in the stream. Each value in the stack represents the depth of an element in the stream that matches with the query guide node to which the stack is associated. Note that this number can uniquely identify a node in the stream as there will be exactly one node at a given depth in the current (or active) path in the document tree, represented by S. The input stream of XML data is first parsed by a SAX parser that generates a stream of SAX events, which is input to the query processor. The algorithm starts by pushing a depth value 0 into the stack for the root node r of the query guide. It then proceeds by responding to the open-tag and close-tag events generated by the SAX parser.
One problem with the basic query guide and the corresponding algorithmic approach mentioned above is the overhead associated with wildcard node processing. Note that, since wildcard matches any tag, the query guide nodes with wild card label are to be processed for every element in the stream. This overhead can be partly overcome by what it is called the *vertical compression* of the query guide and slight modifications of the event processors.

Figure 2.11 (b) illustrates the query guide of the path queries of figure 2.11 (a). Note that, figure 2.11 (c) is a representation of the query guide of (b) where the query edges are labeled with the expected depth. While figure 2.11 (c) is the ver-

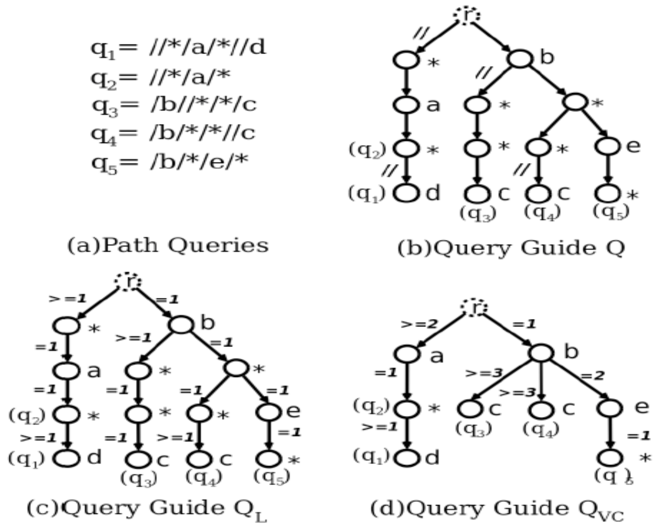


Figure 2.11: A data guide and its vertical compression.

tical compression, they vertically collapse paths in the query guide by eliminating wildcard nodes. For example: consider a path $a \rightsquigarrow * \rightsquigarrow b$ in the query guide. This path can be collapsed into a path without the wild card node. While doing so, the expected depth labels in paths a $\rightsquigarrow b$ and $* \rightsquigarrow b$ are combined.

Thought that SFilter outperforms YFilter [Diao 2002] in term of time and space for path expressions, this approach does not support predicates.

Stream-filtering approaches deliver whole XML documents which satisfy the filtering condition to the interested users. Thus, the burden of selecting the interesting parts from the delivered XML documents is left upon the users. We therefore concentrate on stream-querying as more general and useful approach for our performance prediction (cost) model.

2.3.2 Stream-querying Algorithms

Holistic XML matching algorithms are prevalent for matching pattern queries over stored XML data. They demonstrate good performance due to their ability to minimize unnecessary intermediate results. In particular, [Bruno 2002] proposed the first merge-based algorithm, which scans input data lists sequentially to match twig patterns. Such merge-based algorithms can be further improved by structure indexes that can reduce sizes of input lists [Chen 2005]. Index-based holistic joins [Jiang 2003] were also proposed to speedup the matching of selective queries, as an improvement over merge-based algorithms. In contrast, streaming algorithms assume that XML documents are not parsed in advance and they come in the form of SAX events. Sometimes even ad-hoc XML documents can be regarded as streams of XML data if using a SAX parser is the best way to access them.

A large amount of work has been conducted to process XML documents in streaming fashion. The different stream-querying approaches to evaluate XPath queries on XML data streams can be categorized by the processing approach they use. Most of them are *automata based*, for example: XPush [Gupta 2003], XSQ [Peng 2003], SPEX [Olteanu 2007] or *Parse tree based*, for example: [Chen 2006], [Barton 2003], [Gou 2007]. We highlight below some of the existing work .

In [Peng 2003], authors proposed XSQ a method for evaluating XPath queries over streams of XML data to handle closures, aggregation and multiple predicates. Their method is designed based on hierarchical arrangement of pushdown transducers augmented with buffers. Automata is extended by actions attached to states, extended by a buffer to evaluate XPath queries.
The basic idea of XSQ is to use a pushdown transducer (PDT) to process the events that are generated by a SAX parser when it parses XML streams. A PDT is a pushdown automaton (PDA) with actions defined along with the transition arcs of the automaton. A PDT is initialized in the start state. At each step, based on the next input symbol and the symbols in the stack, it changes state and operates the stack according to the transition functions. The PDT also defines an output operation which could generate output during the transition. In the XSQ system, the PDT is augmented with a buffer so that the output operation could also be the buffer operation.
Notice that the PDT generated for each location step of an XPath expression is called a basic pushdown automaton (BPDT). The BPDTs are combined into one Hierarchical PDT (HPDT).

Figure 2.12 illustrates the HPDT of the query
$//pub[./year > 2000]//book[./author]//name/text()$. The figure shows how BPDTs are combined into one HPDT.
As it is shown in [Gou 2007] XSQ does not support the AND operator. Further, XSQ does not support same node-labels in a query, and requires that each axis node have at most one $('/')$ predicate node child.

[Chen 2006] proposed a lazy stream-querying algorithm, TwigM, to avoid the exponential time and space complexity incurred by XSQ. TwigM extends the multi-stack framework of the TwigStack algorithm [Bruno 2002]. It uses a compact data structure to encode patterns matches rather than storing them explicitly which is a memory advantage. After that, it computes query solution by probing the compact data structure in Lazy fashion without computing pattern matches. The output consists of XML fragments.
In [Chen 2006], it is shown that TwigM can evaluate Univariate XPath in polynomial time and space in the streaming environment. Specifically, TwigM works in $O(|D|.|Q|(|Q|+d_D.B))$ time and uses $O(|Q|.r)$ caching space. Where r is the

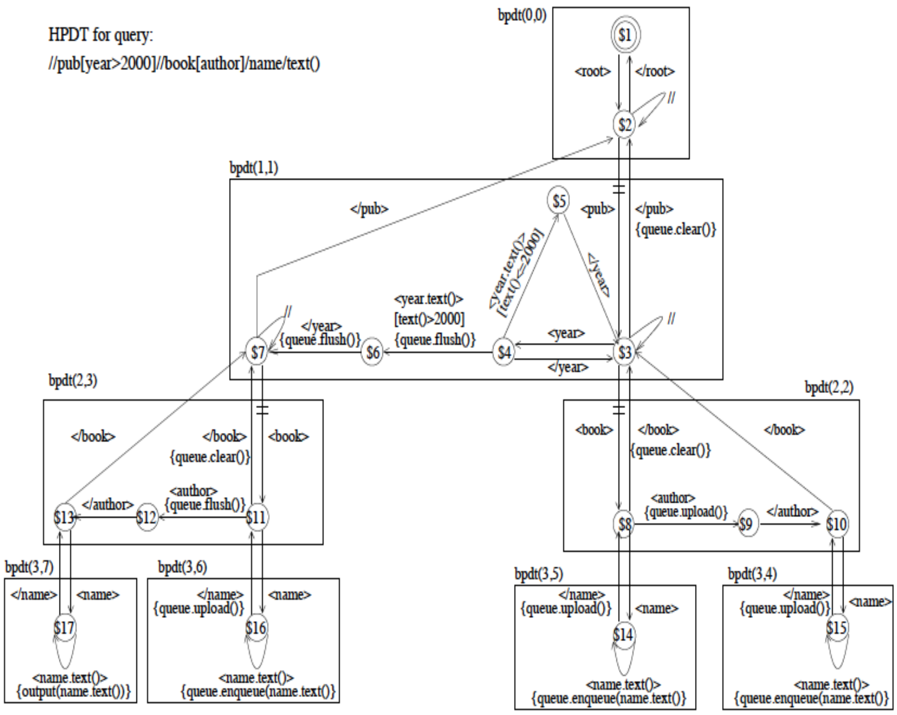


Figure 2.12: HPDT of $//pub[./year > 2000]//book[./author]//name/text()$

recursion in D and d_D is the maximum depth of D. However, like XSQ, TwigM might have to buffer multiple physical copies of a potential answer node at a time, which is a space problem for recursive documents or data sets.

The SPEX [Bry 2005] [Olteanu 2007] system processes XPath expressions with forward axes by mapping it to a network of transducers. Query re-writing methods [Olteanu 2002] are used to transform expressions with backward axes to ones containing only forward axes. Most transducers used are single-state pushdown automata with output tape. For path expressions without predicates, the transducer network is a linear path; otherwise, it is a directed acyclic graph. Each transducer in the network processes, in stepwise fashion, the stream of XML data it receives and transmits it unchanged or annotated with conditions to its successor transducers.
The transducer for the result node holds potential answers, to be output when conditions specified by the query are found to be true by the corresponding transducers. Due to the absence of built-in order information, the system processes and caches large number of stream elements which will be found useless later.

TurboXPath [Josifovski 2005] is an XML stream processor evaluating XPath expressions with downward and upward axis, together with a restricted form of for-let-where (FLOWR in XQuery) expressions. Hence, TurboXPath returns

tuples of nodes instead of nodes.
In TurboXPath [Josifovski 2005] the input query is translated into a set of parse trees. Whenever a matching of a parse tree is found within the stream of XML, the relevant data is stored in form of a tuple that is afterward evaluated to check whether predicate and join conditions are fulfilled. The output is constructed out of those tuples of which have been evaluated to true.

In [Han 2008] the authors studied the problem of extracting flattened tuple data from streaming, hierarchical XML data. For this goal, they proposed StreamTX, in this approach they adapt the holistic twig joins for tuple-extraction queries on streaming XML with two novel features: first, they use the block-and-trigger technique to consume streaming XML data in a best-effort fashion without compromising the optimality of holistic matching; second, to reduce peak buffer sizes and overall running times, they apply query-path pruning and existential-match pruning techniques to aggressively filter irrelevant incoming data.According to their experiments, StreamTX has demonstrated superior performance advantage over TurboXPath [Josifovski 2005], both for positive queries and negative queries. The advantage is particularly significant for negative queries.

Some stream-querying systems for evaluating XQuery queries have been developed, such as BEA/XQRL [Florescu 2003], Flux [Koch 2004], and XSM [Ludascher 2002].

[Zhang 2006a] introduced a streaming XPath algorithm (QuickXScan). It is based on the principles similar to that of attribute grammars. There is a nice solution of using compact stacks to represent a possibly combinatorial explosive number of matching path instantiations with linear complexity like [Jiang 2003], therefore, QuickXScan extends the idea of compact stacks in a technique called matching grid, which is used also in [Ramanan 2005]. QuickXScan represents queries using a query tree, together with a set of variables and evaluation rules associated with each query node. In this approach, there is a set of interrelated stacks, one for each query node to keep XML data nodes that match with the query node. Active query nodes can be precisely tracked with maximum up to the query size.
Though, this approach handles queries containing child and descendant axes with complex predicates, it is not clear whether it supports queries with wildcard.
The time complexity of this QuickXScan $O(|Q|.r.|D|)$ while the space complexity is $O(|Q|.r)$, where $|Q|$ is the query size, $|D|$ is the document size, and r is the recursion in the document.

The authors of [Chen 2004] presented a model of data processing for information system exchange environment. It consists of a simple and general encoding scheme for servers, and algorithms of streaming query processing on encoded stream of XML data for data receivers with constrained computing abilities "binary encoding". The EXPedite query processor takes an encoded stream of XML data and an encoded XPath query as input, and outputs the encoded fragment in the stream of XML data that matches the query. The idea of the query processing algorithm is taken from different proposed techniques [DeHaan 2003], [Grust 2002] for efficient query evaluation based on XML node labels for XML data stored in the database.

In [Gou 2007] authors proposed two algorithms to evaluate XPath over streams of XML data, they are (1) Lazy streaming algorithm (LQ). (2) Eager streaming algorithm (EQ). Algorithms accept XML document as a stream of SAX events. The fragment of XPath used is called *Univariate XPath*. The goal of both algorithms is to prove that Univariate XPath can be efficiently evaluated in $O(|D|.|Q|)$ time in the streaming environment and to show that algorithms are not only time-efficient but also space-efficient.
These algorithms take two input parameters. The first one is the XPath expression (which respects Univariate XPath to allow stream-processing) that will be transformed to a query table throughout stream processing and statically stored on the memory . After that, the main function is called. It reads the second parameter (XML in SAX events syntax) line by line repeatedly, each time generating a tag. Based on that tag a corresponding $startBlock$ or $endBlock$ function is called to process it. Finally, the main function generates as output the result of the XPath query.
Both algorithms were proposed to handle two challenges of stream-querying that were not solved by XSQ [Peng 2003] and TwigM [Chen 2006]. These challenges are: recursion in the XML document and the existence of same node-labels in the XPath expression.
Based on their experiments, both LQ and EQ algorithms show very similar time performance in practice. In non-recursive (there are no nodes of a certain type can be nested in another nodes of the same type) cases, LQ and TwigM [Chen 2006] has the same buffering space costs, as well as, EQ and XSQ [Peng 2003] has the same cost.

In [Nizar 2008] the authors other proposed an approach for encoding and matching XPath queries with forward (child, descendant, following, following-sibling) axes against streaming XML data. For this purpose, they propose an *Order-aware Twig* (OaT) that is a tree structure rooted at a node labeled $'r'$ known as the root of the OaT. There are three types of relationship edges P-C edge, A-D

edge and closure edge (it is used to handle XPath expressions containing an axis step with $following - sibling$). Moreover, OaT has two types of constraint edges *LR* edge and *SLR* edge.
The match of an OaT against an XML document is a mapping from nodes in the OaT to nodes in the document satisfying the node labels and relationships and constraints between the nodes of the OaT.
The algorithm processes branches of the twig in left-to-right order. A branch is never processed unless constraints specified in the preceding branches are satisfied by the stream. Also, the algorithm avoids repeated processing of branches whose constraints have already been satisfied by the stream. The complexity of this algorithm is not given, and only experimentally studied. Recently, the authors also investigate the streaming evaluation of backward axes [Nizar 2009a].

2.3.3 Summary - Lazy Stream-querying Algorithm LQ

In this section, we highlight some important features required in the stream-querying algorithm that we seek, then, we justify our choice for the the lazy stream-querying algorithm LQ.

1. It is well known that XPath can be efficiently evaluated in $O(|D|.|Q|)$ time in a non-streaming environment, where $|D|$ is the XML data size and $|Q|$ is the XPath query size. However, it has been an open problem whether such $O(|D|.|Q|)$ time performance could be achieved in a streaming environment. In fact, many existing streaming algorithms incur much higher time costs than $O(|D|.|Q|)$. Therefore we need an algorithm which processes the fragment of Forward XPath in $O(|D|.|Q|)$ time.

2. We note that XPath features such as (multiple and nested) predicates, closures (descendant axis $'//'$), same node-labels, and aggregations are important usability advantages, especially if the data is semi-structured or has a structure unknown to the query formulator. It is difficult to write a useful query on data whose structure is (partly) unknown without using closure. Similarly, predicates permit a more accurate delineation of the data of interest, leading to smaller, and more usable results. The challenges posed by these features are exacerbated by data that has a recursive structure. A survey of 60 real datasets found 35 to be recursive [Choi 2002]. Therefore, We need and efficient algorithm which handles these features of XPath.

Some XPath or XQuery stream-querying systems, such as BEA/XQRL [Florescu 2003], TurboXPath [Josifovski 2005], and XSM [Ludascher 2002] are not publicly available at this time, while some publicly available XPath or XQuery querying systems, such as Galax [Fernández 2010], XMLTaskForce [Gottlob 2002] and Saxon [Kay 2010], use non-streaming algorithms. XSQ is an open-source system [Peng 2003], while TwigM [Chen 2006] is not publicly available at this time.

The XMLTK system [Green 2003] does not support predicates in XPath expressions. Therefore, whenever it encounters an element that matches the path expression in a query, it can write it to output. In contrast, if the query includes predicates, the membership of an element in the query result cannot be decided immediately in general. The XSM system [Ludascher 2002] handles predicates in the query but it does not handle closures and aggregations (it assumes that the query does not contain the axis $'//'$). As it is explain in [Gou 2007], both XSQ and TwigM do not handle efficiently the recursive structure of the documents XML, neither the existence of the same node-labels in the XPath expression. XSQ and TwigM might have to buffer multiple physical copies of a potential answer node at a time.
Other approaches handle the complete fragment of Forward XPath, for example [Nizar 2008], unfortunately the complexity of this approach is unknown.

In [Gou 2007], authors proved that Univariate XPath can be efficiently evaluated in $O(|D|.|Q|)$ time. Moreover, the proposed algorithms handle recursion in the XML document and existence of the same node-labels in the XPath expression efficiently. Furthermore, their algorithm is clearly explained and can be extended to process the fragment of Froward XPath without changing its complexity.

For these reasons, we chose the Lazy stream-querying algorithm (LQ) of [Gou 2007] as basis of our work. As we will explain later (chapter 4), this algorithm will be extend to handle: $'text()'$, attributes, predicates with ($'and'$, $'or'$, $'not'$), and nested predicates. Then, our selectivity estimation algorithm will be based on the extended LQ algorithm.

In the next chapter 3, we present the *path tree*, a structure for XML-summarization that is used for accurate selectivity estimates, which was informally introduced by [Aboulnaga 2001] and used by [Zhang 2005]. Furthermore, we introduce two techniques to construct this synopsis structure. Finally, we explain the incremental construction process and the updating of the path tree.
In chapter 4, we present our selectivity estimation technique which uses the path tree structure synopsis and our selectivity estimation algorithm (that is inspired from the lazy stream-querying algorithm LQ [Gou 2007]) to estimate the selectivity for any query which belongs to the fragment of Forward XPath.

Chapter 3

Path tree: Definition, Construction, and Updating

Contents

3.1 Introduction

Querying a large stream of XML data poses a challenge to many stream-querying algorithms or applications because of the computational costs associated with this large volume. In many cases, synopsis data structures and statistics can be constructed from streams of XML data to summarize their structure and content, which are useful for a variety of applications. An important example application is query cost estimation: the problem is to provide accurate and efficient estimations of the query's cost in terms of space used and time spent.
Moreover, estimation is useful in itself to judge on the relevance of a query before running it, and is necessary for query optimization.

XML queries are often expressed as XPath expressions because of the tree-structured nature of XML data. Cost-based optimization for querying XML data streams requires calculating the cost of query operators. Usually the cost of an operator for a given XPath query depends heavily on the number of the final results returned by the query in question, and the number of temporary results that are buffered for its sub-queries [Zhang 2005]. Therefore accurate selectivity estimation is crucial for cost-based optimization.

Selectivity is a count of the number of matches for a query Q evaluated on an XML document D. This selectivity does not measure the size of these matches. Furthermore, it measures neither the total amount of memory allocated by the program to find these matches (space used) nor the processor time used by the program to find the matches (time spent). In addition, there are many parameters that influence streaming computational costs (as explained in chapter 1): the lazy vs eager strategy of the stack-automaton, the size and quantity of XPath query results which depend on the XPath query operator, the size and structure of the document etc. The author of an XPath query may have no immediate idea of what to expect in memory consumption and delay before collecting all the resulting sub-documents.

As a result, selectivity estimation appears necessary but incomplete as a technique for managing queries on large documents accessed as streams. We will therefore compute a synopsis data structure from the input XML document D. The purpose is to obtain a small but full structure summary that is traversed by an efficient streaming algorithm to accurately estimate the selectivity and/or to reduce the computational overhead of complex XPath queries on D.

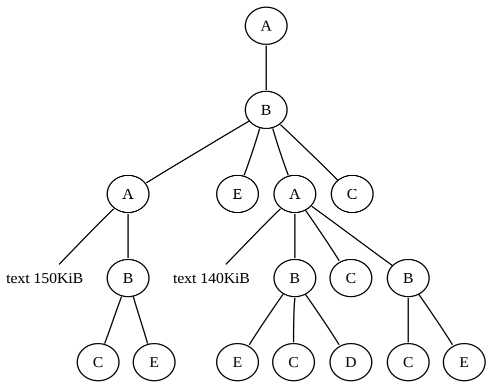
In this chapter, we present the *path tree*, a structure for XML-summarization that is used for accurate selectivity estimates, which was informally introduced by [Aboulnaga 2001] and used by [Zhang 2005]. To the best of our knowledge the path tree was not formally defined in the literature. We therefore, formally define it. Furthermore, we introduce two techniques to construct this synopsis structure, they are: one automaton technique and one streaming technique. Finally, we explain the incremental construction process and updating of the path tree.

3.1.1 The XML Data Model

Before defining the path tree we start by defining the XML document.
An XML document is modeled as a rooted, ordered, labeled tree, where each node corresponds to an element, attribute or a value, and the edges represent (direct) element-subelement or element-value relationships. An XML document, when passed through a SAX [Brownell 2002] parser, will generate a sequence of events. A streaming algorithm processes the SAX events, which are: $startElement(X,l)$ and $endElement(X)$. They are produced respectively when the opening or closing tag of a element is encountered and accept the name of the element X as input parameter. When a text value is encountered, the event $Text(value)$ is activated. The list l for $StartElement(X,l)$ represents the list of attributes for the element name

X.

Figure 3.1 illustrates an example of an XML document D and a snapshot of its SAX parser events.



(a) The XML Document D

```
StartDocument("Doc")
StartElement ("A", [("f","1") ; ("s","22") ] )
StartElement ("B", [("f","1") ; ("s","22") ] )

    ----
    ----
Text("text.....")
    ----
    ----

StartElement ("C", [("f","1") ; ("s","22")  ] )
EndElement ("C")
EndElement ("B")
EndElement ("A")
EndDocument("Doc")
```

(b) A snapshot for the SAX parser events of D

Figure 3.1: The XML Document D and a snapshot of its SAX parser events

Remarks:

- Unlike some (abstract syntax) trees, the number of subtrees at an XML node is not a priori bounded.
- The sequence of SAX events amounts to a leftmost depth-first traversal of the XML tree.

3.2 Path tree Definition

The path tree is a concise, accurate, and convenient summary of the structure of the XML document. To achieve conciseness, a path tree describes every distinct simple node-labeled path from the root of a source XML exactly once with its frequency (the number of times it appears). To ensure accuracy, the path tree does not contain node-labeled paths that do not appear in the source XML document. The structure is convenient because it can be processed by ordinary query evaluation algorithms (stream-querying/stream-filtering algorithms) in place of the actual document.

Given an XML document D, the path tree is (a tree with node labels taken from D) defined as follows:
Let $paths(D) = \{p = A_1, A_2, ...A_k \in \Sigma^*_{(D)} \mid p$ is a node-labeled path starting from

the root of D *i.e.* A_1 is the root$\}$.
Remark: all node-label paths in $path(D)$ have A_1 as a prefix.

Definition 1. *$PathTree(D)$ is a graph whose nodes are $p \in paths(D)$ and edges are the immediate prefix relation: $(path(D), \{(p_1, p_2) \mid \exists A \in \Sigma_{(D)}$ such that $p_2 = p_1 A$ and $p_i \in PathTree(D)\}$).*

Proposition 1. *$PathTree(D)$ is a tree rooted in root (D).*

Proof. $T_{prefix} = (\Sigma^*_{(D)}, \{(p_1, p_2) \mid \exists A \in \Sigma_{(D)}$ such that $p_2 = p_1 A\})$ is the Hasse-diagram of the prefix relation on $\Sigma^*_{(D)}$ and has a tree structure. By construction $PathTree(D)$ is a subgraph of T_{prefix} that is connected. Therefore, $PathTree(D)$ is also a tree. $\square$

Remarks:

- The root of D is the root of the path tree.
- Every path [1] in D also occurs in the *path tree*.
- Each node in the path tree is uniquely identified by its node-labeled path from the root. They can therefore be renamed with shorter identifiers than the paths p themselves.

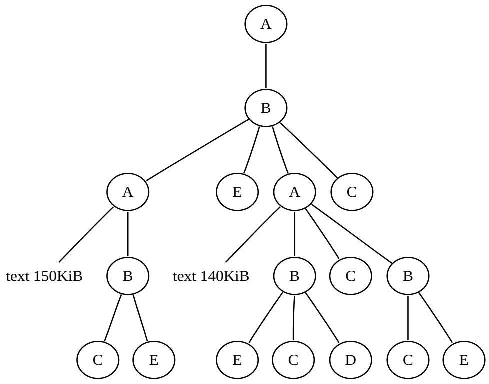
Figure 3.2 illustrates an example of an XML document D and its path tree. We use node numbering in the path tree to show the order of nodes, e.g., the nodes $A1$ and $A2$ have the same node-labels A, but $A1$ appears before $A2$. Also, in the path tree, the number in the bracket exist to the right of each node's label represents its frequency, e.g., $A2(2)$ indicates that the frequency of $A2$ is 2.

3.3 Path tree Construction: Automata Technique

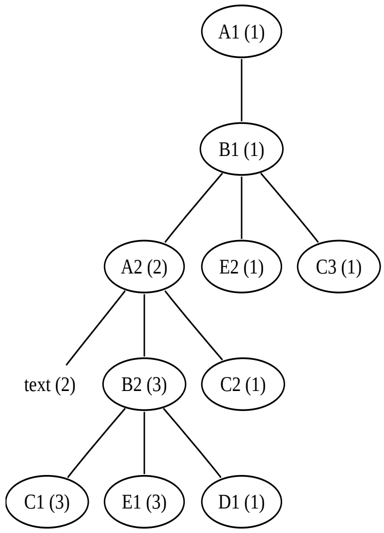
To create a path tree from an XML document D, we consider that D is equivalent to a DFA that we call $\mathbb{A}$ and its path tree is equal to a minimized DFA that we call $\mathbb{A}_{Min}$. For this purpose we use the automata minimizing algorithm (table-filling algorithm) [Hopcroft 1979].

Below, we explain in details the transformation process of an XML document D into its unique path tree.

[1] In this theory a path is a sequence of node labels, not to be confused with Dewey paths which are edge-label paths



(a) The XML Document D



(b) The path tree of D

Figure 3.2: The XML Document D and its path tree

3.3.1 Automaton Definition $\mathbb{A}$

We define the automaton associated with the XML document D as
$\mathbb{A}= (Q, \Sigma, \delta, Q_0, f)$ where:

- Q: the finite set of states. Their names are defined in the recursive definition below.
- Σ: the finite set of input symbols. Transition labels in the automaton are taken from node labels in D.
- $\delta : Q * \Sigma \rightarrow Q$. The transition function.
- $start_{(D)}$: without loss of generality we assume many initial or start states, $start_{(D)} \subseteq Q$.
- f: a final or accepting state, $f \in Q$.

For a document D consisting of one node, we construct its associated automaton $\mathbb{A}$ by using the function $\mathbb{A}ut()$ defined recursively on the tree strucutre of D as follows.

$\mathbb{A}ut(D) = (Q_{(D)}, \delta_{(D)}, \Sigma_{(D)}, start_{(D)}, f_{(D)})$, where (see figure 3.3):

- $Q_{(D)} = \{up_{(D)}, down_{(D)}, err_{(D)}\}$
- $\Sigma_{(D)} = \{root_{(D)}\}$
- $\delta_{(D)}(q, root_{(D)}) = \left\{ \begin{array}{l} up_{(D)} \mid q = down_{(D)} \\ err_{(D)} \mid q = err_{(D)} \text{ or } q = up_{(D)} \end{array} \right\}$

- $f_{(D)} = up_{(D)}$
- $start_{(D)} = down_{(D)}$

Here $root_{(D)}$ is the label of D's (unique, root) node.

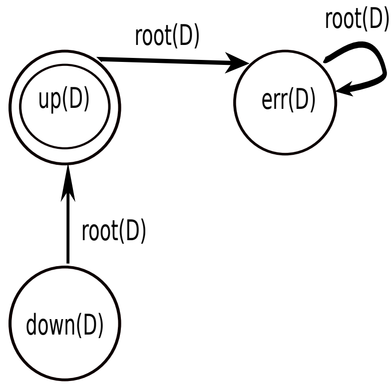


Figure 3.3: The associated automaton $\mathbb{A}$ of a one-node document D

By construction, the language $L(\mathbb{A}) = \{root_{(D)}\}$ the only node-labelled path in D. See below for the value of L in general.

Based on the above base case, we define the automaton associated to a general XML D (see figure 3.4) as follows:

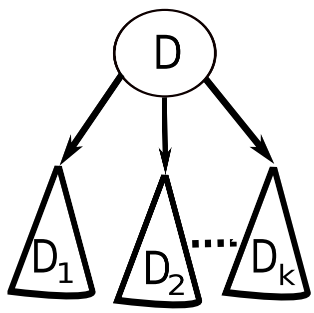


Figure 3.4: XML document D

$\mathbb{A}ut(D) = (Q_{(D)}, \delta_{(D)}, \Sigma_{(D)}, start_{(D)}, f_{(D)})$, where:

- $\Sigma_{(D)} = (\cup_{i=1}^{k} \Sigma_{(D_i)}) \cup \{root_{(D)}\}$
- $Q_{(D)} = (\cup_{i=1}^{k} Q_{(D_i)}) - (\cup_{i=1}^{k} \{up_{(D_i)}, err_{(D_i)}\}) \cup \{up_{(D)}, down_{(D)}, err_{(D)}\}$
- $\delta_{(D)}(q,a) = \left\{ \begin{array}{l} up_{(D)} \mid q = down_{(D)} \wedge a = root_{(D)} \\ \delta_{(D_i)}(q,a) \mid q \in Q_{(D_i)} \wedge \delta_{(D_i)}(q,a) \notin \{up_{(D_i)}, err_{(D_i)}\} \\ down_{(D)} \mid q \in Q_{(D_i)} \wedge \delta_{(D_i)}(q,a) = up_{(D_i)} \\ err_{(D)} \mid q = err_{(D)} \; or \; q = up_{(D)} \end{array} \right\}$
- $f_{(D)} = up_{(D)}$

- $start_{(D)} = \cup_{i=1}^{k}\ start_{(D_i)}$

Figure 3.5 shows an example of the transformation process of XML document D into its associated automaton $\mathbb{A}$ by using the function $\mathbb{A}ut(D)$.

$\mathbb{A}ut(\ )$ is applied recursively on the XML document D as follows: $\mathbb{A}ut(A)$, calls $\mathbb{A}ut(D1)$ and $\mathbb{A}ut(D2)$. The dashed arrows coming out from states $down(D1)$ and $down(D2)$ indicate that these states and transitions belong to $Q_{(D)}$.

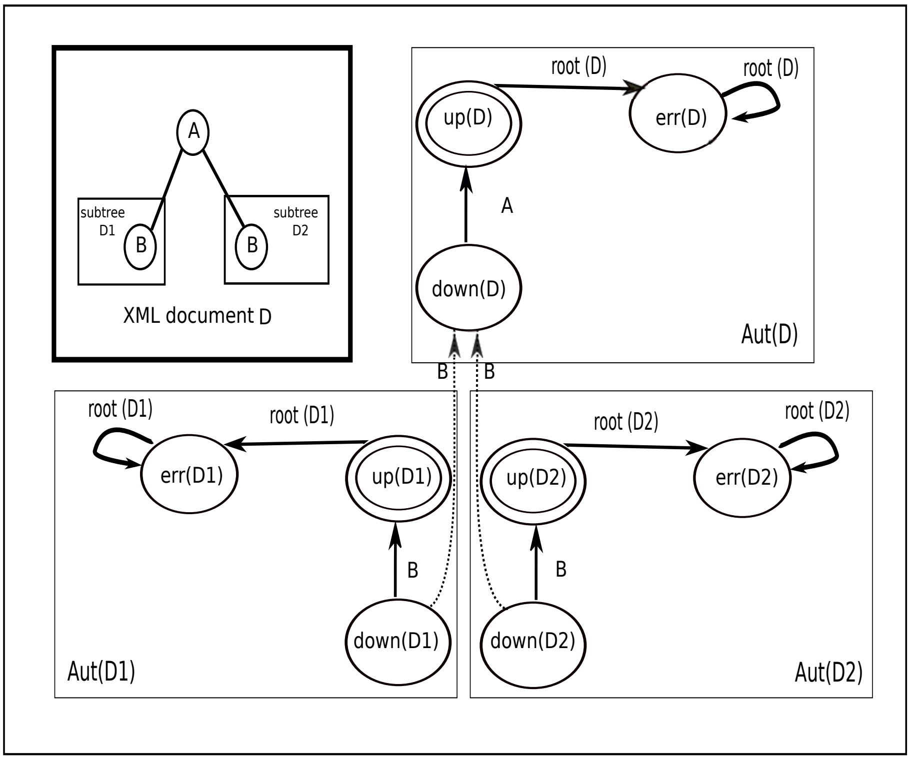


Figure 3.5: An XML document D and the construction process of its associated automaton $\mathbb{A}$

The final automaton associated to the XML document D is illustrated in figure 3.6.
The language of the automaton $\mathbb{A}$ is the union of all node-labeled paths from start states ($start_{(D)}$) to f, *i.e.* $L(\mathbb{A}) = \{w \in \Sigma^*_{(D)} \mid \delta^*_{(D)}(q_0, w) = f \wedge q_0 \in start_{(D)}\}$

Applying the function $\mathbb{A}ut(\ )$ on the document D requires removing specific states and their transitions (of the figure 3.5) from the final automaton of the document D that is illustrated in the figure 3.6. For example: states $err(D1)$, $err(D2)$ and their transitions were removed.

Another more general and complete example of this transformation process is explained in section 3.3.4.

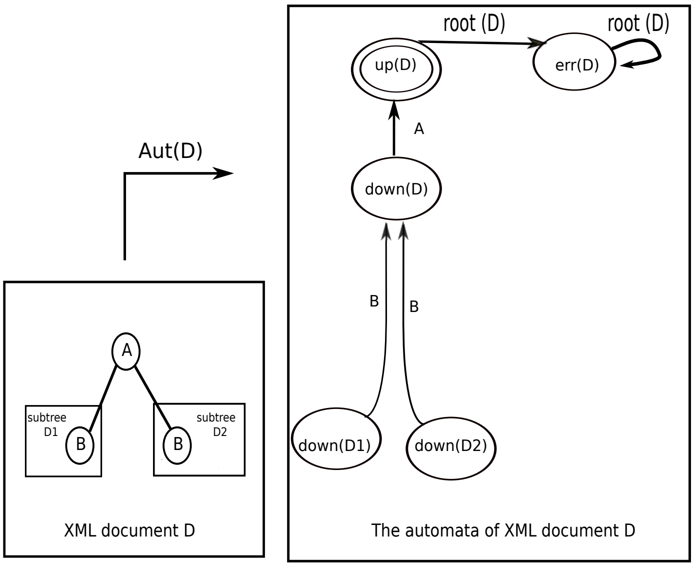


Figure 3.6: An XML document D and its associated automaton $\mathbb{A}$

3.3.2 Automata Transformation into a Graph $\mathbb{D}oc(\mathbb{A})$

The edge graph [Harary 1960] associated with a given graph is defined as follows. Given a graph G, its edge graph $L(G)$ is a graph such that:

- each vertex of $L(G)$ represents an edge of G; and
- two vertices of $L(G)$ are adjacent if and only if their corresponding edges share a common endpoint ("are adjacent") in G.

We now define the function $\mathbb{D}oc$ which inverts our DFA to an edge labeled graph like the original document.
Given the automaton associated to the XML document D as $\mathbb{A}$ (see figure 3.7), we transform $\mathbb{A}$ into D using the function $\mathbb{D}oc(\ )$ as follows:

$\mathbb{D}oc(\mathbb{A}) = (Nodes_{(\mathbb{A})}, Edges_{(\mathbb{A})})$, where:

- $Nodes_{(\mathbb{A})} = \{(q,E) \mid \ \delta(q,E) \neq err_{(D)} \ \wedge \ q \in Q_{(D)} \wedge \ E \in \Sigma_{(D)}\}$.

- $Edges_{(\mathbb{A})} = \{((q_2,E_2),(q_1,E_1)) \mid \ \delta(q_1,E_1) = q_2 \wedge \ q_1,q_2 \in Q_{(D)} \wedge \ E_1,E_2 \in \Sigma_{(D)}\}$.

As illustrated by figure 3.7, $Nodes_{(\mathbb{A})}$ are:
$(down_{(D)},A)$, $(down_{(D_1)},B)$ and $(down_{(D_2)},B)$. For simplicity we name them n_1, n_2 and n_3 respectively.

The edges of $Edges_{(\mathbb{A})}$ are:
$\Big((down_{(D)},A),\,(down_{(D_1)},B)\Big)$ and $\Big((down_{(D)},A),\,(down_{(D_2)},B)\Big)$, that is equal to (n_1,n_2) and (n_1,n_3). For simplicity we call them E_1 and E_2 respectively.

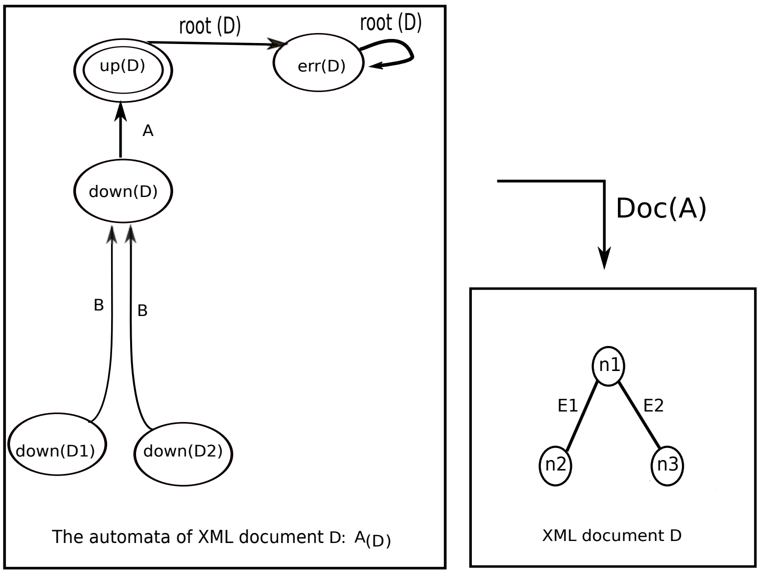


Figure 3.7: The transformation process of $\mathbb{A}$ into D

The result of $\mathbb{D}oc(\mathbb{A})$ is the XML document illustrated in figure 3.7.

3.3.3 Automata Minimization $\mathbb{A}_{Min}$

Automata theory defines that two states q and p are equivalent if: for all input words w, $\delta(q,w)$ is an accepting state if and only if $\delta(p,w)$ is an accepting state. Otherwise, the they are called distinguishable [Hopcroft 1979].

To compute states that are equivalent, we find pairs of states that are distinguishable. Any pairs of states that we do not find distinguishable are equivalent according to the table-filling algorithm. It attempts discovery of distinguishable pairs in the automaton until none are found.

Below, we explain how to minimize the automaton associated with the XML document D by using the automata minimizing algorithm [Hopcroft 1979].

The minimization algorithm for a given $\mathbb{A} = (Q_{(D)}, \delta_{(D)}, \Sigma_{(D)}, start_{(D)}, f_{(D)})$ denoted by $Min(\mathbb{A})$ is:

1. Initialize a boolean matrix of all unordered pairs of states of $\mathbb{A}$ by setting all entries to false. This table represents pairs of states known to be distinguishable. The initialization is $M[p,q] = false$, $\forall(p,q)$, i.e the algorithm initially

assumes that all states are equivalent (or non-distinguishable). The algorithm proceeds by accumulating evidence that proves certain pairs of states to be distinguishable.

2. For every pair $(p, f_{(D)})$ where $p \neq f_{(D)}$, mark $(p, f_{(D)})$ to be distinguishable (and vice versa). These are states which can not be equivalent.
   $M[p, f_{(D)}] = true$.

3. For each unmarked pair (p,q) and $a \in \Sigma_{(D)}$ if $(\delta_{(D)}(p,a), \delta_{(D)}(q,a))$ is marked, then mark (p,q). $M[p,q] = true$.

4. Repeat 3 until there are no changes.

5. Combine states: for each unmarked (p,q) such that $p \neq q$ which means $(M[p,q] = false)$ then

   - For any state $s \in Q_{(D)}$ such that $q = \delta_{(D)}(s,a)$ then
     remove $q = \delta_{(D)}(s,a)$.
     add $p = \delta_{(D)}(s,a)$
   - For any state $s \in Q_{(D)}$ such that $s = \delta_{(D)}(q,a)$ then
     remove $s = \delta_{(D)}(q,a)$
     (for all $p \in Q_{(D)}$ and $a \in \Sigma_{(D)}$ *i.e.* remove q and all transactions leading to and from q).

6. The algorithm's output is the minimized Automaton $\mathbb{A}_{Min}$

If $\mathbb{A}$ is a DFA and $Min(\mathbb{A})$ constructed from $\mathbb{A}$ by the automata minimizing algorithm, then $Min(\mathbb{A})$ has as few states as any DFA equivalent to $\mathbb{A}$ [Hopcroft 1979], and moreover $L(Min(\mathbb{A})) = L(\mathbb{A})$.

The complexity of this DFA minimizing algorithm is quadratic $O(n^2)$. An $O(n\ log\ n)$ algorithm for DFA minimizing was introduced in [Hopcroft 1971]. But to the best of our knowledge, there is no a streaming algorithm which minimizes the DFA in $O(n)$. Therefore, in the section 3.4, we present a streaming algorithm to create the path tree synopsis in linear time.

3.3.4 Example of Path tree Construction: Automata Technique

The construction process of the path tree from an XML document D is summarized as follows (see figure 3.8): (1) transforming D into its associated automaton $\mathbb{A}$ by using $\mathbb{A}ut(D)$. (2) minimizing $\mathbb{A}$ by using the automata minimization algorithm $Min(\mathbb{A})$. (3) transforming the minimized automaton $\mathbb{A}_{Min}$ into its path tree $PathTree$ by using $\mathbb{D}oc(\mathbb{A}_{Min})$.

Below we present and explain a complete example of this process.

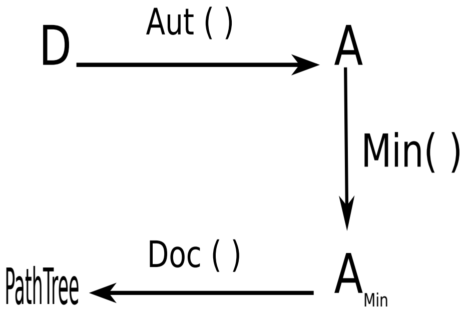


Figure 3.8: The construction steps of the path tree - automata technique

1. Transforming D into its associated automaton $\mathbb{A}$:

   Figure 3.9 represents the XML document D.

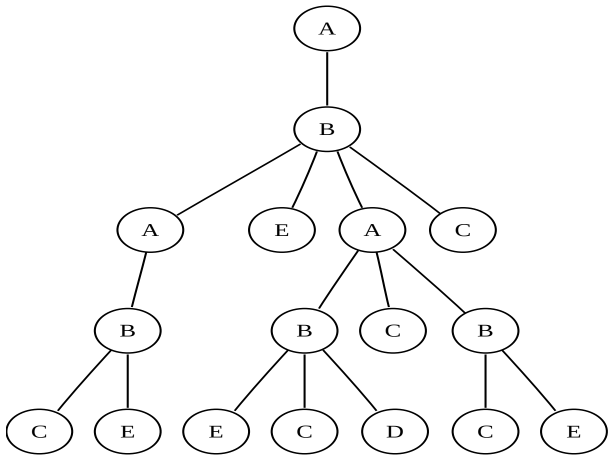
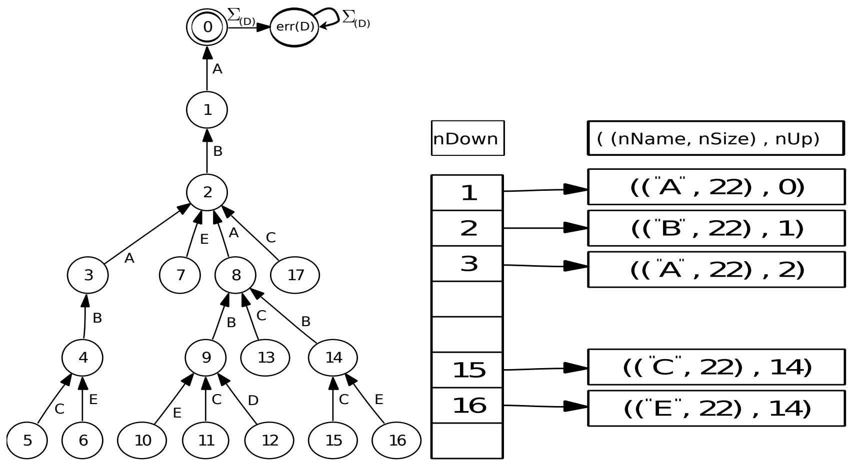


Figure 3.9: The XML document D

With respect to the automaton definition in section 3.3.1, we recursively transform D into its associated automaton $\mathbb{A}$ by using the function $\mathbb{A}ut(D)$. A hash table is used to store $\mathbb{A}$. Figure 3.10 represents the automaton associated to D and part of its hash table.
We start by explaining the structure of this hash table. *nName*: is the label of the node, where $nName \in \Sigma_{(D)}$. *nDown* and nUp: are counters for naming the states in the automaton (e.g. 1, 2, ...etc.). Their initialized values $= 0$. Note that $\delta(nDown, nName) = nUp$. *nSize*: is the size in byte of *nName*. These fields are illustrated in figure 3.14.

In figure 3.10(a), the state 0 is by default the final state of $\mathbb{A}$, it is the final state for the node-label A which is the root of D (see figure 3.9).
As it is mentioned in section 3.3.1: without loss of generality we assume many initial or start states for $\mathbb{A}$. These states are *nDown* states for the leaf nodes in D.



(a) $\mathbb{A}$: the associated automaton of D (b) A part of the hash table of $\mathbb{A}$

Figure 3.10: $\mathbb{A}(D)$ and its hash table

Next, we show how to minimize $\mathbb{A}$ by using the automata minimization algorithm ($Min(\mathbb{A})$).

2. Minimizing $\mathbb{A}$ by using the automata minimization algorithm ($Min(\mathbb{A})$):

Minimizing $\mathbb{A}$ requires finding all its final equivalent states, then combining them. To achieve this purpose, we use the automata minimizing algorithm that is explained in details in section 3.3.3. The algorithm's output is the minimized automaton $\mathbb{A}_{Min}$.

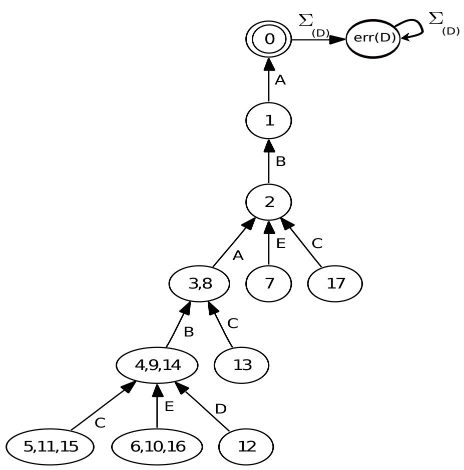


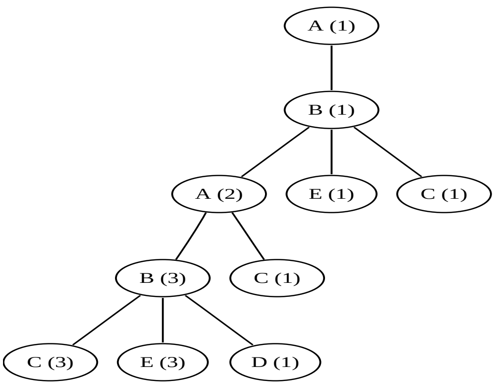
Figure 3.11: The minimized automaton $\mathbb{A}_{Min}$ of $\mathbb{A}$ in figure 3.10(a)

Figure 3.11 illustrates the minimized automaton computed from figure 3.10(a). In this figure, we see the combined equivalent states, for example:

stares 3 and 8 are combined together because they are equivalent.

3. Transforming the minimized automaton $\mathbb{A}_{Min}$ into its path tree by using $\mathbb{D}oc(\mathbb{A}_{Min})$:

   The path tree is a graph (as we defined in section 3.2), and the minimized automaton has the form of an inverted graph. Therefore the transformation process of the minimized automaton into its path tree is straightforward by using the function $\mathbb{D}oc$ (defined in section 3.3.2). In our example, figure 3.12(a) illustrates the result of this transformation process ($\mathbb{D}oc(\mathbb{A}_{Min})$) which generates the path tree of $\mathbb{A}_{Min}$. The number to the right of each node-label represents its frequency.



(a) The path tree of $\mathbb{A}_{Min}$

```
StartDocument("Doc")
StartElement ("A", [("f","1") ; ("s","22") ] )
StartElement ("B", [("f","1") ; ("s","22") ] )
StartElement ("A", [("f","2") ; ("s","44")  ] )
StartElement ("B", [("f","3") ; ("s","66")  ] )
StartElement ("E", [("f","3") ; ("s","66")  ] )
EndElement ("E")
StartElement ("C", [("f","3") ; ("s","66")  ] )
EndElement ("C")
StartElement ("D", [("f","1") ; ("s","22")  ] )
EndElement ("D")
EndElement ("B")
StartElement ("C", [("f","1") ; ("s","22")  ] )
EndElement ("C")
EndElement ("A")
StartElement ("E", [("f","1") ; ("s","22")  ] )
EndElement ("E")
StartElement ("C", [("f","1") ; ("s","22")  ] )
EndElement ("C")
EndElement ("B")
EndElement ("A")
EndDocument("Doc")
```

(b) The SAX parser events of the path tree

Figure 3.12: The path tree of $\mathbb{A}_{Min}$ and its SAX parser events.

Figure 3.12(b) represents the SAX parser events of the path tree. The list of attributes l for each $StartElement(nName, l)$ contains two attributes: f which is the frequency of $nName$ and s which is the size in byte of $nName$.

The SAX parser events of the path tree are used by our selectivity estimation algorithm to predict the computation cost (time/memory) for a given XPath query. Detailed explanation about the selectivity estimation can be found in the coming chapter (chapter 4).

3.4 Path tree Construction: Streaming Technique

Creating a path tree using the automata technique (as we explained in previous section 3.3) can be done by creating the DFA completely then applying the automata minimizing algorithm (table-filling algorithm) [Hopcroft 1979]. This approach is practical for small XML documents, while it might cause a memory bottleneck once it is used to generate a path tree for a large XML document, because the whole document should be buffered to generate the automaton associated to the XML document.
An elegant solution for this problem is to generate the minimized automaton associated to the XML document directly while streaming through it .
In this section, we propose a streaming algorithm to construct the path tree for very large XML documents. Moreover, we explain how to cover update transitions with the updating process of the path tree.

3.4.1 Path tree Construction

We propose a streaming algorithm which takes as input the SAX parser events of *D* and creates directly its minimized automaton. Figure 3.13 show the steps for constructing the path tree using the streaming approach.

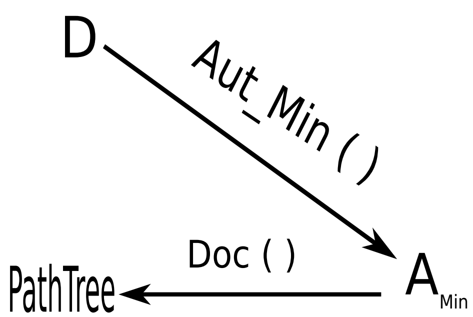


Figure 3.13: The construction steps of the path tree- Streaming technique

We explain our algorithm through the example below.

Example of path tree construction: the minimized automaton is illustrated in figure 3.14 (**autoTable**). We start by explaining the structure of this table. *nName*: is the label of the node, where $nName \in \Sigma_{(D)}$. *depth*: is the node's depth in *D*. *nDown* and *nUp*: are counters for naming the states in the automaton (e.g. 1, 2, ...etc.). Their initialized values = 0. Note that $\delta(nDown, nName) = nUp$. *nFreq*: is the frequency of *nName* in *D* which have the same node-labeled path. *nSize*: is the size in byte of *nName* in *D* which have the same node-labeled path.

In our algorithm, a stack named *pathStack* is used to store the node-labeled path during the construction process of the path tree. At each SAX event

$StartElement(nName)$, *pathStack* is pushed with (*nName*, *nDown)*, and at each $EndElement(nName)$, the top of *pathStack* is popped out.

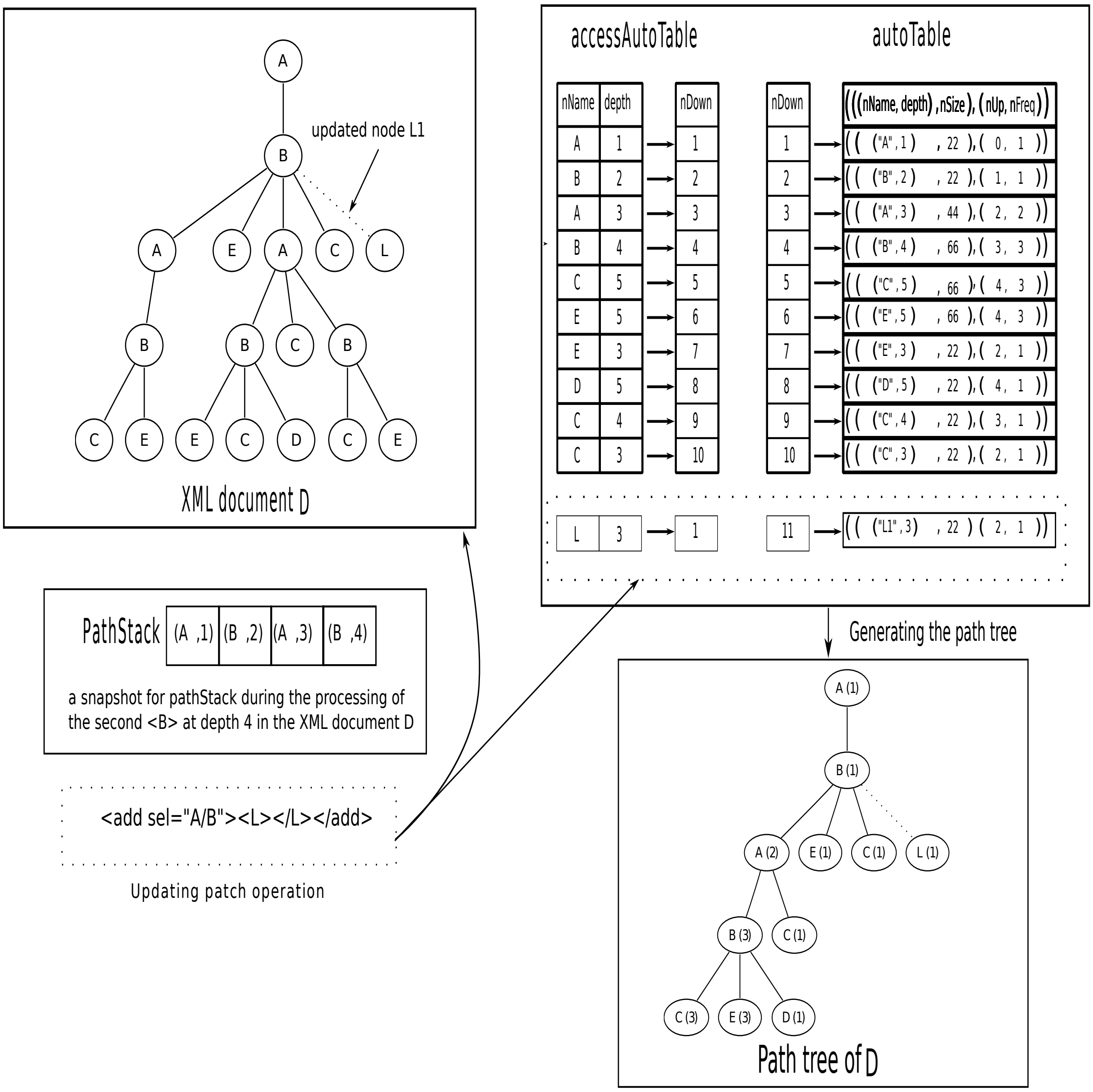


Figure 3.14: Path tree: construction and updating

When $< A >$ the root of D is read, $depth = 1$ (algorithm 1 line 1) then, we add A with its information to *accessAutoTable*, *autoTable* and *pathStack* (algorithm 1 lines $2-6$).

Note that *nUp* of A=0. When $< B >$ with $depth = 2$ is read, the function *checkSameNodePath* is called (algorithm 1 line 9). As long as B is not yet a member of *accessAutoTable* (algorithm 2 line 1), then we add B with its information to *accessAutoTable*, *autoTable* and *pathStack* (algorithm 2 lines $21-27$).

Algorithm 1: createAutoTable (depth, nName, nSize)

```
1  if (depth=1) then
2      nDown ← nDown + 1 nFreqStack = [1]//initializethearraywithnFreq = 1
3      nSizeStack= [nSize]// initialize the array with nSize (node Size)
4      addNodeKey (depth, nName, nDown) // add a new node to accessAutoTable
5      addNode (nDown, nName, depth, nSizeStack, nUp, nFreqStack) // add a new node to autoTable. Note that
       nUp=0 ;
6      pushPathStack (depth, nName, nDown) //update the pathStack;
7  else
8      checkSameNodePath (depth, nName, nSize) ;
```

Algorithm 2: checkSameNodePath (depth, nName, nSize)

```
1  if (isMemeber accessAutoTable (depth, nName)) then
2      l= get the list of all nDown in accessAutoTable which have the same key (depth,nName) ;
3      let nodePathExist = false ;
4      foreach nDown ∈ l do
5          nodenUp= get nUp of nDown from autoTable ;
6          nodenDownPathStack=get nDown of (depth-1) from pathStack ;
7          if (nodenUp= nodenDownPathStack) then
8              nodePathExist= true ;
9              augmentFrequeny (nFreqStack) // augment the nFreq of nName by 1 ;
10             augmentSize (nSizeStack, nSize) // augment the value in nSizeStack by nSize ;
11             pushPathStack(depth, nName, nDown) //update the pathStack ;

12     if (isNodePathExist = false) then
13         nDown ← (nDown) + 1
14         nDownPathStack=get nDown of (depth-1) from pathStack ;
15         nFreqStack= [1] // initialize the array with nFreq =1 ;
16         nSizeStack= [nSize]// initialize the array with nSize (node Size);
17         addNodeKey (depth, nName, nDown) // add a new node to accessAutoTable;
18         addNode (nDown, nName, depth, nSizeStack, nDownPathStack, nFreqStack) // add a new node to
           autoTable. Note that nUp=nDownPathStack ;
19         pushPathStack (depth, nName, nDown) //update the pathStack ;

20 else
21     nDown ← (nDown) + 1
22     nDownPathStack=get nDown of (depth-1) from pathStack ;
23     nFreqStack= [1] // initialize the array with nFreq =1 ;
24     nSizeStack= [nSize]// initialize the array with nSize (node Size);
25     addNodeKey (depth, nName, nDown) // add a new node to accessAutoTable;
26     addNode (nDown, nName, depth, nSizeStack, nDownPathStack, nFreqStack) // add a new node to autoTable.
       Note that nUp=nDownPathStack ;
27     pushPathStack (depth, nName, nDown) //update the pathStack ;
```

When the second $< B >$ with $depth = 4$ is read, B is already a member of *accessAutoTable* (algorithm 2 line 1), therefore, we check whether the node-labeled path of the received B exists or not in *autoTable* (algorithm 2 lines $2-19$). The value of nUp for B with $depth = 4$ (which is already exist in *autoTable*) is 3 (see algorithm 2 line 5 and *autoTable* of figure 3.14). Also, in *pathStach* the value *nDown* for the parent ($depth-1$) of the received B is 3 (see algorithm 2 line 6 and *pathStack* of figure 3.14), both values are equals because the parents of both *nName* B have the same node-labeled path, which mean both *nName* B also have the same node-labeled path. Therefore, we increment the frequency and size of B (see algorithm 2 lines $8-11$). If the node-labeled path of B was not exist in *autoTable* (see algorithm 2 line 12), then node B with its information is added (see algorithm 2 lines $13-19$). The moment $< /A >$ (EndElement of the root) is processed, the complete path tree can be generated and output in SAX events syntax.

The construction process of the path tree is incremental, it allows constructing different incomplete path trees before the construction of the complete one. An *incomplete path tree* is a partial path tree constructed for a part of an XML document.

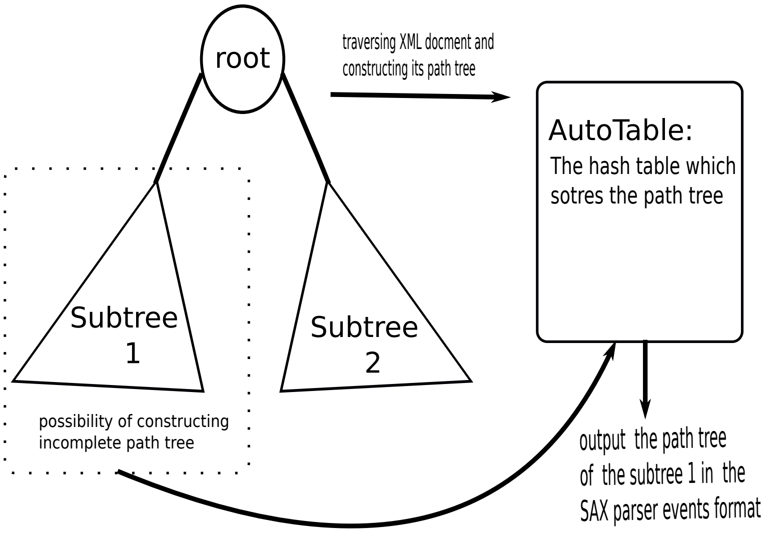


Figure 3.15: Example on the incremental construction of the path tree

Figure 3.15 illustrates an example on the incremental construction process of the path tree. As it is shown in the figure, the XML document is received in streaming mode and the path tree is incremental built and stored in the hash table *AutoTable*. The moment we finish of streaming the first subtree (*i.e.* we encountered the closing tag of the root of the subtree 1), it is possible to output the incomplete path tree of this subtree in the SAX parser events based on the needs of the application concerned.

Our streaming algorithm has time complexity $O(|depth(D)|.|D|)$ and space complexity $O(|depth(D)|.|pathTree(D)|)$ because the minimized automaton is buffered in the RAM.

3.4.2 Path tree Updating

When the underlying XML document is updated, *i.e.* some elements are added or deleted, the path tree can be incrementally updated using XML patch operations [Urpalainen 2008].

We explain this procedure by a short example below:

- Adding an element: Figure 3.14 shown an example of a patch operation to update the XML document D. This operation adds an empty element L as a last child under "A/B" where element A is the root of D. The same patch will be sent to the path tree (*accessAutoTable* and *autoTable*) for updating. Thus, we check whether the node-labeled path of L that is ABL exists or not in *autoTable*.

  In this example, it is not, therefore, we add the new node L with its information to *accessAutoTable* and *autoTable* (see figure 3.14). Otherwise (node-labeled path of L is exist), the frequency and the size of node L will be updated as we shown in algorithm 2 (lines 7-11).

In this section, we provided a general idea about the possibility of updating our path tree synopsis. The information about the path tree structure facilitates its updating process. Actually, minimal synopsis size seems desirable but won′t be the best because incremental maintenance would be difficult [Goldman 1997]. This is the case of both TreeSketch [Polyzotis 2004a] and XSeed [Zhang 2006b]. While in our approach, incremental update is possible by using the patch operations as we explained above.

A complete updating algorithm is under study. Our first impression that the updating algorithm resembles the path tree creation algorithm with slight modification. This point was already confirmed in [Goldman 1997]. Their incremental algorithm turns out to be only a slightly modified version of their DataGuide creation algorithm.

In this chapter, we presented the *path tree*, a structure for XML-summarization that is used for accurate selectivity estimates. To the best of our knowledge the path tree was not formally defined in the literature. We therefore, formally defined it. Furthermore, we introduced two techniques to construct this synopsis structure, they are: one automaton technique and one streaming technique. Finally, we explained the incremental construction process and updating of the path tree.

In the next chapter 4, we present our selectivity estimation technique which uses the path tree structure synopsis and our selectivity estimation algorithm (that is inspired from the lazy stream-querying algorithm LQ [Gou 2007]) to estimate the selectivity for any XPath query which belongs to the fragment of Forward XPath.

CHAPTER 4

Selectivity Estimation Techniques

Contents

4.1 Introduction

Developing performance prediction models for query optimization is significantly harder for XML queries than for traditional relational queries. The reason is that XML query operators are more complex than relational operators such as table scans and joins. Moreover, the query evaluation process of XML data streams raises extra challenges compared to non-streaming environments: the recursive nature of XML documents, the single sequential forward scan of a stream of XML data, also the presence of descendant axes, the predicates, and the wildcard nodes in the XPath query.

Selectivity estimation is an estimate of the number of matches for a query Q evaluated on an XML document D. It is desirable in interactive and internet applications. With it, the system could warn the end user for example that his/her query is so coarse that the amount of results will be overwhelming.

This selectivity does not measure the size of these matches. Furthermore, it measures neither the total amount of memory allocated by the program to find these matches (space used) nor the processor time used by the program to find the matches (time spent). As a result, selectivity estimation appears necessary but incomplete as a technique for managing queries on large documents accessed as streams and it is not sufficient to model the query cost.

To estimate the cost for a given XPath query Q evaluated (in streaming mode) on an XML document D, we need to estimate precisely the parameters which determine the cost of Q. The parameters we use are as the following:

1. *NumberOfMatches*: is the number of answer elements found during processing of the XPath query Q on the XML document D.

2. *Cache*: is the number of elements cached in the run-time stacks during processing of the XPath query Q on the XML document D. They correspond to the axis nodes of Q.

3. *Buffer*: is the number of potential answer elements buffered during processing of the XPath query Q on the XML document D.

4. *OutputSize*: is the total size in MiB of the number of answer elements found during processing of the XPath query Q on the XML document D.

5. *WorkingSpace*: is the total size in MiB for the number of elements cached in the run-time stacks and the number of potential answer elements buffered during processing of the XPath query Q on the XML document D.

6. *NumberOfPredEvaluation*: is the number of times the query's predicates are evaluated (their values are changed or passed from an element to another).

A precise performance prediction model needs a selectivity estimation technique to measure accurately the values of the parameters above mentioned in order to estimate/predict the cost for a given query.

In chapter 2 (state of the art), we presented the different techniques of selectivity estimation. Moreover, we justified the need for a new technique. In this chapter, we present our selectivity estimation technique that is used to estimate the values of the parameters that determine the cost for a given XPath query. More precisely, these parameters are above six.

The selectivity estimation technique consists of:

1. The path tree structure synopsis (which we defined in detail in chapter 3): a concise, accurate, and convenient summary of the structure of the XML document.

2. Selectivity estimation algorithm: an efficient streaming algorithm used to traverse the path tree synopsis for estimating the values of the parameters which determine the cost of a given XPath query.

The remainder of the chapter is structured as follows.

In section 4.2, we explain our extension for the lazy stream-querying algorithm LQ which was introduced by [Gou 2007]. The extended algorithm processes the fragment of Forward XPath (we defined this fragment in section 1.1.1.2). Furthermore, in the same section, we present several examples (by using several XPath queries) on the stream-querying process to explain the behavior of our extended algorithm LQ. All examples use the same XML document D, but in each example we use a different XPath query Q and we measure the values of the parameters that determine the cost for Q.

In section 4.3, we present our selectivity estimation algorithm which we inspired from our extended lazy stream-querying algorithm LQ (of section 4.2). This algorithm is used to traverse the path tree synopsis for estimating the values of the cost parameters. In the same section, we explain how our selectivity estimation technique (the path tree synopsis plus the selectivity estimation algorithm) functions. We explain that through several examples. All examples use the path tree synopsis of the XML D above mentioned (used in the examples of the stream-querying process). Moreover, the queries used in these examples are the same queries used in the examples of the stream-querying process (of section 4.2). In each example, we estimate the values of the parameters that determine the cost of the given XPath query.
Finally, we measure the accuracy of our selectivity estimation technique. We compare the values measured of the stream-querying process with the estimated ones on examples of the selectivity estimation technique.

4.2 Lazy Stream-querying Algorithm

The lazy stream-querying algorithm LQ was introduced by [Gou 2007] to prove that univariate XPath can be efficiently evaluated in $O(|D|.|Q|)$ time in the streaming environment. This algorithm does not process XPath queries that contain the following: $attributes$, $'text()'$, nested predicates, and predicates with ($'and'$, $'or'$, $'not'$). Therefore, we extended LQ to processes the Forward XPath fragment (defined in chapter 1) with the same complexity.

The current extended version of LQ processes queries that belong to the fragment of Forward XPath. Our algorithm was implemented using the functional language OCaml release 3.11 [Leroy 2010b] which combines relatively high performance with strong typing and ML-language constructs for tree processing.

Our extended LQ takes two input parameters (see figure 4.1). The first one is the XPath query (which belongs to Forward XPath to allow stream-processing) that will be transformed to a query table statically using our Forward XPath Parser. After that, the main function is called. It reads the second parameter (XML document in SAX parser events) line by line repeatedly, each time generating a

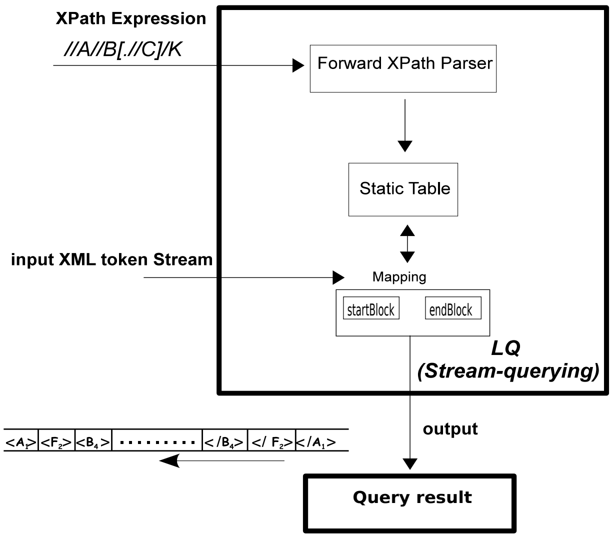


Figure 4.1: Extended LQ (Lazy stream-querying)

tag. Based on that tag a corresponding *startBlock* or *endBlock* function is called to process it. Finally, the main function generates as output the result for the sent XPath query. The result is the measured values of the cost parameters already defined in section 4.1.

We begin this section by introducing the extension process of the query-preprocessing phase (section 4.2.1). In section 4.2.2 we explain our extension of the LQ algorithm (the main functions *startBlock* and *endBlock*). Then, in section 4.2.3, we present several examples on stream-querying process by using our LQ extended.

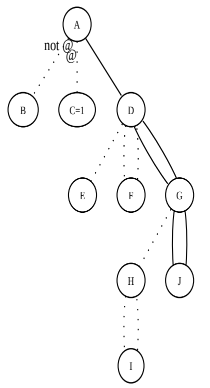
4.2.1 Query Preprocessing

The query table illustrated in figure 4.2(b) is statically stored in the memory throughout stream processing. Each column in the table of the XPath query Q corresponds to a node of Q. A virtual node represents the column number 0, this node is the parent of the XPath query's root node.

To access any column of the XPath query table we implemented a hash table over all (*nName*, *nNumber*) pairs, thus we can retrieve the *nNumber* of any query node by giving the *nName* of that node.

Each node in the XPath query table has specific fields. Our contribution is adding new fields which allow the processing of more complex queries, for example queries which contain predicates with *'not'* operator. The new fields are *p_Children_List*, *att_Name_List*, *att_Value_List*, *bool_Op_List*, and *arith_Op_List*. The fields of each node in the XPath query table are explained below:

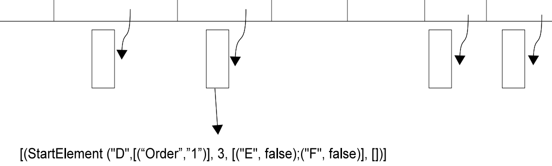
XPath: //A[not(@B) and @C=1]/D[./E or .//F]//G[./H[.//I]]//J



(a) XPath query

hash table (nName, nNumber)

nNumber	0	1	2	3	4	5	6	7	8
nName	vir	A	D	E	F	G	H	I	J
axis	-	//	/	/	//	//	/	//	//
type	-	Axis	Axis	Predicate	Predicate	Axis	Predicate	Predicate	Result
parent	-	0	1	2	2	2	5	6	5
host	-	2	2	-	-	5	-	-	-
p_Children_List	-	-	[3;4]	-	-	[6]	[7]	-	-
att_Name_List	-	[B;C]	-	-	-	-	-	-	-
att_Value_List	-	[Nul;1]	-	-	-	-	-	-	-
bool_Op_List	-	[and;not;non]	[or;non;non]	-	-	-	-	-	-
arith_Op_List	-	[Null;Null;Equal]	-	-	-	-	-	-	-
leaf	-	false	false	true	true	false	false	true	true
stack									



(b) Query Table

Figure 4.2: XPath transformation into a query table

1. *axis*: represents the axis of the node. It can be a child $'/'$ or a descendant $'//'$.

2. *type*: represents the node types which are: Axis, Predicate, Result.

3. *parent*: represent the *nNumber* of the parent node of the cn (context node).

4. host: in stream-querying we partition the main path of the XPath query into multiple segments by removing all descendant axes. For each segment there is a host node that is at the tail of that segment. Notice that (1) only axis nodes can be host nodes, (2) the segment which include the result node has no host node. For example: the main path of the XPath query in figure 4.2(b) is $//A/D//G//J$. We partition the main path into three segments $//A/D$, $//G$ and $//J$. In this case the host of node A is the node D ($host[A] = D$), the host node of D is the node D itself ($host[D] = D$), the host node of G is the node G itself ($host[G] = G$), while for the segment $//J$, node J has no host node because it is the result node.

5. *p_Children_List*: represents the list of all predicate nodes (with axis $'/'$ or $'//'$) children of the cn.

6. *att_Name_List*: represents the list of all attribute names of the cn. In figure 4.2(b), the attributes names of node A are B and C.

7. *att_Value_List*: represents the list of all attributes values of the cn. The values order in this list corresponds to the attribute names order in *att_Name_List*. For example: in figure 4.2(b), the attribute B is not associated with any value,

this why we associate to the value ($Null$), while attribute C is associated with the value 1.

8. *bool_Op_List*: represents the list of all boolean operators associated with *p_Children_List* and *att_Name_List* of the cn. For example: the *bool_Op_List* of A is $[and; not, non]$. The first element of this list is $'and'$, it indicates that A has two conditions. It returns a value of true if both its operands (conditions) are true, and false otherwise. The second element is $'not'$, it is a negation of the first condition of A, which is $not(@B)$. The third element is $'non'$, it indicates that the second condition is not associated with a boolean operator, that is $@C$ in our example.

9. *arith_Op_List*: represents the list of all arithmetic operators associated with *att_Name_List* of the cn. In our example: the *arith_Op_List* of A is $[Null; Null; Equal]$. The third element of this list is $'Equal'$, it indicates that the value of the third element of *att_Name_List* is associated with the value of the third element of *att_Value_List*, which in our example $C = 1$.

10. Leaf: to know whether cn is a leaf node or not. This field is a boolean value.

11. stack: for each non-leaf query node, a run-time stack is created. The OCaml structure of this stack is typed:
    *(token*int*(string*bool)list*token list)list*

    An example of stack's content of the node D in figure 4.2(b) is:
    $\Big[\Big($ StartElement("D",[("Order","1")]), 3, [("E",false); ("F", false)], [] $\Big)\Big]$.

    In this example, $StartElement(\text{"}D\text{"}, [(\text{"}Order\text{"}, \text{"}1\text{"})])$ is an element name D which has the attribute "*Order*" with the value "1". The integer 3 represents the depth of the element D in the XML document. The list $[(\text{"}E\text{"}, false); (\text{"}F\text{"}, false)]$ is the predicate list of the node D. The $false$ value means that so far there is no match between the parent node D and its predicate nodes E and F. The moment a match for a predicate node (e.g., the node C) is found, its $false$ value will be changed to $true$. The last list that is called potential answers list is to buffer or append the potential answer nodes during the evaluation process of the XPath query.

4.2.2 LQ - Blocks Extension

After the transformation of the XPath query into a query table (query preprocessing) as we explained in section 4.2.1, the main function in LQ will be called. It reads the XML document (in SAX parser events) line by line repeatedly, each time generating a tag. Based on that tag a corresponding *startBlock* or *endBlock* function is called to process it.

Algorithm 3: startBlock ((*nName*,*l*), *nNumber*, *depth*)

```
if (parent stack of the node is not empty) then
    if (node type ≠ Predicate) or (Predicate's value is still false) then
        if (node axis =Descendant) or (node axis = Child) then
            if (node = leaf) then
                if node type = Predicate) then
                    evaluate the predicate node ;
                else
                    if (node type =Result) then
                        if (node is the query's root) then
                            output answers ;
                        else
                            buffer and append the node ;
            else
                push stack: (nName,l) , depth, list of the predicates, an empty list for potential answers  /*l
                in (nName,l) indicates the list of attributes of the context node.  */
                ;
```

The algorithms 3, 4, 5 and 6 represent the pseudo code of the main functions (*startBlock* and *endBlock*) of our extended LQ (Lazy stream-querying algorithm). These algorithms will be explained through different examples in section 4.2.3.

Algorithm 4: endBlock (*nName*, *nNumber*, *depth*)

```
if (node ≠ leaf) || (node's stack is empty) then
    let s =get the top of the node's stack ;
    if (node's depth = current depth) then
        pop out the node from its stack;
        if (node's stack is not empty) then
            check and update the predicates with descendant axis;
        let bool_Op_List= get the boolean operators associated with predicate children of the node
        match (head bool_Op_List ) with
        | Not → if(the negation is true )then
            processNodeType nNumber s ;
                                                        /*the algorithm 5 */
        else
            appendOrDestroy nNumber s ;
                                                        /*the algorithm 6 */
        | And → if(all predicates are matched)then
            processNodeType nNumber s ;
                    /*if the predicate does not contain a boolean operator, it will be
            processed as And.  */
        else
            appendOrDestroy nNumber s
        | Or → if(one predicate is matched )then
            processNodeType nNumber s
        else
            appendOrDestroy nNumber s
        | Non → if(node has no predicate )then
            processNodeType nNumber s
```

Algorithm 5: processNodeType (*nNumber*, *s*)

```
if (node type = Axis) then
    if (node is the query's root) then
        let potential_answers_list = the list of the potential answers nodes of the current node
        if  (potential_answers_list of the current node is not empty) then
            output the content of potential_answers_list: answers ;
    else
        if (potential_answers_list of the current node is not empty) then
            append potential_answers_list to the same list of the parent of the current node
else
    if (node type =Predicate) then
        check and update the predicate
        if (node axis = Descendant) then
            clear the predicate's stack
    else
        if (node type =Result) then
            if (node is the query's root) then
                output answers
            else
                append node to the potential answers list of the node's parent
```

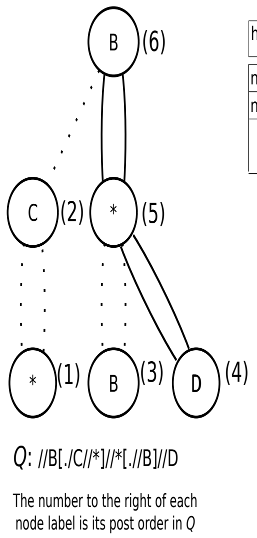
Algorithm 6: appendOrDestroy (*nNumber*, *s*)

if *(node type = Axis)* **then**
  if *the stack of the host node of the current node is empty* **then**
    destroy s ;
  else
    append the list of the potential answers of the current node to the same list of the top node of the host stack (the host stack of the current node) ;

Before presenting several examples on XPath query processing by using LQ, we will explain how LQ calls the functions *startBlock* and *endBlock* to process efficiently the wildcard nodes and the same node-labels. After that, we explain how the attributes are processed eagerly by LQ.

- **The processing of the wildcard nodes and same node-labels**

  To process queries with wildcard nodes and the same node-labels, the XPath query preprocessing (query table) still creates one column for each query node. But each *nName* in the hash table corresponds to a sequence of column numbers (*nNumber*) whose corresponding query nodes either have that *nName* or are wildcard nodes, see figure 4.3.



hash table (nName, nNumber)

nNumber	0	1	2	3	4	5	6
nName	vir	B	C	*	*	B	D
.....	.....	.....	.....	.....	.....	.....	.....

nName	Post order sequence of nName	nNumber sequence of nName
*	1, 5	3, 4
B	1, 3, 5, 6	3, 5, 4,1
C	1, 2, 5	3, 2, 4
D	1, 4, 5	3, 6, 4

Open tag: the order of calling the function startBlock

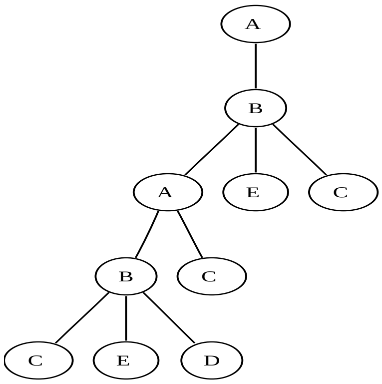
nName	Pre order sequence of the post order	nNumber sequence of nName
*	1, 5	4, 3
B	6, 1, 5,3	1, 3, 4, 5
C	2, 1, 5	2, 3, 4
D	1, 5, 4	3, 4, 6

Close tag: the order of calling the function endBlock

Figure 4.3: The XPath query Q, and the sequence of calling *startBlock* and *endBlock*

A special sequence for the column numbers (*nNumber*) of all wildcard nodes is also created. All nodes in column sequences follow a special order (post-order), such that each node must not have any of its ancestor nodes in front of itself.

During reception of the stream of XML data, for each open tag (where the element name belongs to the query's nodes) , for example: $< B >$, the function *startBlock* is called iteratively according to the post-order of *nName* B in Q, that is 1, 3, 5, 6. The equivalent *nNumber* order of B is 3, 5, 4, 1 (see the hash table). And, for each close tag $< /B >$, the function *endBlock* is called



(a) XML document D

```
StartDocument("Doc")
StartElement ("A", [("f","1") ; ("s","22") ] )
StartElement ("B", [("f","1") ; ("s","22") ] )

 - - - -
 - - - -
 - - - -
 - - - -
 - - - -

StartElement ("C", [("f","1") ; ("s","22")  ] )
EndElement ("C")
EndElement ("B")
EndElement ("A")
EndDocument("Doc")
```

(b) A snapshot of the SAX parser events of D

Figure 4.4: The XML D and a snapshot of its SAX parser events

iteratively in the pre-order of *nName* B in Q, that is 6, 1, 5, 3. The equivalent *nNumber* order of B is 1, 3, 4, 5 (see the hash table).

Further explanation of the processing of the wildcard nodes and the same node-labels can be found in [Gou 2007].

- **Attributes processing**

  After the transformation of the XPath query into a query table (query preprocessing) as we explained in section 4.2.1, the main function in LQ will be called. It reads the XML document (in SAX parser events) line by line repeatedly, each time generating a tag. Based on that tag a corresponding *startBlock* or *endBlock* function is called to process it. Figure 4.4 illustrates an XML document D and a snapshot of its SAX parser events.

  In our lazy stream-querying algorithm, we process attributes eagerly. This means, the moment the main function generates $StartElement(e,l)$ (where e is the element name and l is the list of attributes of the element), attributes (if needed) will be evaluated first, and according to the result of this evaluation, the function *startBlock* might be called, or the main function will generate a new tag. The advantage of processing the attributes eagerly is to avoid buffering or caching unnecessary elements as we explain in the example below.
  Given the XPath query Q as $//A[@f \ and \ @kk]//B$ and the XML document D as in figure 4.4, the moment we receive the event $StartElement(”A”, [(”f”, ”1”); (”s”, ”22”)]$ (see the SAX parser event of D), we evaluate first the predicate of node A in Q, as long as the predicate condition is not satisfied, the function *startBlock* will not be called, this means that the element A will not be pushed in its corresponding stack.

In the next section, we present several examples on the stream-querying process by using our LQ extended.

4.2.3 Examples of Query Processing Using LQ-Extended

In this section, we present several examples on the XPath query evaluation process by using our lazy stream-querying algorithm.
We first present examples on the XPath query processing by using simple path, then by using twig path (simple path and twig path are defined in chapter 1). The results for each example are important and will be used in the next section 4.3 (Selectivity Estimation Algorithm) to compare the measured and estimated results for each examples.
We number the nodes of the XML document that is used to explain the examples of this section. The purpose of this numbering is to show the node order in the XML document. For example: for a given document D which has two nodes $A1$ and $A2$, both nodes have the same node-labels, but node $A2$ appears after the node $A1$ according to the pre-order traversal of D.

4.2.3.1 Query Processing - Simple Path

We first start by showing how LQ processes a simple path p, where p does not contain same node-labels neither wildcard nodes. After that, we adapt to cases where p contains same node-labels and wildcard nodes.

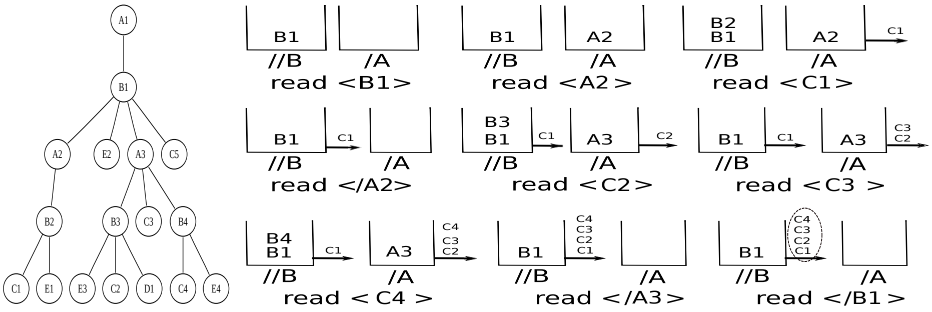
- **Simple path without wildcard nodes or same node-labels**:

  Figure 4.5 illustrates the XML document D and snapshots of the run-time stacks for the evaluation process of D on the simple path $//B/A//C$ which returns the sequences $C1$, $C2$,$C3$ and $C4$ as answers. For each non-leaf node, the algorithm creates a stack. Therefore, in this example, a stack is created for the root node B and another one for the node A.

  When $< A1 >$ is read, the parent stack (stack B) of $A1$ is empty, therefore, $A1$ is discarded (algorithm 3 line 1). When $< B1 >$ is read, it is pushed in its corresponding stack B (algorithm 3 line 14). When $< A2 >$ is read, its parent stack (stack B) is not empty, therefore, $A2$ is pushed in its corresponding stack B (algorithm 3 line 14).

  When $< C1 >$ is read, as long as its parent stack is not empty, $C1$ is buffered to the potential answers list of its parent node $A2$ (algorithm 3 line 12). Note that node $B2$ was already pushed in its corresponding stack.

  When $< /C1 >$ is read, $C1$ is a leaf node, this is why LQ proceed by processing the next SAX event (algorithm 4 line 1). When $< E1 >$ is read, $E1$ is not a member of the query's nodes, therefore, it will be discarded immediately.



(a) XML document D (b) Snapshots of the run-time stacks for the evaluation of D on Q

Figure 4.5: Snapshots of the run-time stacks for the evaluation of D on Q ($//B/A//C$)

When $< /A2 >$ is read, it is popped out from its stack and its potential answers list is appended to the list of $B1$ (algorithm 4 lines 21-22 then algorithm 5 lines 7-8).

When $< C2 >$ is read, note that nodes $B3$ and $A3$ were already pushed in their corresponding stacks. As long as the parent stack of $C2$ is not empty, $C2$ is buffered to the potential answers list of its parent node $A3$. Same thing for $< C3 >$ and $< C4 >$.

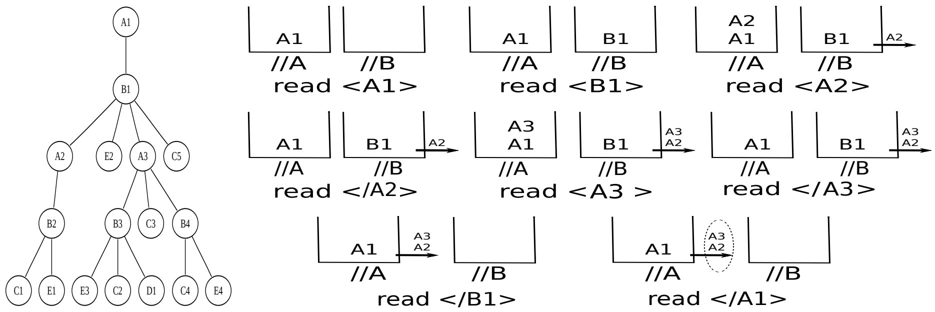
When $< /A3 >$ is read, it is popped out from its stack and its potential answers list is appended to the list of $B1$. When $< /B1 >$ is read, $B1$ is the root node of the query, therefore, the content of its potential answers list is flushed as answers (algorithm 4 lines 21-22 then algorithm 5 lines 2-5). Finally, when $< C5 >$ is read, thought $C5$ is a potential answer, but it is discarded because its parent stack (stack A) is empty.

The result of the XPath query evaluation is as follows (measured values): $NumberOfMatches$: the value is 4, they are: $C1$, $C2$, $C3$ and $C4$. $Buffer$: for any simple path, the value of $Buffer$ is the same as the value of $NumberOfMatches$. $Cache$: the value is 6, they are: $B1$, $B2$, $B3$, $B4$ and $A2$, $A3$. $WoringSpace$: its size was measured to 0.0002MiB. $OutputSize$: its size was measured to 0.00008MiB.

- **Simple path with same node-labels**:

Figure 4.6 illustrates different snapshots for the evaluation process of D on the simple path $//A/B//A$, which returns $A2$ and $A3$ as results nodes.
For each non-leaf node, the algorithm creates a stack. Therefore, in this example, a stack is created for the root node A and another one for the node B.

When $< A1 >$ is read, the function $startBlock$ is called in the post order of



(a) XML document D (b) A snapshot of the run-time stacks for the evaluation of D on Q

Figure 4.6: Snapshots of the run-time stacks for the evaluation of D on Q ($//A//B//A$)

A in Q, that is 3,1. The node $A1$ with order 3 is not buffered because its parent stack (stack B) is empty (algorithm 3 line 1). While $A1$ with order 1 is pushed in its corresponding stack A (algorithm 3 line 14). When $< B1 >$ is read, it is pushed in its corresponding stack B (algorithm 3 line 14).

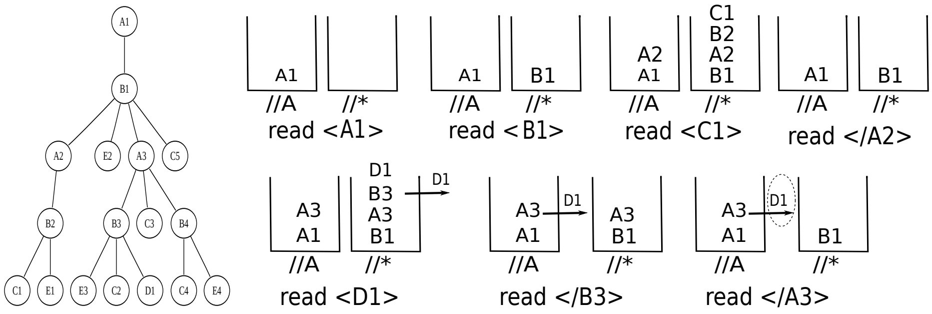
When $< A2 >$ is read, the function *startBlock* is called in the post-order of A in Q, that is 3,1. The $A2$ with order 3 is buffered to the potential answers list of node $B1$ (algorithm 3 line 12). While $A1$ with order 1 is pushed in its corresponding stack A.

When $< /A2 >$ is read, the function *endBlock* is called in the pre-order of A in Q that is 1, 3. $A2$ with order 1 is popped out from its stack (algorithm 4 line 4 then algorithm 5 lines 6-8). While for $A2$ with order 3, the algorithm will proceed by processing the next SAX event of D, because $A2$ is a leaf node (algorithm 4 line 1).

When $< A3 >$ is read, the function *startBlock* is called in the post-order of A in Q, that is 3,1. The node $A3$ with order 3 is buffered to the potential answers list of node $B1$ (algorithm 3 line 12). While $A1$ with order 1 is pushed in its corresponding stack A (algorithm 3 line 14).

When $< /A3 >$ is read, it is processed in the same manner as $< /A2 >$. When $< /B1 >$ is read, it is popped out from its stack and its potential answers list is appended to the same list of its parent node $A1$ (algorithm 4 lines 21-22 then algorithm 5 lines 7-8). Finally when $< /A1 >$ is read, $A1$ is the root node of the query, therefore, the content of its potential answers list is flushed as answers (algorithm 4 lines 21-22 then algorithm 5 lines 2-5).

The result of the XPath query evaluation is as follows (measured values): $NumberOfMatches$: the value is 2, they are: $A2$ and $A3$. $Buffer$: for any simple path, the value of $Buffer$ is the same as the value of $NumberOfMatches$. *Cache*: the value is 7, they are: $B1$, $B2$, $B3$, $B4$ and $A1$, $A2$, $A3$. *WoringSpace*: its size was measured to 0.0002MiB. *OutputSize*: its size was measured to 0.00004MiB.



(a) XML document D (b) Snapshots of the run-time stacks for the evaluation of D on Q

Figure 4.7: Snapshots of the run-time stacks for the evaluation of D on Q ($//A//*//D$)

- **Simple path with a wildcard**:

Figure 4.7(b) illustrates different snapshots of the evaluation process of D on the simple path $//A//*//D$ which returns $D1$ returns as a result node. For each non-leaf node, the algorithm creates a stack. Therefore, in this example, a stack is created for the root node A and another one for the node $*$.

When $< A1 >$ is read, the function *startBlock* will is called in the post-order of A in Q, that is 2,1. The node $A1$ with order 2 is not cached because its parent stack (stack A) is empty (algorithm 3 line 1). While $A1$ with order 1 is pushed in its corresponding stack A (algorithm 3 line 14).

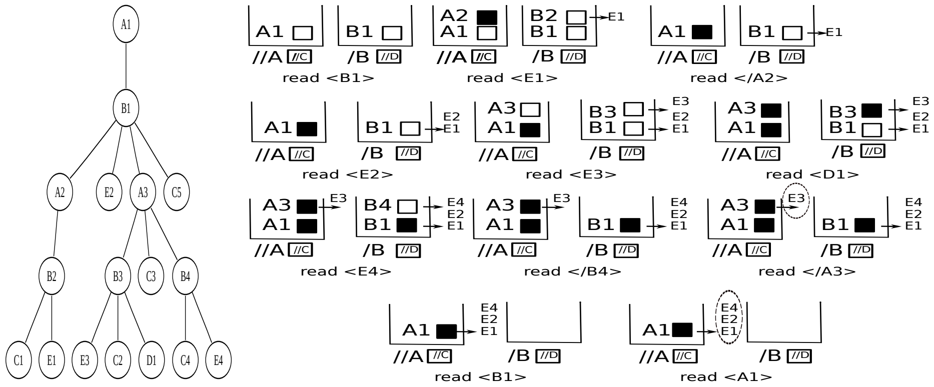
When $< B1 >$ is read, it is pushed in its corresponding stack $*$ (algorithm 3 line 14). When $< C1 >$ is read, it is pushed in its corresponding stack $*$ (algorithm 3 line 14).

When $< /A2 >$ is read, the function *endBlock* is called in the pre-order of A in Q that is 1, 2. $A2$ with order 1 is popped out from its stack (algorithm 4 line 4 then algorithm 5 lines 6-8). The same thing for $A2$ with order 2, $A2$ is popped out from stack $*$.

When $< D1 >$ is read, the function *startBlock* is called in the post-order of A in Q, that is 3, 2. The node $D1$ with order 3 is buffered to the potential answers list of its parent node $B3$ (algorithm 3 line 12). While $D1$ with order 2 is pushed in its corresponding stack $*$ (algorithm 3 line 14).

When $< /B3 >$ is read, it is popped out from its stack (stack $*$) and its potential answers list is appended to the same list of its parent node $A3$ (algorithm 4 lines 21-22 then algorithm 5 lines 7-8).

When $< /A3 >$ is read, $A3$ is the root node of the query, therefore, the content of its potential answers list is flushed as answers (algorithm 4 lines 21-22 then algorithm 5 lines 2-5).



(a) XML document D (b) Snapshots of the run-time stacks for the evaluation of D on Q

Figure 4.8: Snapshots of the run-time stacks for the evaluation of D on Q ($//A[.//C]/B[.//D]//E$)

The result of the XPath query evaluation is as follows (measured values): $NumberOfMatches$: the value is 1, it is $D1$. $Buffer$: for any simple path, the value of $Buffer$ is the same as the value of $NumberOfMatches$. $Cache$: the value is 19, we present them based on their stacks as follows: stack A contains $A1$, $A2$, and $A3$. While stack $*$ contains $B1$, $A2$, $B2$, $C1$, $E1$, $E2$, $A3$, $B3$, $E3$, $C2$, $D1$, $C3$, $B4$, $C4$, $E4$ and $C5$. $WoringSpace$: its size was measured to 0.0004MiB. $OutputSize$: its size was measured to 0.00002MiB.

In the next section, we present several examples on the stream-querying process by using our LQ extended and twig paths.

4.2.3.2 Query Processing - Twig Path

Below, we present two examples on twig path processing by using our stream-querying algorithm LQ.

- **Twig path with multi predicates**:

Figure 4.8(b) illustrates different snapshots of the evaluation process of D on the twig path $//A[.//C]/B[.//D]//E$ which returns $E1$, $E2$, $E3$ and $E4$ as result nodes. For each non-leaf node, the algorithm creates a stack. Therefore, in this example, a stack is created for the root node A and another one for the node B.

When $< B1 >$ is read, it is pushed in its corresponding stack B (algorithm 3 line 14). Note that, the node $A1$ was already pushed in its corresponding stack.

When $< E1 >$ is read, the node $A2$ and $B2$ were read and already pushed in their corresponding stacks. Moreover, the node $C1$ was read, it is a descendant of $A2$, so the value of the predicate C for $A2$ was changed from false to true (the black rectangle in the figure for this event) (algorithm 3 lines 4-6). Concerning $E1$, it is buffered to the potential answers list of its parent node $B2$ (algorithm 3 line 12).

When $< /A2 >$ is read, the node $B2$ was already popped out from its stack, as long as the predicate of $B2$ (that is D) was not satisfied (algorithm 4 lines 5-6), the function $appendOrDestroy$ was called (algorithm 4 line 16). The host stack of B is the stack B itself ($host[B] = B$, for further information see our definition of the host node in section 4.2.1), as long as this stack was not empty (it contained node $B1$), the potential answers list of $B2$ was appended to the same list of $B1$. Concerning the node $A2$, it is popped out from its stack, but as long as its predicate condition (node C) is satisfied and this predicate node has a descendant axis, then, the predicate value of $A1$ is changed from false to true (algorithm 4 line 5-6). Note that $A1$ is an ancestor of $A2$ with the same node-labels.

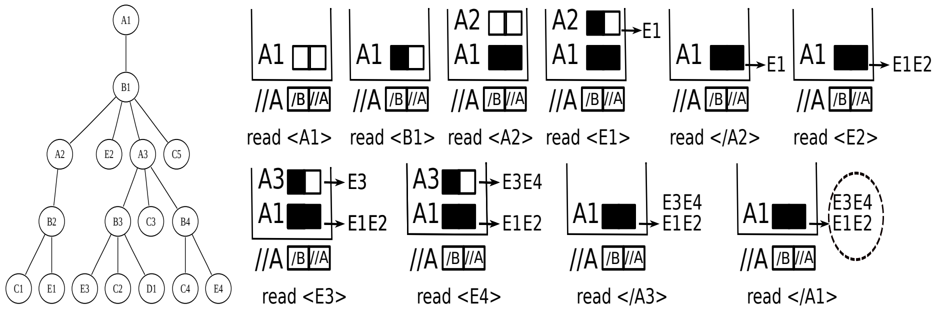
When $< E2 >$ is read, it is buffered to the potential answers list of its parent $B1$ (algorithm 3 line 2).

When $< E3 >$ is read, the nodes $A3$ and $B3$ were read and already pushed in their corresponding stacks. Concerning $E3$, it is buffered to the potential answers list of its parent node $B3$ (algorithm 3 line 12).

When $< D1 >$ is read, the node $C1$ was read, it is a descendant of $A3$, so the value of the predicate C for $A3$ was changed from false to true (algorithm 3 lines 4-6). Concerning $D1$, it is descendant node of $B3$, therefore, the value of the predicate D for $B3$ is changed from false to true.

When $< E4 >$ is read, the node $B3$ was popped out from its stack, and as long as its predicate condition was satisfied, then, its potential answers list was appended to the same list of its parent node $A3$. In addition, $B3's$ predicate condition (node C) was satisfied and this predicate node has a descendant axis, therefore, the predicate value of $B1$ was changed from false to true (algorithm 4 line 5-6). The node $B4$ was pushed in its corresponding stack. Concerning $E4$, it is buffered to the potential answers list of its parent node $B4$.

When $< /B4 >$ is read, node $B4$ is popped out from its stack, as long as the predicate of $B4$ (that is D) is not satisfied (algorithm 4 lines 5-6), the function $appendOrDestroy$ is called (algorithm 4 line 16), as long as this stack is not empty (it contains node $B1$), the potential answers list of $B4$ is appended to the same list of $B1$.



(a) XML document D (b) Snapshots of the run-time stacks for the evaluation of D on Q

Figure 4.9: Snapshots of the run-time stacks for the evaluation of D on Q ($//A[./B\ and\ .//A]//E$)

When $< /A3 >$ is read, $A3$ is the root node of the query, therefore, the content of its potential answers list is flushed as answers (algorithm 4 lines 21-22 then algorithm 5 lines 2-5).

When $< /B1 >$ is read, it is popped out from its stack and its potential answers list is appended to the list of $A1$ (algorithm 4 lines 21-22 then algorithm 5 lines 7-8).

Finally When $< /A1 >$ is read, $A1$ is the root node of the query, therefore, the content of its potential answers list is flushed as answers (algorithm 4 lines 21-22 then algorithm 5 lines 2-5).

The result of the XPath query evaluation is as follows (measured values): $NumberOfMatches$: the value is 4, they are: $E1$, $E2$, $E3$ and $E4$. $Buffer$: in this example, the value of $Buffer$ is the same as the value of $NumberOfMatches$. $Cache$: the value is 7, we present them based on their stack as follows: stack A contains $A1$, $A2$, and $A3$. While stack B contains $B1$, $B2$, $B3$ and $B4$,

$WoringSpace$: its size was measured to 0.0002MiB. $OutputSize$: its size was measured to 0.00008MiB. $NumberOfPredEvaluation$: its value is 6. This value represents the number of times the values of the predicate nodes (C and D) were changed or passed from an element to another during the query evaluation process.

- **Twig path with *and* operator and same node-labels**:

Figure 4.9(b) illustrates different snapshots of the evaluation process of D on the twig path $//A[./B\ and\ .//A]//E$ which returns $E1$, $E2$, $E3$ and $E4$ as result nodes. For each non-leaf node, the algorithm creates a stack. Therefore, in this example, a stack is created for the root node A.

When $< A1 >$ is read, the function $startBlock$ is called in the post-order of A in Q, that is 3,1. The predicate node $A1$ with order 3 is not evaluated because its parent stack (stack A) is empty (algorithm 3 line 1). While $A1$ with order 1 is pushed in its corresponding stack A with false values for its both predicate nodes B and A (algorithm 3 line 14).

When $< B1 >$ is read, $B1$ is a direct child for the node $A1$, therefore its value is changed from false to true.

When $< A2 >$ is read, the function *startBlock* is called in the post-order of A in Q, that is 3,1. The value of the predicate node A with order 3 for $A1$ is changed to true because its parent stack (stack A) is not empty, it contains the node $A1$ (algorithm 3 line 6). While $A2$ with order 1 is pushed in its corresponding stack A with false values for its both predicate nodes B and A (algorithm 3 line 14).

When $< E1 >$ is read, the node $B2$ was already read, therefore, the value of the predicate node B of $A2$ was changed from false to true. Concerning $E1$, as long as it is a descendant of $A2$, so it is buffered to the potential answers list of $A2$.

When $< /A2 >$ is read, it is popped out from its stack. $A2$ is the root node, but its predicate node A is not satisfied (algorithm 4 line 13), therefore, the function *appendOrDestroy* is called (algorithm 4 line 16). The host stack of A is the stack A itself ($host[A] = A$), as long as this stack is not empty (it contains node $A1$), the potential answers list of $A2$ is appended to the same list of $A1$.

When $< E2 >$ is read, $E2$ is a descendant of $A1$, therefore $E2$ is buffered to the potential answers list of $A1$.

When $< E3 >$ is read, the node $A3$ was read and processed in the same manner of the node $A2$. Concerning $E3$, it is a descendant of $A3$, therefore $E3$ is buffered to the potential answers list of $A3$. Same thing for the node $E4$, it is buffered to the potential answers list of $A3$.

When $< /A3 >$ is read, it is popped out from its stack. $A3$ is the root node, but its predicate node A is not satisfied (algorithm 4 line 13), therefore, the function *appendOrDestroy* is called (algorithm 4 line 16). The host stack of A is the stack A itself ($host[A] = A$), as long as this stack is not empty (it contains node $A1$), the potential answers list of $A3$ is appended to the same list of $A1$.

Finally, when $< /A1 >$ is read, it is popped out from the stack. $A1$ is the root node, as long as the values of its predicates B and A are true, then, the content of its potential answers list ($E1$, $E2$, $E3$ and $E4$) is flushed as a final answer.

The result of the XPath query evaluation is as follows (measured values): $NumberOfMatches$: the value is 4, they are $E1$, $E2$, $E3$ and $E4$. $Buffer$ the value is 4 , they are $E1$, $E2$, $E3$ and $E4$. *Cache*: the value is 3, they are $A1$, $A2$, $A3$. *WoringSpace*: its size was calculated to 0.0001 MiB. $OutputSize$: its size was calculated to 0.00008 MiB. $NumberOfPredEvaluation$: its value is 4. This value represents the number of times the values of the predicate nodes (B and A) were changed or passed from an element to another during the query evaluation process.

In the next section, we introduce our selectivity estimation technique, To measure the accuracy of this technique, we estimate the selectivity for same XPath queries which were used in the section 4.2.3. Then, we compare the measured and the estimated values for these queries.

4.3 Selectivity Estimation Algorithm

Selectivity estimation predicts the values of cost parameters that we defined in section 4.1. These parameters are: *NumberOfMatches*, *Buffer*, *OutputSize*, *WorkingSpace*, and *NumberOfPredEvaluation*.

To enable the selectivity estimation process, we inspired our selectivity estimation algorithm from the extended LQ (lazy stream-querying algorithm) that we defined in section 4.2.
The current version of our estimation algorithm processes queries which belong to the fragment of Forward XPath.

The estimation algorithm takes two input parameters. The first one is the XPath query (which respects Forward XPath [Alrammal 2009a] to allow stream-processing) that will be transformed to a query table statically using our Forward XPath Parser. After that, the main function is called. It reads the second parameter (the path tree that is defined in chapter 2) line by line repeatedly, each time generating a tag. Based on that tag a corresponding $startBlock$ or $endBlock$ function is called to process it. Finally, the main function generates as output the selectivity estimation result (estimated values) for the sent XPath query.

The algorithms 7, 8, 9 and 10 represent the pseudo code of the main functions ($startBlock$ and $endBlock$) of our selectivity estimation algorithm. The pseudo

Algorithm 7: startBlock (nName, nFreq, nSize, depth)

```
if (parent stack of nNumber is not empty) then
    if (node type ≠ Predicate) or (Predicate's value is still false) then
        if (node axis =Descendant) or (node axis = Child) then
            if (node = leaf) then
                if node type = Predicate) then
                    evaluate the predicate node and increase the predicate evaluation's counter
                    (predCounter) by the value of nFreq ;
                else
                    if (node type =Result) then
                        if (node is the query's root) then
                            if (nName= text()) then
                                calculate: NumberOfMatches, OutputSize   /*Here we do not
                                output the real value of the text node, in stead, we
                                output its real nSize and its nFreq */
                                output answers
                            else
                                calculate: Buffer, WorkingSpace
                                buffer and append the node to the potential answers list of parent of
                                the current node
            else
                calculate: Cache, WorkingSpace ;
                push stack: nName, depth, list of the predicates, an empty list for the potential answers,
                nFreq, nSize   /*the size and the frequency of nName are pushed as well.
                */ ;
```

code and the selectivity estimation process are explained through several examples in section 4.3.1.

4.3.1 Examples of the Selectivity Estimation Process

In this section, we present several examples on the selectivity estimation process by using our selectivity estimation algorithm.
We first present examples on the selectivity estimation by using simple paths, then by using twig paths. The results (estimated values) of the examples are important and will be compared with the result (measured values) of the same examples which we introduced in section 4.2.3.

Figure 4.10, illustrates the XML document D, the path tree of D and the SAX parser events of the path tree. We use node numbering in the path tree to show the order of nodes, e.g., the nodes $A1$ and $A2$ have the same node-labels A, but $A1$ appears before $A2$. Also, in the path tree, the number in the bracket exist to the right of each node's label represents its frequency (*nFreq*), e.g., $A2(2)$ indicates that the frequency of $A2$ is 2. In the SAX parser events of the path tree, the list of attributes l for each $StartElement(e,l)$ contains two attributes: *nFreq* which is the frequency of e and *nSize* which is the size in byte of e. These attributes are important for the selectivity estimation process.

Algorithm 8: endBlock ($nName$, $nNumber$, $depth$)

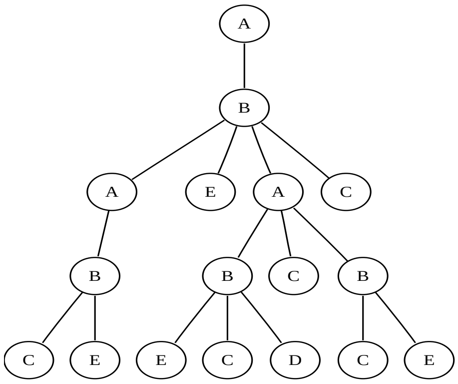
```
if (node ≠ leaf) || (node's stack is empty) then
    let s =get the top of the node's stack ;
    if (node's depth = current depth) then
        pop out the node ;
        if (node's stack is not empty) then
            check and update the predicates with descendant axis. If predicate node has a descendant axis,
            then increase predCounter by 1;
        let bool_Op_List= get the boolean operators associated with predicate children of the node ;
        match (head bool_Op_List ) with
        | Not → if(the negation is true )then
            processNodeType nNumber s ;
                                                            /*the algorithm 5 */
        else
            appendOrDestroy nNumber s ;
                                                            /*the algorithm 6 */
        | And → if(all predicates are matched)then
            processNodeType nNumber s ;
                    /*if the predicate does not contain a boolean operator, it will be
            processed as And.  */
        else
            appendOrDestroy nNumber s
        | Or → if(one predicate is matched )then
            processNodeType nNumber s
        else
            appendOrDestroy nNumber s
        | Non → if(node has no predicate )then
            processNodeType nNumber s
```

Algorithm 9: processNodeType ($nNumber$, s)

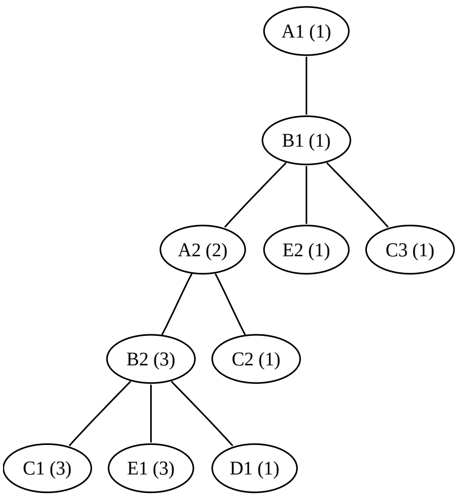
```
if (node type = Axis) then
    if (node is the query's root) then
        let potential_answer_list = the list of the potential answers nodes of the current node
        if  (potential_answers_list of the current node is not empty) then
            calculate: NumberOfMatches, OutputSize ;
            output the content of potential_answers_list: answers ;
    else
        if (potential_answers_list of the current node is not empty) then
            append potential_answers_list to the same list of the parent of the current node
else
    if (node type =Predicate) then
        check and update the predicate and increase predCounter by 1
        if (node axis = Descendant) then
            clear the predicate's stack
    else
        if (node type =Result) then
            if (node is the query's root) then
                calculate: NumberOfMatches, OutputSize ;
                output answers ;
            else
                append node to the potential answers list of the node's parent
```

Algorithm 10: appendOrDestroy (*nNumber*, *s*)

```
if (node type = Axis) then
    if the stack of the host node of the current node is empty then
        destroy s ;
    else
        append the list of the potential answers of the current node to the same list of the top node of the host
        stack (the host stack of the current node) ;
```



(a) XML *D*



(b) path tree of *D*

```
StartDocument("Doc")
StartElement ("A", [("nFreq","1") ; ("nSize","22") ] )
StartElement ("B", [("nFreq","1") ; ("nSize","22") ] )
StartElement ("A", [("nFreq","2") ; ("nSize","44")  ] )
StartElement ("B", [("nFreq","3") ; ("nSize","66")  ] )
StartElement ("E", [("nFreq","3") ; ("nSize","66")  ] )
EndElement ("E")
StartElement ("C", [("nFreq","3") ; ("nSize","66")  ] )
EndElement ("C")
StartElement ("D", [("nFreq","1") ; ("nSize","22")  ] )
EndElement ("D")
EndElement ("B")
StartElement ("C", [("nFreq","1") ; ("nSize","22")  ] )
EndElement ("C")
EndElement ("A")
StartElement ("E", [("nFreq","1") ; ("nSize","22")  ] )
EndElement ("E")
StartElement ("C", [("nFreq","1") ; ("nSize","22")  ] )
EndElement ("C")
EndElement ("B")
EndElement ("A")
EndDocument("Doc")
```

(c) SAX parser events of the path tree

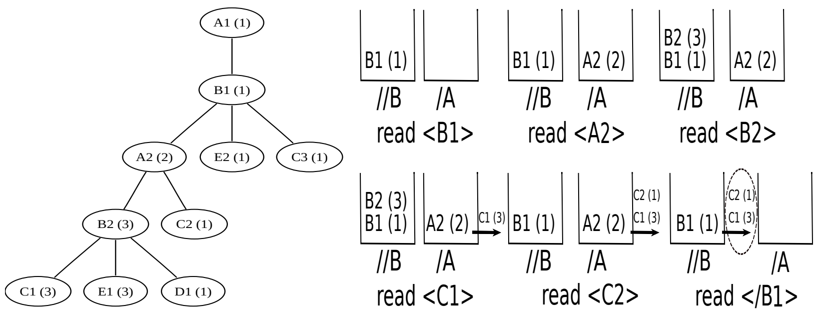
Figure 4.10: The XML document *D*, its path tree, and the SAX parser events of the path tree.

4.3.1.1 Selectivity Estimation - Simple Path

We first start by showing how our algorithm estimates the selectivity of a simple path p, where p does not contain same node-labels or wildcard nodes. After that, we explain this estimation once p contains same node-labels and a wildcard node.

- **Simple Path without wildcard nodes or same node-labels**:

  Figure 4.11(b) illustrates different snapshots of the evaluation process of the path tree of D on the simple path $//B/A//C$, which returns sequence $C1(3)$, $C2(1)$ as estimated answer nodes. For each non-leaf node, the algorithm creates a stack. Therefore, in this example, a stack is created for the root node B and another one for the node A.



(a) path tree of D (b) SAX parser events of the path tree

Figure 4.11: Snapshots of the run-time stacks for the evaluation of the path tree of D on Q ($//B/A//C$)

When $< A1 >$ is read, the parent stack (stack B) of $A1$ is empty, therefore, $A1$ is discarded (algorithm 7 line 1). When $< B1 >$ is read, $B1$ is pushed (with its information, e.g. *nSize* and *nFreq*) in its corresponding stack B. Same thing for $< A2 >$ and $< B2 >$ (algorithm 7 lines 16-17), Note that the values of $Cache$ and $WorkingSpace$ are updated.

When $< C1 >$ is read, $C1$ is a descendant of the node $A2$, as long as its parent stack is not empty, $C1$ is buffered (with its information) to the potential answers list of its parent node $A2$, also the values of $Buffer$ and $WorkingSpace$ are updated (algorithm 7 lines 13-14).

When $< C2 >$ is read, the node $B2$ was already popped out from its stack. Concerning $C2$, it is a descendant of the node $A2$. As long as its parent stack is not empty, $C2$ is buffered (with its information) to the potential answers list of its parent node $A2$, also the values of $Buffer$ and $WorkingSpace$ are updated (algorithm 7 lines 13-14).

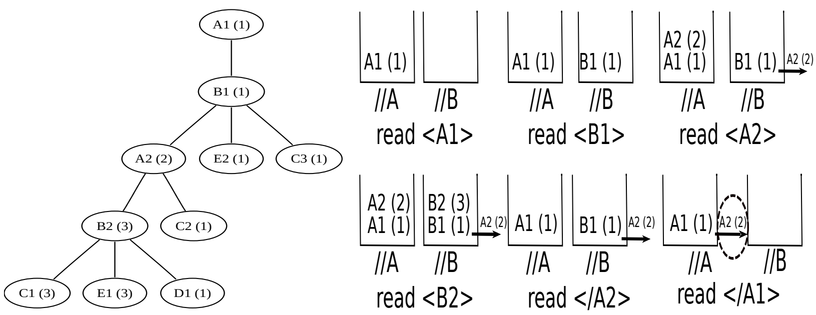
When $< /B1 >$ is read, the node $A2$ was popped out from its stack and its potential answers list was appended to the same list of its parent node $B1$

(algorithm 8 lines 21-22 then algorithm 9 line 8-9). The node $B1$ is the root node of the query, therefore, the content of its potential answers list is flushed as answers (algorithm 8 lines 21-22 then algorithm 9 lines 2-6).

Before, we show the estimated values, we remind that in the SAX parser events of the path tree, the list of attributes l for each $StartElement(e,l)$ contains two attributes: *nFreq* which is the frequency of e and *nSize* which is the size in byte of e.
The result of the XPath query estimation is as follows (estimated values):
$NumberOfMatches$: the value is 4, they are: $C1(3)+C2(1) = 3+1 = 4$. $Buffer$: for any simple path, the value of $Buffer$ is the same as the value of $NumberOfMatches$. $Cache$: the value is 6, they are: $B1(1)$, $A2(2)$ and $B2(3)$. Based on this, the value of $Cache$ is $1+2+3=6$ $WoringSpace$: its size was estimated to $(88+132) = 220$ byte $= 0.0002$MiB. $OutputSize$:its size was estimated to 88 byte $= 0.00008$MiB.

- **Simple path with same node-labels**:

Figure 4.12(b) illustrates different snapshots of the evaluation process of the path tree of D on the simple path $//A//B//A$, which returns $A2(2)$ as estimated answer. For each non-leaf node, the algorithm creates a stack. Therefore, in this example, a stack is created for the root node A and another one for the node B.



(a) path tree of D (b) SAX parser events of the path tree

Figure 4.12: Snapshots of the run-time stacks for the evaluation of the path tree of D on Q ($//A//B//A$)

When $< A1 >$ is read, the function *startBlock* is called in the post-order of A in Q, that is 3,1. The node $A1$ with order 3 is not buffered because its parent stack (stack B) is empty (algorithm 7 line 1). While $A1$ with order 1 is pushed (with its information, e.g. *nSize* and *nFreq*) in its corresponding stack A and the values of *Cache* and *WorkingSpace* are updated (algorithm 7 lines 16-17).

When $< B1 >$ is read, it is pushed (with its information, e.g. *nSize* and *nFreq*) in its corresponding stack B and the values of *Cache* and *WorkingSpace* are updated (algorithm 7 lines 16-17).

When $< A2 >$ is read, the function *startBlock* is called in the post-order of A in Q, that is 3,1. The node $A2$ with order 3 is buffered (with its information) to the potential answers list of node $B1$ and the values of $Buffer$ and *WorkingSpace* are updated (algorithm 7 lines 13-14). While $A1$ with order 1 is pushed (with its information) in its corresponding stack A, and the values of *Cache* and *WorkingSpace* are updated (algorithm 7 lines 16-17).

When $< /B1 >$ is read, it is popped out from its stack and its potential answers list is appended to the same list of its parent node $A1$ (algorithm 8 lines 21-22 then algorithm 9 lines 8-9).

Finally, when $< /A1 >$ is read, $A1$ is the root node of the query, therefore, the content of its potential answers list is flushed as answers (algorithm 8 lines 21-22 then algorithm 9 lines 2-6).

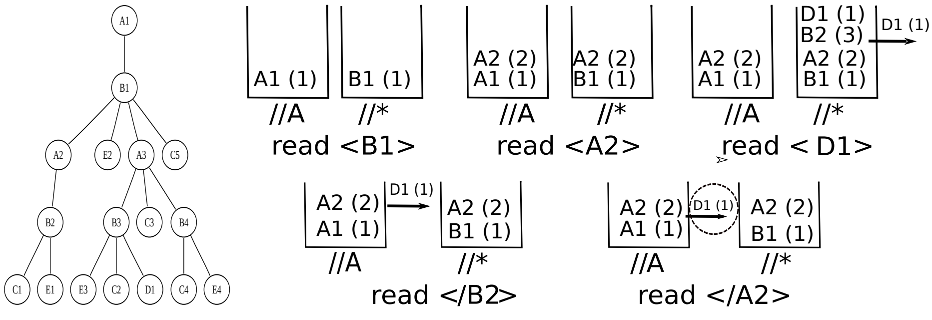
The result of the XPath query estimation is as follows (estimated values): $NumberOfMatches$: the value is 2, they are: $A2(2) = 2$. $Buffer$: for any simple path, the value of $Buffer$ is the same as the value of $NumberOfMatches$. *Cache*: the value is 7, they are: $A1(1)$, $A2(2)$ and $B1(1)$, $B2(3)$. Based on this, the value of *Cache* is $1+2+1+3 = 7$. *WoringSpace*: its size was estimated to $(44+154) = 198$ byte $= 0.0002$MiB. $OutputSize$: it size was estimated to 44 byte $= 0.00004$MiB.

- **Simple path with a wildcard**:

Figure 4.13(b) illustrates different snapshots of the evaluation process of the path tree of D on the simple path $//A//*//D$ which returns $D1(1)$ as a result node. For each non-leaf node, the algorithm creates a stack. Therefore, in this example, a stack is created for the root node A and another one for the node $*$.

When $< A1 >$ is read, the function *startBlock* will is called in the post-order of A in Q, that is 2,1. The node $A1$ with order 2 is not pushed in its stack (stack $*$) because its parent stack (stack A) is empty (algorithm 3 line 1). While $A1$ with order 1 is pushed (with its information, e.g. *nSize* and *nFreq*) in its corresponding stack A and the values of *Cache* and *WorkingSpace* are updated (algorithm 7 lines 16-17).

When $< B1 >$ is read, it is pushed in its corresponding stack $*$ (algorithm 3 line 14).



(a) XML document D (b) Snapshots of the run-time stacks for the evaluation of D on Q

Figure 4.13: Snapshots of the run-time stacks for the evaluation of the path tree of D on Q ($//A//*//D$)

When $< A2 >$ is read, the function *startBlock* is called in the post-order of A in Q, that is 2,1. The $A2$ with order 2 is pushed (with its information) in its corresponding stack $*$ and the values of *Cache* and *WorkingSpace* are updated (algorithm 7 lines 16-17). While $A2$ with order 1 is pushed (with its information) in its corresponding stack A and the values of *Cache* and *WorkingSpace* are updated (algorithm 7 lines 16-17).

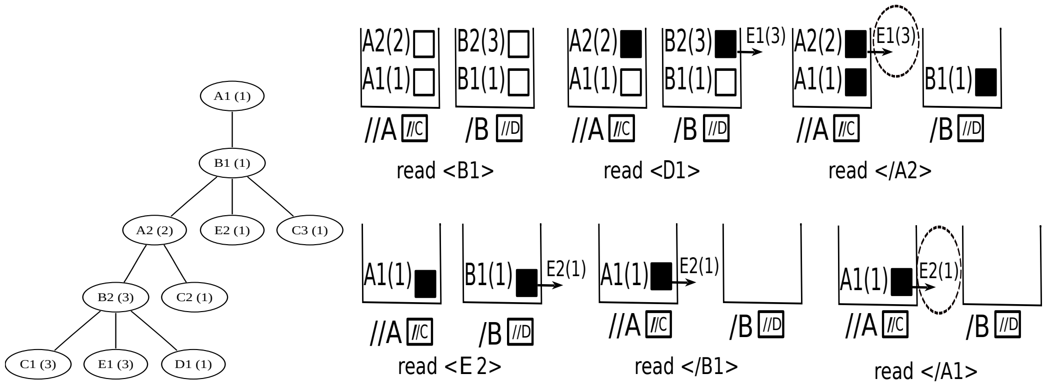
When $< D1 >$ is read, the function *startBlock* is called in the post-order of A in Q, that is 3, 2. Note that the node $B2$ was read and pushed (with its information) to its corresponding stack $*$, therefore, the node $D1$ with order 3 is buffered (with its information) to the potential answers list of its parent node $B3$, and the values of *Buffer* and *WorkingSpace* are updated (algorithm 7 lines 13-14). While the node $D1$ with order 2 is pushed in its corresponding stack $*$.

When $< /B2 >$ is read, $B2$ is popped out from its stack, and its potential answers list is appended to the same list of its parent node $A2$ (algorithm 8 lines 21-22 then algorithm 9 lines 8-9).

Finally, when $< /A2 >$ is read, the function *endBlock* is called in the pre-order of A in Q, that is 2, 3. The node $A1$ with order 1 is the root node of the query, therefore, the content of its potential answers list is flushed as answers (algorithm 8 lines 21-22 then algorithm 9 lines 2-6). While for the node $A2$ with order 2, it is popped out from its corresponding stack $*$.

The result of the XPath query estimation is as follows (estimated values): *NumberOfMatches*: the value is 1, it is$D1(1) = 1$. *Buffer*: for any simple path, the value of *Buffer* is the same as the value of *NumberOfMatches*. *Cache*: the value is 19, we present them based on their stacks as follows: stack A contains $A1(1)$, $A2(2) = 1+2 = 3$. While stack $*$ contains $A1(1)$, $A2(2)$, $B1(1)$, $B2(3)$, $C1(3)$, $C2(1)$, $C3(1)$, $E1(3)$, $E2(1) = 1+2+1+3+3+1+1+3+1 = 16$. So the estimated value is $3+16 = 19$.

WoringSpace: its size was estimated to $(22+418) = 440$ byte $= 0.0004$MiB. *OutputSize*: it size was estimated to 22 byte $= 0.00002$MiB.



(a) path tree of D (b) Snapshots of the run-time stacks for the evaluation process

Figure 4.14: Snapshots of the run-time stacks for the evaluation of the path tree of D on Q ($//A[.//C]/B[.//D]//E$)

In the next section, we present several examples on th selectivity estimation by using our selectivity estimation technique with twigs.

4.3.1.2 Selectivity Estimation - Twig Path

Below, we present two examples on the selectivity estimation of twig paths.

- **Twig path with multi predicates**:

  Figure 4.14 illustrates different snapshots of the evaluation process of the path tree of D on the twig path $//A[.//C]/B[.//D]//E$ which returns $E1(3)$, $E2(1)$ as result nodes. For each non-leaf node, the algorithm creates a stack. Therefore, in this example, a stack is created for the root node A and another one for the node B.

  When $< B2 >$ is read, the nodes $A(1)$, $B1(1)$, and $A2(1)$ were read and pushed (with their information) in their stacks. Concerning the node $B2$, it is also pushed (with its information) in its stack B. Note that for each pushed node, the values of $Cache$ and $WorkingSpace$ are updated. (algorithm 7 lines 16-17).

  When $< D1 >$ is read, the node $C1$ was read, therefore, the value of the predicate C of $A2$ was changed to true, and the value of $NumberOfPredEvaluation$ was updated. The node $E1$ was read, therefore, $E1$ was buffered (with its information) to the potential answers list of its parent node $B2$, and the values of $Buffer$ and $WorkingSpace$ were updated (algorithm 7 lines 13-14). Moreover, by reading $D1$, the value of the predicate D of $B2$ was changed to true and the value of $NumberOfPredEvaluation$ was updated.

  When $< /A2 >$ is read, the node $B2$ was popped out from its stack, and the true value of its predicate C was passed to its ancestor $B1$, and the value of

$NumberOfPredEvaluation$ was updated (algorithm 8 line 6). Furthermore, the potential answers list of $B2$ was appended to the same list of its parent node $A2$ (algorithm 9 lines 8-9). Concerning $A2$, it is popped out of its stack, and as long as it is the root node, the content of its potential answers list is flushed as answers (algorithm 8 lines 13-14 then algorithm 9 lines 2-6).

When $E2$ is read, it is buffered (with its information) to the potential answers list of its parent node $B2$, and the values of $Buffer$ and $WorkingSpace$ are updated (algorithm 7 lines 13-14).

When $< /B1 >$ is read, it is popped out from its stack and its potential answers list is appended to the same list of its parent node $A1$. Finally, when $< /A1 >$ is read, it is popped out from its stack, $A1$ is the root node, therefore, the content of its potential answers list is flushed as answers (algorithm 8 lines 13-14 then algorithm 9 lines 2-6).

The result of the XPath query estimation is as follows (estimated values):
$NumberOfMatches$: the value is 4, they are: $E1(3)$, $E2(1) = 3 + 1 = 4$. $Buffer$: in this example, the value of $Buffer$ is the same as $NumberOfMatches$.

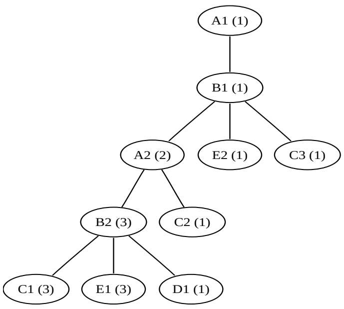
$Cache$: the value is 7, we present them based on their stacks as follows: stack A contains $A1(1)$, $A2(2)$, while stack B contains $B1(1)$, $B2(3)$. The value then $1+2+1+3 = 7$. $WoringSpace$: its size was estimated to 0.0002MiB.
$OutputSize$: its size was estimated to 0.00008MiB.
$NumberOfPredEvaluation$: its estimated value is 6, that is $C1(3)$ + $D1(1)$ + two times the predicate values were passed between elements= 6.

- **Twig path with *and* operator and same node-labels**:

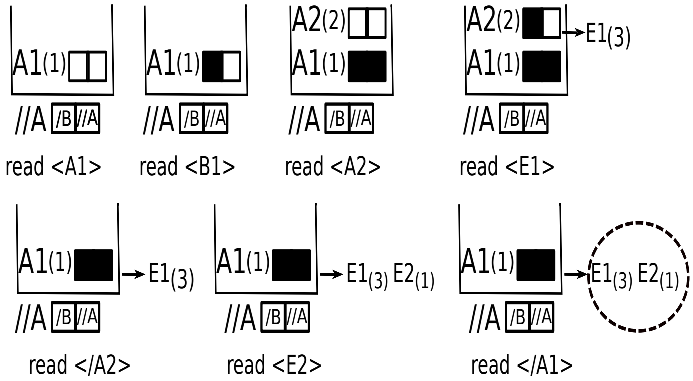
Figure 4.15 illustrates different snapshots of the evaluation process of the path tree of D on the twig path $//A[./B \ and \ .//A]//E$ which returns $E1(3)$, $E2(1)$ as result nodes. For each non-leaf node, the algorithm creates a stack. Therefore, in this example, a stack is created for the root node A.

When $< A1 >$ is read, the function $startBlock$ is called in the post-order of A in Q, that is 3,1. The predicate node $A1$ with order 3 is not evaluated because its parent stack (stack A) is empty (algorithm 7 line 1). While $A1$ with order 1 is pushed (with its information) in its corresponding stack A with false values for its both predicate nodes B and A, moreover, the values of $Cache$ and $WorkingSpace$ are updated (algorithm 7 lines 16-17).

When $< B1 >$ is read, $B1$ is a direct child for node $A1$, therefore the value of the predicate B of the node $A1$ is changed from false to true. Moreover, the value of $NumberOfPredEvaluation$ is updated (algorithm 7 lines 5-6).



(a) path tree of D



(b) Snapshots of the run-time stacks for the evaluation process

Figure 4.15: Snapshots of the run-time stacks for the evaluation of the path tree of D on Q ($//A[./B \; and \; .//A]//E$)

When $< A2 >$ is read, the function *startBlock* is called in the post-order of A in Q, that is 3, 1. The value of the predicate node A with order 3 for $A1$ is changed from false to true because its parent stack (stack A) is not empty, it contains the node $A1$. Furthermore, the value of $NumberOfPredEvaluation$ is updated (algorithm 7 line 5-6). While $A2$ with order 1 is pushed (with its information) in its corresponding stack A with false values for its both predicate nodes B and A, moreover, the values of *Cache* and *WorkingSpace* are updated (algorithm 7 line 14).

When $< E1 >$ is read, as long as it is a descendant of $A2$, the node $E1$ is buffered (with it information) to the potential answers list of its parent node $A2$.

When $< /A2 >$ is read, it is popped out from its stack. $A2$ is the root node, but its predicate node A is not satisfied (algorithm 8 line 13), therefore, the function *appendOrDestroy* is called (algorithm 8 line 16). The host stack of A is the stack A itself ($host[A] = A$), as long as this stack is not empty (it contains node $A1$), the potential answers list of $A2$ is appended to the same list of $A1$.

When $< E2 >$ is read, as long as it is a descendant of $A1$, $E2$ is buffered (with it information) to the potential answers list of its parent node $A1$.

Finally, when $< /A1 >$ is read, it is popped out from its stack. $A1$ is the root node, as long as the values of its predicates B and A are true, then, the content of its potential answers list ($E1(3)$ and $E2(1)$) is flushed as a final answer.

The result of the XPath query estimation is as follows (estimated values): $NumberOfMatches$: the value is 4 , they $E1(3)$, $E2(1)$ $= 3 + 1 = 4$.

$Buffer$ is this example, the $Buffer$ has the same value as the the value of $NumberOfMatches$ that is 4. $Cache$: the value is 3, they are $A1(1)$, $A2(2)$ $= 1+2 = 3$. $WoringSpace$: its size was estimated to $= (22+44)+(66+22) = 154$ byte $= 0.0001$MiB. $OutputSize$: its size was estimated to 88 byte $= 0.00008$MiB. $NumberOfPredEvaluation$: its estimated value is 6, that is $B1(1)$ + $A2(2)$ +$B2(3)$ = 6.

4.3.2 Accuracy of the Selectivity Estimation Technique

The average relative error was used to measure the accuracy of our approach, it is defined as follows: $\frac{1}{n}\sum_{i=1}^{n}|\frac{M_i-P_i}{M_i}|$, where M_i is the measured value of the $i-$th query in the workload and P_i is its predicted one.

Table 4.1 summarizes the measured and the estimated values of the five XPath queries used in our precedent examples. As it can be seen, the measured and the estimated values are equal which is an indication for the accuracy of our selectivity estimation technique.

XPath query	Measured values-LQ						Estimated values-Estimation algorithm					
	$NumberOfMatches$	$Buffer$	$Cache$	$WorkingSpace$	$OutputSize$	$NumberOfPredEvaluation$	$NumberOfMatches$	$Buffer$	$Cache$	$WorkingSpace$	$OutputSize$	$NumberOfPredEvaluation$
$//A/A//C$	4	4	6	0.0002	0.00008	-	4	4	6	0.0002	0.00008	-
$//A//B//A$	2	2	7	0.0002	0.00004	-	2	2	7	0.0002	0.00004	-
$//A//*//D$	1	1	19	0.0004	0.00002	-	1	1	19	0.0004	0.00002	-
$//A[.//C]/B[.//D]//E$	4	4	7	0.0002	0.00008	4	4	4	7	0.0002	0.00008	6
$//A[./B$ and $.//D]//E$	4	4	3	0.0001	0.00008	6	4	4	3	0.0001	0.00008	6

Table 4.1: Measures to show the accuracy of the selectivity estimation technique

After an exhaustive testing on real and synthetic data sets (e.g., TreeBank [Suciu 1992] and XMark [Schmidt 2001]), we noticed that the selectivity estimation of our technique for any simple path p is 100% correct due to the complete structure of the path tree synopsis. Moreover, the selectivity estimation for twig paths of our technique is very accurate due to the complete structure of the path tree synopsis and the efficiency of our selectivity estimation algorithm.

In this chapter, we introduced our selectivity estimation technique. The result of the selectivity estimation process (estimated values) of our technique, makes it well suited to be embedded in a cost model for XPath query evaluation.
In the next chapter, we present our performance prediction (cost) model. Moreover, we show the important rule of our selectivity technique in the performance prediction model.

CHAPTER 5

Performance Prediction Model

Contents

5.1 Introduction

XML [Bray 2008] is one of the most important standards for document storage and interchange, and its convenient syntax improves the interoperability of many applications. Yet the format is intrinsically costly in space and efficient access to XML data requires careful processing of XPath queries. Despite a logically clean structure, the computational complexity of XPath [Berglund 2010], XQuery [Boag 2010] queries can vary dramatically [TenCate 2009] [Gottlob 2005] and the unconstrained use of XPath leads to unpredictable space and time costs.

One proposed approach to combine the simplicity of XML data, the declarative nature of XPath queries and reasonable performance on large data sets is to impose their processing by purely streaming algorithms. The result is that queries must be restricted to a fragment of XPath but on the other hand processing space can be limited and very large documents can be accessed efficiently.

There are many parameters that influence processing space and time (as we explained in chapter 1): the lazy vs eager strategy of the stack-automaton, the size and quantity of query results, the size and structure of the document etc.
The author of an XPath query may have no immediate idea of what to expect in memory consumption and delay before collecting all the resulting sub-documents. This unpredictability can diminish the practical use of XPath stream-processing.

In chapter 4 (Selectivity Estimation Techniques) we explained that selectivity estimations is desirable in interactive and internet applications. The system could warn the end user that for example his/her query is so coarse that the amount of results will be overwhelming. But is not sufficient to model the query cost. Moreover, it measures neither the total amount of memory allocated by the program to find these matches (space used) nor the processor time used by the program to find the matches (time spent).

In this chapter, we start by explaining the main idea and the general structure of the performance prediction model. This model provides us in advance with expected time/space for a sent XPath query Q on a document D. After that, we present in details two performance prediction models, they are: (1) Performance Prediction Model - Simple Path, and (2) Performance Prediction Model - Twig Path.

5.2 Performance Prediction Model- Preliminaries

Our *Performance Prediction Model* is a cost model which estimates the cost (in terms of space used and time spent) of an XPath query before actually executing it.

There are many parameters that influence processing space and time: the lazy vs eager strategy of the stack-automaton, the size and quantity of XPath query results, the size and structure of the document etc. The author of an XPath query may have no immediate idea of what to expect in memory consumption and delay before collecting all the resulting sub-documents. This unpredictability can diminish the practical use of XPath stream-processing. Therefore, to estimate the cost for a given XPath query, we need to determine these parameters and to decide their relations with time and space.

A stream of XML data is the depth-first, left-to-right traversal of an XML document [Bray 2008]. By definition stream-processing linearizes data-access and assumes limited temporary storage (heap or stack size). Moreover, bounded-memory processing is abstractly equivalent to efficient parallel processing [Parberry 1987] and XPath must be restricted to have a parallel-efficient [Gottlob 2005] or even decidable [TenCate 2009] querying problem. We are thus forced to consider a fragment of XPath but work remains to make even this limited problem efficient and predictable.

In this section, we start by presenting our motivations for a performance prediction (cost) model (section 5.2.1). After that, we present a preliminary study we performed to confirm the linear relationship between the stream-processing and the data-access resources (section 5.2.2). Finally, we present the general structure for the performance prediction model (5.2.3).

5.2.1 Performance Prediction Model - Motivations

We summarize our motivations for improving stream-querying and cost prediction as follows:

- In certain situations processing the XML document occurs through portable devices with small memory sizes, hence the need to minimize space or predict its overflow.

- Developing cost models for query optimization is significantly harder for XML queries than for traditional relational queries. The reason is that XPath query operators are much more complex than relational operators such as table scans and joins.

- Selectivity estimations are highly desirable in interactive and internet applications. The system could warn the end user that his/her query is so coarse that the amount of results will be overwhelming.

- Selectivity is necessary but not sufficient to model costs: for example the existence of 5 matches somewhere at the beginning of a very large XML document, might cost less than finding one match somewhere at the end of it. Hence the need to model match-position distribution and match-tree sizes.

- A 2005 study [Teevan 2005] of Yahoo's query logs revealed that 33% of the queries from the same user were repeated and that 87% of the time the user would click on the same result as earlier: repeat queries are used to revisit information. This suggests that systems learn from past queries and make performance repeatable if not gradually improving. Indeed, tabulation can be seen as an extreme case of a performance model: it contains so much information about performance that it converges to the actual query results.

In the next section, we study the relationship between stream-processing and data-access resources.

5.2.2 Performance Measurements Towards the Optimization of Stream-processing for XML Data

In this section, we present a study we performed to confirm the linear relationship between the stream-processing and data-access resources [Alrammal 2009b]. As we will see later (section 5.3 and section 5.4), this linear relationship has an important role in our performance prediction models.

5.2.2.1 Prototype `O-Search`

We developed the core of a prototype called `O-Search` to have better understanding for the complexity of *stream-querying* algorithms in practice, with respect to different structures of XML documents (wide, depth, various size). The evaluation technique used in our prototype is Lazy. Figure 5.1 shows the structure of `O-Search`.

`O-Search` will become our intermediate prototype for *stream-querying* of XML

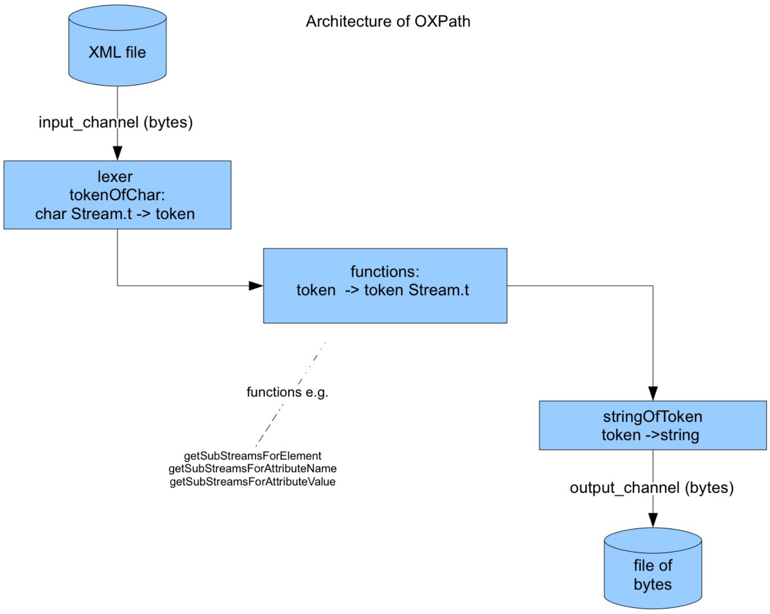


Figure 5.1: Architecture of `O-Search`.

Data. It is implemented using the functional language OCaml [1] [Leroy 2010b]. We have implemented the basic search functions necessary for realizing XPath queries. To explain figure 5.1 we start in the input that is a simplified XML file which has the abstract syntax as in figure 5.2.

```
type token =
       StartDocument of string
      | StartElement of string * (string * string) list
      | EndElement of string
      | Text of string
      | EndDocument of string ;;
```

Figure 5.2: Abstract Syntax

An example of the concrete syntax for figure 5.2 is figure 5.3(a). Notice that 5.3(b) is its XML tree model. There are basically two types of nodes in the XML tree model:

- Element nodes: these correspond to tags in XML documents, and correspond to `StartElement` token in our concrete syntax, for example `StartElement("A",[])`. An attribute list is associated (optionally) with tags in the XML document, therefore it is associated with `StartElement` tokens in our concrete syntax, for example

[1]Ocaml is a language of the ML family developed by INRIA since the early 1980's. It is well adapted to tree processing and its optimizing compiler ocamlopt produces fast executables

```
StartDocument("Doc1")
StartElement("A",[])
StartElement("B",[])
EndElement("B")
StartElement("C",[])
Text("Any text")
EndElement("C")
EndElement("A")
EndDocument("Doc1")
```

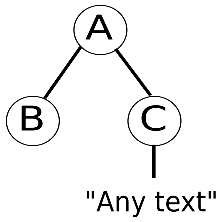


Figure 5.3: XML Tree Model.

`StartElement("B",[("attr","2")])`. Note that, the attribute list is not nested (an attribute can not have any sub-elements), not repeatable (tow same-name attributes can not occur under one element) and unordered (attributes of an element can freely interchange their occurrences location under element). These constraints are standard to XML.

- Text nodes: these correspond to data values in XML document, and correspond to `Text` token in our concrete syntax, for example `Text("Any text")`.

To read the input file (input_channel), we implemented a lexer named `tokenOfCharStream`. It reads the input file line by line as a stream of characters and generates a token for each line, see the function below:

```
val tokenOfCharStream:
char Stream.t -> token Stream.t
```

The token generated by the lexer will be used by the processing function for matching and processing purposes. After each match the lexer is called repeatedly to generate another token. An example of this function is:

```
val getSubStreamsForElement:
string ->in_channel -> string -> unit
```

were the input arguments are:

- *string*: our query.
- *in_channel*: the input file.
- *string*: the name of the output file.

`getSubStreamsForElement` calls recursively many other internals functions to generate the result as stream of tokens then call recursively the function:

```
val stringOfToken:
token -> string
```

to convert each token in the stream to string and sent it to the output_channel.

5.2.2.2 Experimental Results

Experimental Setup

In this section we explain the experimental setup needed for the performance measurements by using our prototype.

Data sets: to conduct the performance measurements, we implemented two types of synthetic data sets. These data sets are described below:

1. **Wide tree data set**: it has a shallow structure that does not include recursive elements. To generate the wide tree data set, we use the following function:

```
val generateWideTree:
string -> int -> in_channel
```

   where:

   - *string*: is the output file name which will contain the wide tree data set.
   - *int*: is the number of the shallow subtrees in the wide tree data set. Queries will refer to each subtree's "token rank" (see figure 5.4).

   To know the total number of the tokens generated in our wide tree data set that was used for the performance tests, we use the following equation:

$$\text{Tree/Data set size(tokens)= } (n * 10) + 4$$

   where:

   - n: is the loop number. It is proportional to the tree data set width
   - 10: is the number of tokens generated in each subtree.
   - 4: is a constant number that represents:

```
1- StartDocument ("Doc 1")
2- StartElement("root", [])
3- EndElement("root")
4- EndDocument("Doc1")
```

   Figure 5.4 is an example of wide tree data with the following size:

$$\text{Tree/Data set size(tokens)= } (100000 * 10) + 4$$

2. **Deep tree data set**: it has a narrow deep structure. To generate the deep tree data set, we use the following function:

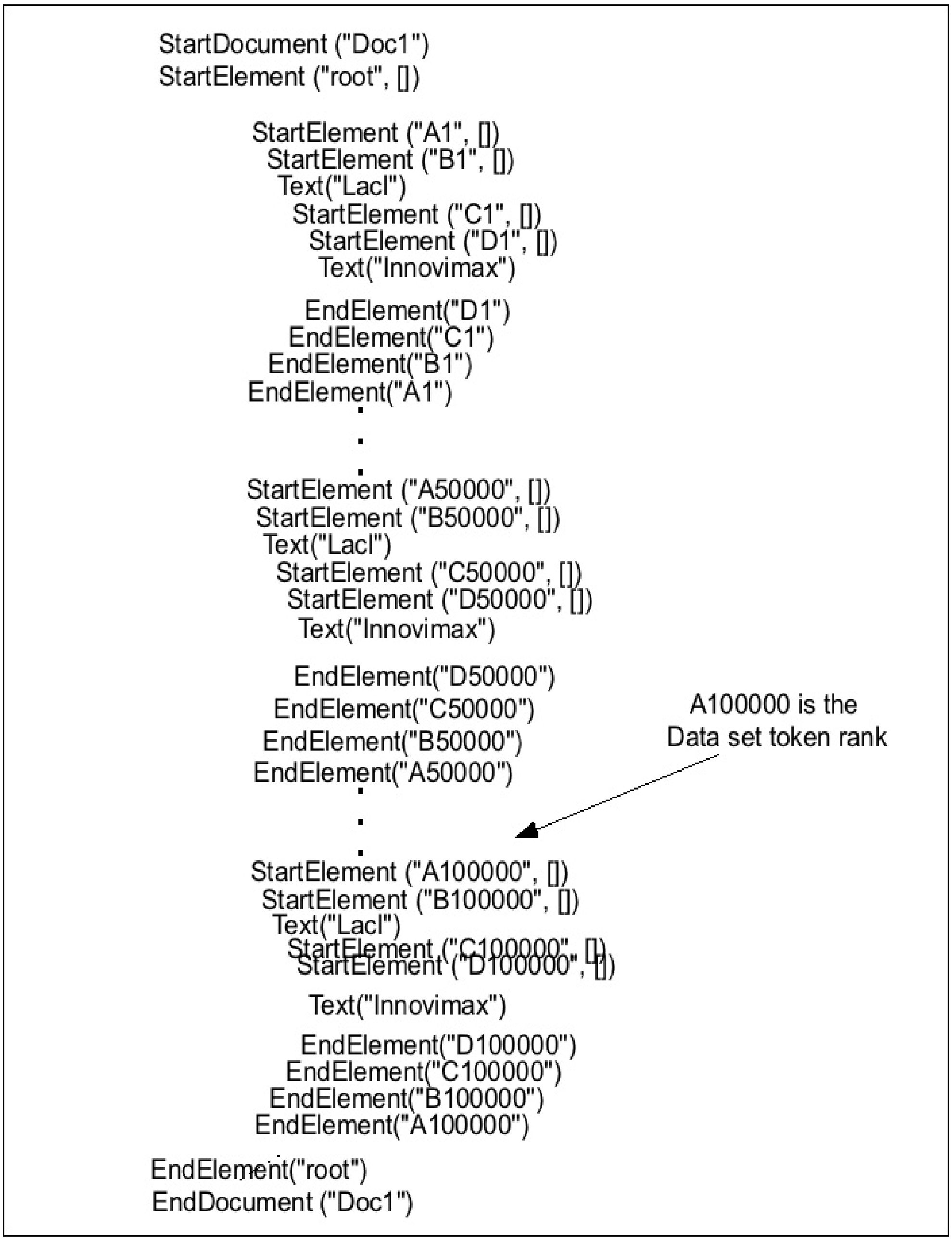


Figure 5.4: Wide tree data set

Processor name	Intel Core 2 Duo
Processor speed	2.4 GHz
Memory	4 GB.
OS	Mac OSX Version 10.5.5.

Table 5.1: Specifications of the test machine

```
val generateDeepTree:
string -> int -> in_channel
```

where:

- *string*: is the output file name which will contain the deep tree data set.
- *int*: is the loop depth of the deep tree data set. Queries will refer to "token rank" in this tree data set (see figure 5.5).

To know the total number of tokens generated in our deep tree data set that was used for the performance tests, we use the following equation:

$$\text{Tree/Data set size(tokens)=}\ ((n*6)+(n*4))+\mathbf{4}$$

where:

- n: is the loop number. It is proportional to the depth tree data set.
- 6: is the number of tokens (`StartElement and Text`) that are generated in each recursion.
- 4: is the number of tokens (`EndElement`) that are generated in each recursion.
- **4**: is a constant number that represents:

```
1- StartDocument ("Doc 1")
2- StartElement("root", [])
3- EndElement("root")
4- EndDocument("Doc1")
```

Figure 5.5 is an example of deep tree data set with the following size:

$$\text{Tree/Data set size(tokens)=}\ ((100000*6)+(100000*4))+\mathbf{4}$$

Test machine: experiments were performed using a MacBook machine with the following technical specifications:

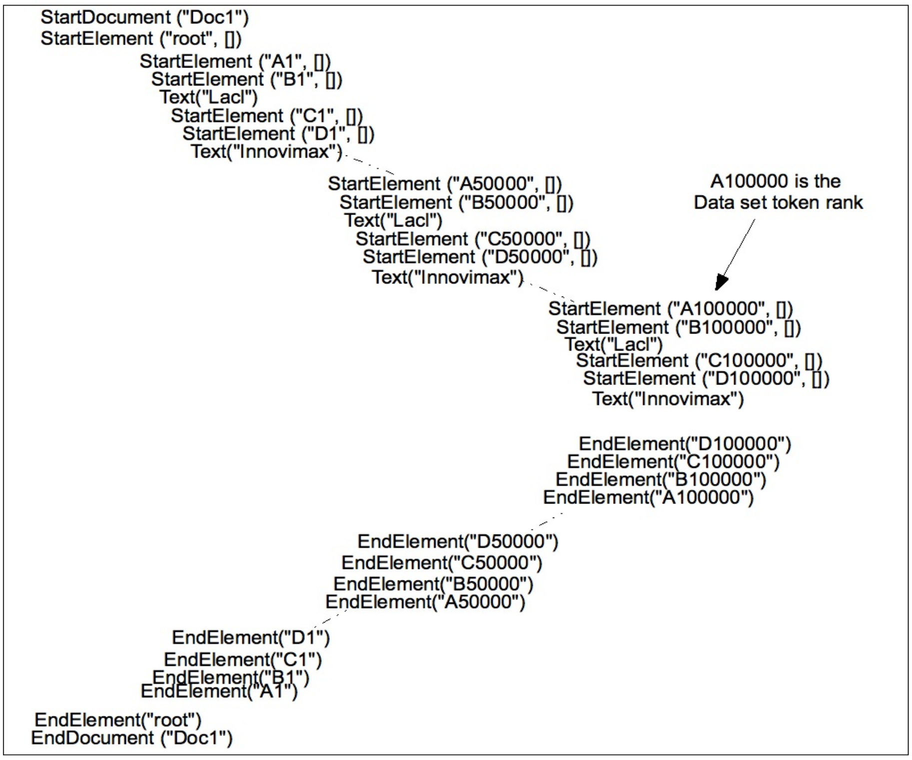


Figure 5.5: Deep tree data set

Test measurements:

- Time: to measure execution time, we use the following function `Sys.time();;`. This function exists in the module Sys[2] of OCaml. It has a type: `unit -> float`, and it returns the processor time (in seconds) used by the program since the beginning of execution. To return the time of generating (for example) a wide tree data set, we use the following code:

```
let temp=ref (Sys.time()) ;;
generateWideTree "inputFile.txt" n ;;
 print_float (Sys.time()-. !temp);;
print_string " Second\n"; temp:=Sys.time();;
```

  To return the total time of a specific test, for example: `getSubStreamsForElement`, we use the following code:

```
let temp=ref (Sys.time());;
getSubStreamsForElement <token name>
                 (generateDeepTree "inputFile.txt" <Data set token rank>)
```

[2]Sys: portable system calls.

```
                    "outputFile.txt" ;;
print_float (Sys.time()-. !temp);;
print_string " Second\n"; temp:=Sys.time();;
```

- Memory:

  We measure the maximum depth of the running time stack (caching) using the following function:

```
val getMaxDepth:  unit -> int
       getMaxDepth();;
```

  this function returns an integer which indicates the maximum instantaneous number of tokens in the stack.

Queries: we used the following processing function: `getSubStreamsForElement`. Our tests have the following form:

```
getSubStreamsForElement<Token name>
        (generateWideTree "inputFile,txt"<data set token rank>)
        "outputFile,txt";;
```

where `Token name` can have the following values:

- `A1`: return the subtree of the element which exists at the beginning of the tree data set.

- `A<token rank/2>`: return the subtree of the element which exists in the middle of the tree data set.

- `A<Data set token rank>`: return the subtree of the element which exists at the end of the tree data set.

Furthermore, we use Positive and Negative queries, where:

- Positive: is a query that does not return an empty result.

- Negative: is a query which return an empty result. We use negative queries with the two types of the tree data sets (Wide and Deep) as a reference for the performance measurement.

Note that, `O-Search` support other processing functions, for example:

```
getSubStreamsForAttributeName<Token name>
        (generateWideTree "inputFile,txt" <Data set token rank>)
        "outputFile,txt" ;;
getSubStreamsForAttributeValue<Token name>
        (generateWideTree "inputFile,txt"<Data set token rank>)
        "outputFile,txt";;
getSubStreamsForTextEqual<Token name >
        (generateWideTree "inputFile,txt"<Data set token rank>)
        "outputFile,txt" ;;
```

Compiler: to test the execution time, we compile the ML file using ocamlopt: the Objective Caml high-performance native-code compiler. Generated code is almost 10 times faster than generated code by ocamlc.
To test the memory consumption, we use ocamlc which compiles CAML source files to byte-code object files and links these object files to produce standalone bytecode executable files. These executable files are then run by the bytecode interpreter ocamlrun. For memory tests, it is recommended to use ocamlc because it is more accurate than ocamlopt [Leroy 2010a].

Results

This section details our measurements for (time/space) for the following data sets (Wide tree/Deep tree) for positive queries. Those tests are then repeated for negative queries.

Wide tree data set (Positive queries): we performed two tests using this data set, they are:

1. Time test
   To explain this test, we start in explaining table 5.2 which include all the information needed:
   - **Query**:

     ```
     getSubStreamsForElement
         <Token name>
         (generateWideTree "inputFile,txt" <Data set token rank> "outputFile,txt");;
     ```

   - **Test type**: table 5.2 contains three tests, they are: y1, y2, and y3. We change the value of Token query rank with each test.
   - **Token name**: the token name we search.
   - **Data set token rank**: the rank of token of subtree, for better understanding see figure 5.4.
   - **Input tree size (tokens)**: the total number of tokens generated in our wide tree data set.

Test type	Token name	Data set token rank	Input tree size(tokens)	T(total) In seconds	T(data set) In seconds	T(total-data set) In seconds	Stack max. depth In (tokens)
y1	A1	100000	1000004	2,42	0,61	1,81	3
y1	A50000	100000	1000004	2,41	0,61	1,8	3
y1	A100000	100000	1000004	2,42	0,61	1,81	3
y2	A1	500000	5000004	12,42	3,12	9,3	3
y2	A250000	500000	5000004	12,41	3,12	9,29	3
y2	A500000	500000	5000004	12,49	3,12	9,36	3
y3	A1	1000000	10000004	24,89	6,24	18,65	3
y3	A500000	1000000	10000004	24,92	6,24	18,67	3
y3	A1000000	1000000	10000004	26,17	6,24	19,92	3

Table 5.2: Wide tree data set - time and memory tests

- **T(total)**: the processor time in seconds since the beginning of the execution to generate the tree data set and to answer the query.
- **T(data set)**: the processor time in seconds since the beginning of the execution to generate the tree data set.
- **T(total-data set)**: the processor time in seconds since the beginning of the execution to answer the query.
- **Stack max. depth (token)**: is the maximum depth of the running time stack.

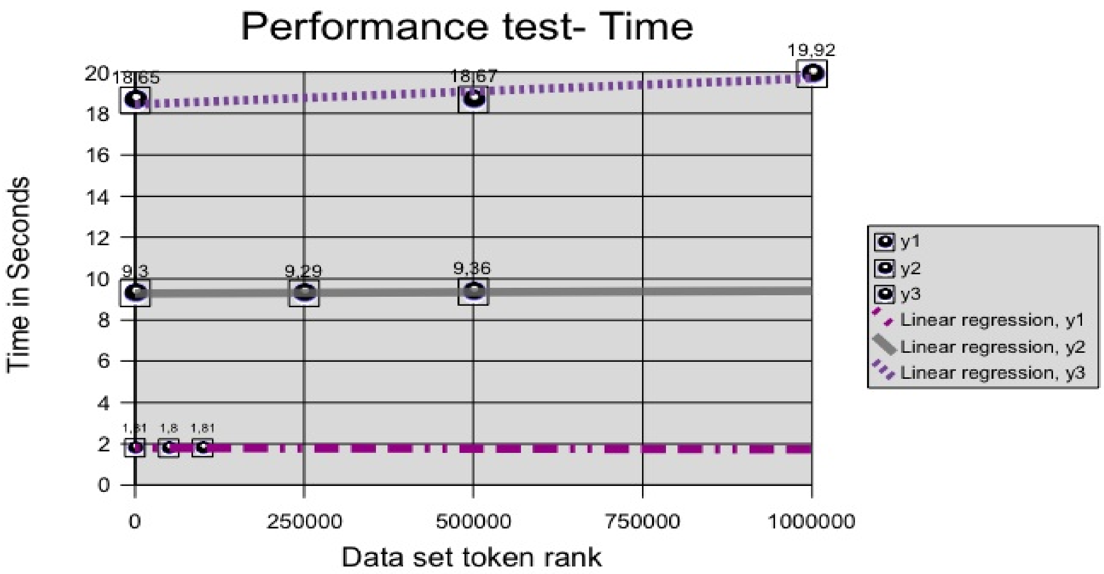


Figure 5.6: Wide tree data set - Time

Figure 5.6 represents three tests to measure the execution time in the wide tree data set. Test y1 uses a 1MiB token document, y2 a 5MiB token document and y3 a 10MiB token document. We noticed that test y1 is steady linear at 1,81 seconds irrespective of the data set token rank. Also, test y2 is almost steady linear at 9,3 seconds. While test y3 is almost steady linear at 18,7 seconds with a slight increasing particularly at the point (A1000000 -

1000000). Therefore, we conclude that execution time is independent of the data set token rank in the wide tree data set. We observe directly proportional to the input tree size: curves y2, y3 are multiples of y1 in this proportion.

2. Memory test
   Table 5.2 contains all the information needed. The query used:

   - **Query**:

```
getSubStreamsForElement
 <Token name>
(generateWideTree "inputFile,txt" <Data set token rank> "outputFile,txt");;
```

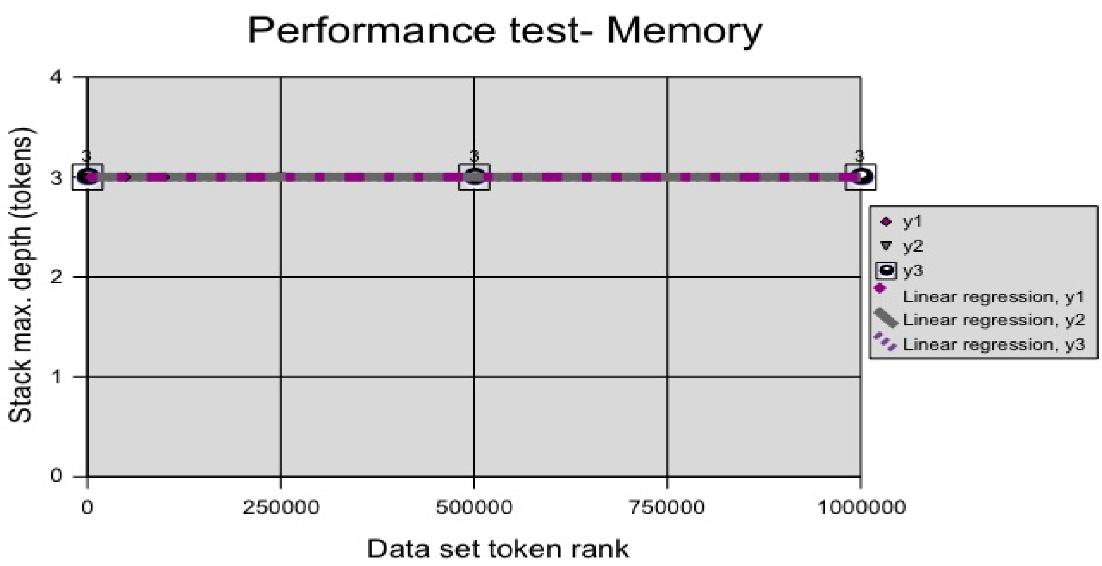


Figure 5.7: Wide tree data set - Memory.

Figure 5.7 represents three tests to measure the memory allocated to answer our query in the wide tree data set. Tests y1, y2 and y3 have the same value for the maximum number of tokens in the running stack which is 3 due to the symmetry of all subtrees in the wide tree data set. Therefore, we conclude that the stack max. depth (tokens) is independent of the data set token rank in the wide tree data set.

Deep tree data set (Positive queries): we performed two tests using this data set, they are:

1. Time
   The terms used in table 5.3 are the same as in table 5.2.

   Figure 5.8 represents three tests to measure the execution time in the deep tree data set. We noticed that execution time increases with the increasing of the data set token rank and the decreasing of the token name's value (see section 5.2.2.2) due to the increasing of the tokens those correspond the query.

Test type	Token name	Data set token rank	Input tree size(tokens)	T(total) In seconds	T(data set) In seconds	T(total-data set) In seconds	Stack max. depth In (tokens)
y1	A1	1000	10004	5,73	0,01	5,72	3999
y1	A500	1000	10004	0,82	0,01	0,81	2003
y1	A1000	1000	10004	0,03	0,01	0,02	3
y2	A1	5000	50004	223,37	0,04	223,33	19999
y2	A2500	5000	50004	41,47 1	0,04	41,43	10003
y2	A5000	5000	50004	0,12	0,04	0,08	3
y3	A1	10000	100004	1193,32	0,07	1193,26	39999
y3	A5000	10000	100004	232,42	0,07	232,35	20003
y3	A10000	10000	100004	0,23	0,07	0,16	3

Table 5.3: Deep tree data set - time and memory tests

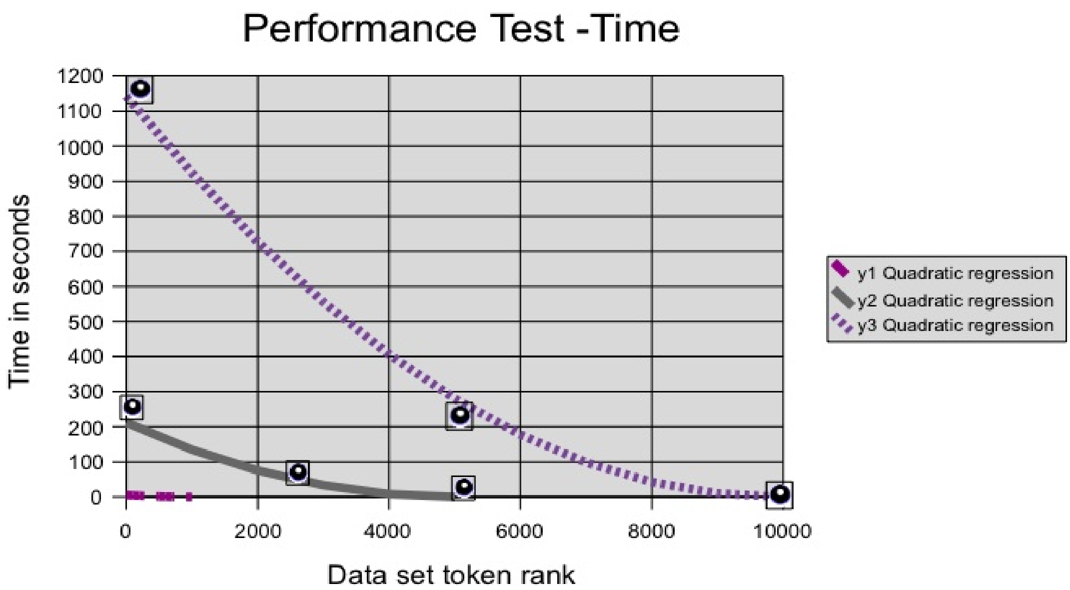


Figure 5.8: Deep tree data set - Time.

More precisely, in test y1 the relationship between the execution time (y) and data set token rank (x) is $y = (2.3 - 0.002 * x)^2$. In test y2 the relation is: $y = (14.5 - 0.03 * x)^2$. While for the test y3 it is: $y = (33.8 - 0.003 * x)^2$. Therefore, we conclude that execution time is negative-quadratic proportional to the data set token rank in the deep tree data set. The time-rank relationship should normally be negative-linear and its quadratic behavior in our tests is due to a naive list implementation of one primitive. This problem was solved in our new implementation for the stream-querying algorithm in section 5.3.1.

2. Memory
   The terms used in table 5.3 are the same as table 5.2.

   Figure 5.9 represents three tests to measure the memory allocated to answer our query in the deep tree data set. We noticed that increasing the data set token rank and decreasing the value of query name will increase the value of stack max. depth (tokens). Furthermore, our evaluation technique is lazy,

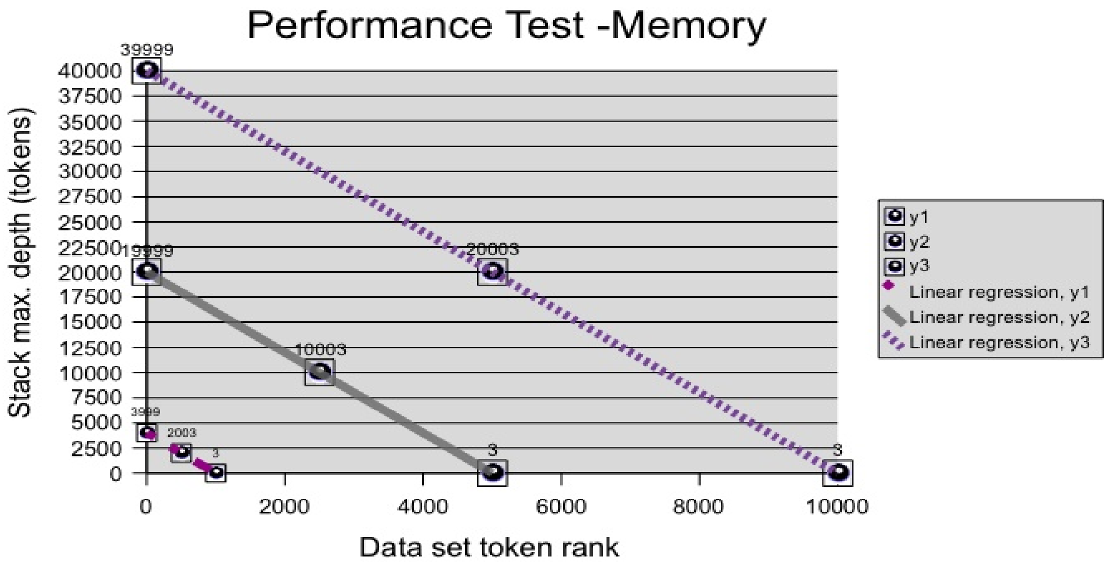


Figure 5.9: Deep tree data set - Memory.

Test type	Token name	Data set token rank	Input tree size(tokens)	T(total) In seconds	T(data set) In seconds	T(total-data set) In seconds	Stack max. depth In (tokens)
y1	A1000001	1000000	10000004	25,16	6,69	18,47	0
y1	A5000001	5000000	50000004	131,87	34,84	97,02	0
y1	A10000001	10000000	100000004	263,97	67,68	196,29	0

Table 5.4: Wide tree data set - time and memory tests (negative queries)

therefore we are obliged to buffer the whole subtree. In test y1 the relationship between the memory usage (y) and data set token rank (x) is as the following $y = -4*x+4003$. In test y2 the relation is: $y = -4*x+20003$. While for the test y3 it is: $y = -4*x+40003$. Therefore, we conclude that stack max. depth (tokens) is inverse-linearly related to the data set token rank in the wide tree data set, and linear proportional to the document size.

Wide tree data set (Negative queries): we performed two tests using this data set, they are:

1. Time
   Terms in table 5.4 have the same meaning as in table 5.2.

   Figure 5.10 represents a test to measure the execution time in the wide tree data set for negative queries. We noticed that execution time increases with the increasing of the data set token rank due to the increasing of matching processes. More precisely, in test y1 the relationship between the execution time (y) and data set token rank (x) is as the following $y = 0,00002*x-1.82$. The importance of this test is to compare the measurements between the wide tree data set and the deep tree data set by using negative queries.

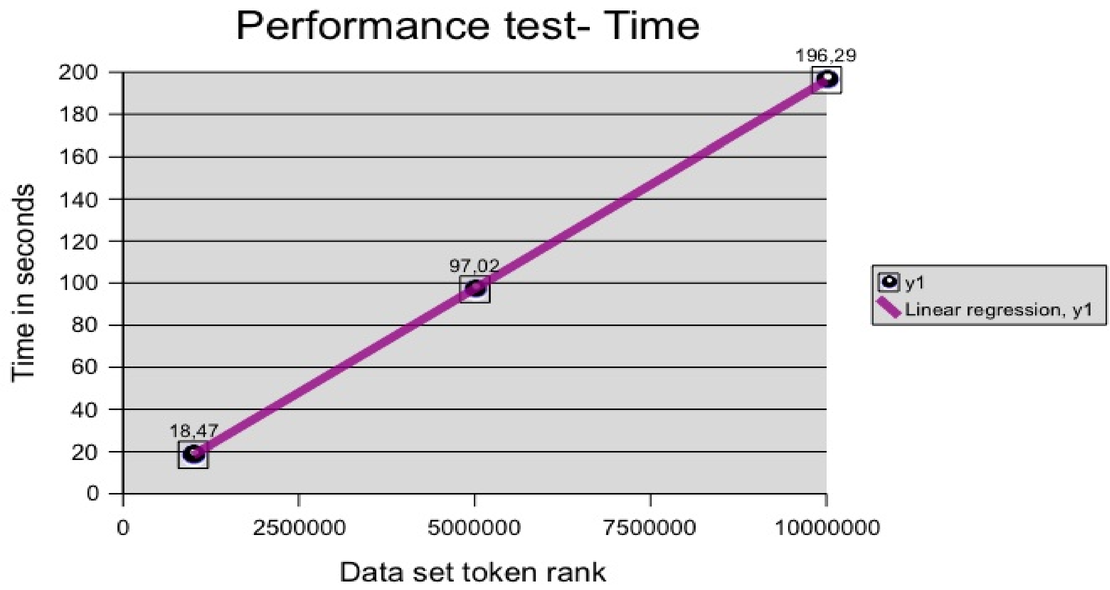


Figure 5.10: Wide tree data set - Time (negative queries).

Test type	Token name	Data set token rank	Input tree size(tokens)	T(total) In second	T(data set) In second	T(total-data set) In second	Stack max. depth In(tokens)
y1	A1000001	1000000	10000004	24,79	6,73	18,06	0
y1	A5000001	5000000	50000004	129,17	34,66	94,51	0
y1	A10000001	10000000	100000004	260,4	68,69	191,71	0

Table 5.5: Deep tree data set - time and memory tests (negative queries)

2. Memory
   Terms in table 5.4 have the same meaning as table 5.2.

   Figure 5.11 represents a test to measure the maximum depth of the running stack (in tokens) to answer our negative query in the wide tree data set. We notice that the increasing the data set token rank does not affect the stack max. depth because our query is negative so we do not need to cache any element. We conclude that stack max. depth (tokens) is independent of the data set token rank in the wide tree data set.

Deep tree data set (Negative queries): we performed two tests using this data set, they are:

1. Time
   Terms in table 5.5 have the same meaning as in table 5.2.
   Figure 5.12 represents a test to measure the execution time in the deep tree data set using negative queries. We noticed that execution time increases with the increasing of the data set token rank due to the increasing of matching processes. More precisely, in test y1 the relationship between the execution time (y) and data set token rank (x) $y = 0.00002 * x - 1.56$. Comparing the two linear equations to measure the execution time between both deep/Wide

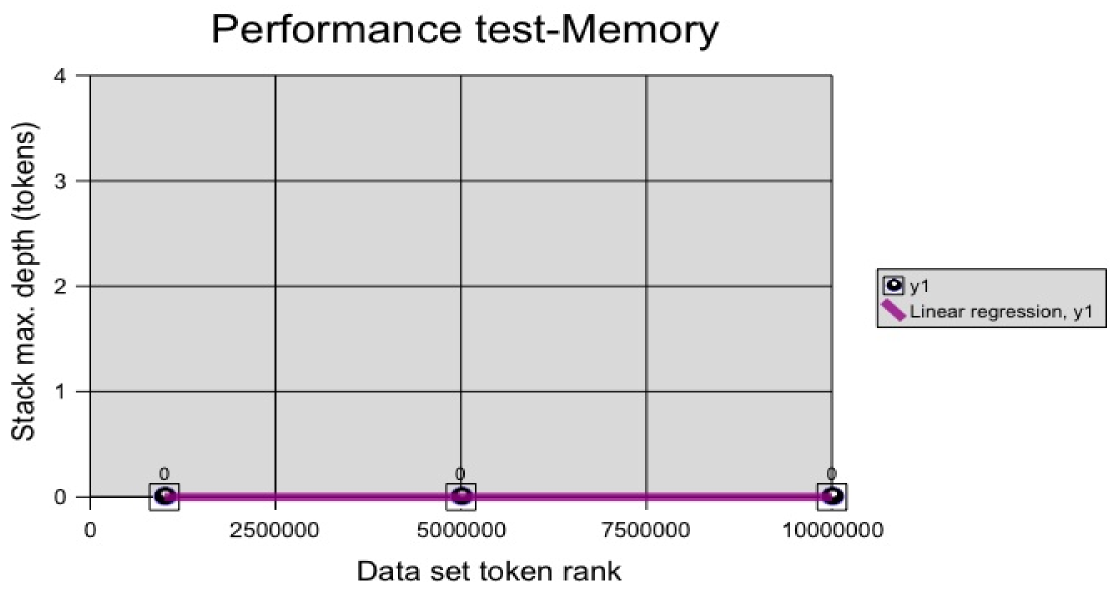


Figure 5.11: Wide tree data set - Memory (negative queries).

tree data sets using negative query indicates that deep tree data set has a better time performance than wide tree data set.

2. Memory
   Terms in table 5.5 have the same meaning as in table 5.2.

   Figure 5.13 represents a test to measure the maximum depth of the running stack (in tokens) to answer our negative query in the deep tree data set. We notice that the increasing the data set token rank does not affect the stack max. depth because our query is negative so we do not need to cache any element. We conclude that stack max. depth (tokens) is independent of the data set token rank in the deep tree data set.

5.2.2.3 Conclusion

The study performed above confirmed the linear relationship between the stream-processing and data-access resources. This relationship indicates the following:

1. The stream-querying algorithm used for stream-processing should not have a complexity more than linear. Notice that the complexity of our lazy stream-querying algorithm LQ (introduced in chapter 4) which processes the fragment of Forward XPath is linear.

2. The complexity of the selectivity estimation algorithm used in a performance prediction model should not have a complexity more than linear. Note that, our selectivity estimation algorithm (introduced in chapter 4) has a linear complexity

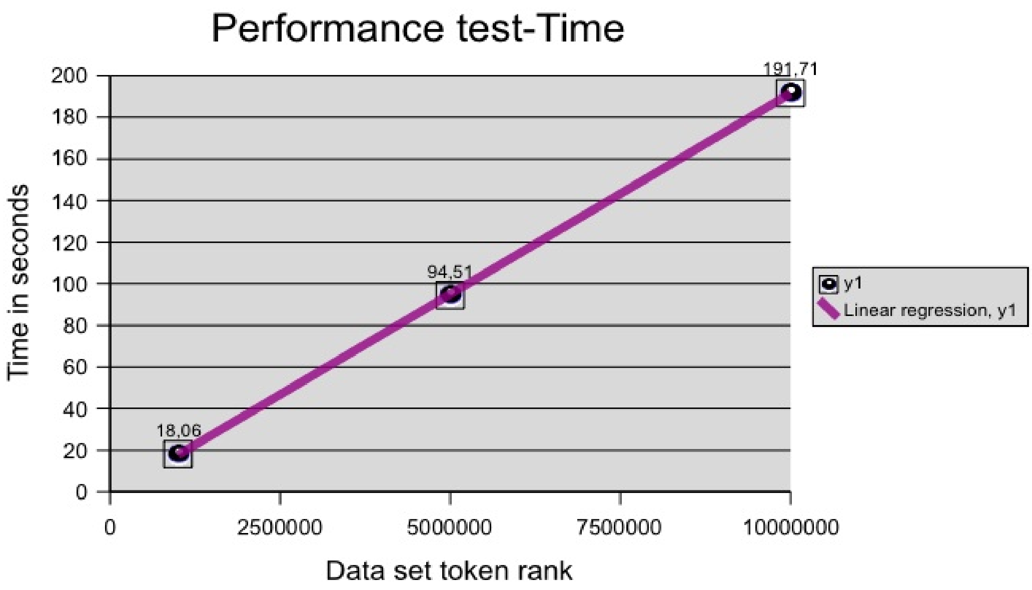


Figure 5.12: Deep tree data set - Time (negative queries).

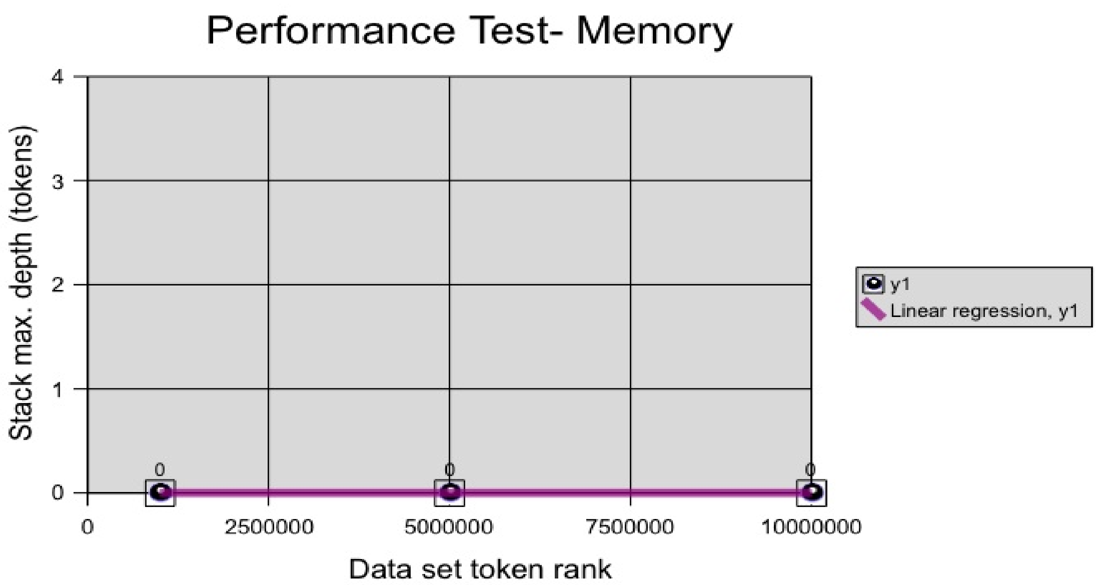


Figure 5.13: Deep tree data set - Memory (negative queries).

3. A linear regression approach can be used (in the performance prediction model) to model the cost for a given query over stream of XML data.

In sections 5.3 and 5.4 we explain the importance of the linear regression approach in our performance prediction models.

In the next section, we present the general structure for the performance prediction model.

5.2.3 Performance Prediction Model - General Structure

The performance prediction model consists of a large number of input (XPath, XML, Machine) - response (Statistics) pairs, used to construct an estimate of the input-performance relationship by capturing the underlying trends and extracting them from the noise. Then, a part of the information is discarded and the resulting model is used to predict the responses of the new input.

Figure 5.14 illustrates our performance prediction (cost) model.

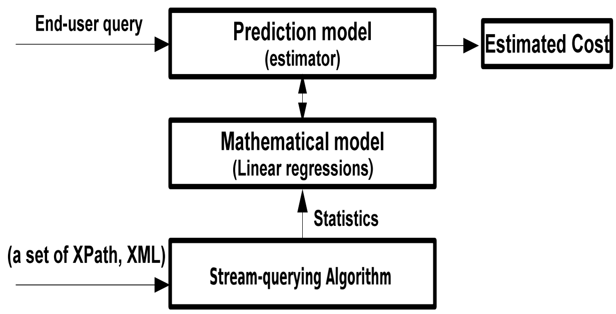


Figure 5.14: General structure - Performance Prediction Model

To built this model, we need a stream-querying algorithm to send training queries (a set of XPath queries) on the target XML document in order to get on the statistics needed (layer 1 of figure 5.14). After that, statistics are used to build a mathematical model which consists of a set of linear regression functions that will be used to estimate the cost for a given XPath query (layer 2 of figure 5.14). Then, the moment the end user send an XPath query, the function *estimator* analyses it and estimates the values of the input parameters of the mathematical model by using certain prediction technique (layer 3 of figure 5.14). *estimator* provides the end user with the estimated cost for his XPath query (which was calculated by the mathematical model).

In the next section, we present the performance prediction model - simple path. We will explain in details the whole process: how to get on the statistics needed, how to build the mathematical model, what are the prediction rules or technique used and how to optimize the stream-querying process if the cost estimated does not fit the end user needs and resources.

5.3 Performance Prediction Model - Simple Path

The performance prediction model -simple path is a cost model which estimates the cost (in terms of space used and time spent) for a simple path before actually executing it.

It consists of a large number of input (XPath, XML, Machine) - response (statistics) pairs, used to construct an estimate of the input-performance relationship by capturing the underlying trends and extracting them from the noise. Then, a part of the information is discarded and the resulting model is used to predict the responses of the new input. More precisely, the resulting cost model contains two functions, they are:

`estimator: XPath*SearchingRange*XML*Machine` $\rightarrow$ `Cost Estimated (space used/time pent)`

The function *estimator* respects the syntax of the functional language OCaml [Leroy 2010b]. It takes as input four parameters and gives as output the cost estimated for the given XPath query. The input parameters are:

- *XPath*: is a sub fragment of XPath [Berglund 2010] where the used XPath is a structural pattern composed of sub expressions called steps. Each step consists of child or descendant axis (defines the tree-relationship between the selected nodes and the current node), a name nodeTest (identifies a node within an axis), and zero or one predicates (to further refine the selected node-set) at the last step of XPath. e.g. $//A/B/C[.//D]$.

- ***SearchingRange***: is a part of the statistics used to construct the cost model. It is augmented implicitly to XPath to orient the searching process. In this syntax, we search over a subset of the XML document as specified by the searching range (an interval of search as will see later). Notice that, the end user will get a complete answer for his XPath query.

- *XML document*: is the XML document used to construct the cost model and we query it to find answers for the input parameter XPath.

- *Machine*: is the machine used to construct the cost model.

If the cost estimated by *estimator* do not fit the end user needs, in this case, we use the second function of the cost model *optimizer* which is described below.

`optimizer: XPath*ImposedValues*XML*Machine` $\rightarrow$ `Cost Estimated (space used/time spent)`

To conduct further optimization, we propose *optimizer*, it is *estimator* augmented with the input parameter *ImposedValues*: which are values imposed by the end user based on his needs and resources. For example: the end user can impose the total amount of memory allocated by the program to process an XPath query Q. *optimizer* performs implicitly several mathematical estimations to adapt (recalculate the searching range) the stream-querying process with the value imposed by the end user.

Figure 5.15 illustrates our performance prediction (cost) model - simple path. The number to the right of each layer of the model corresponds to the number of the section where this layer is explained in details. To built this model, we need

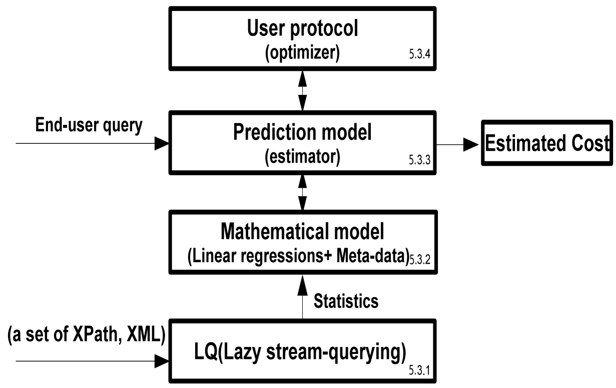


Figure 5.15: Layers of our performance prediction model - simple path

a stream-querying algorithm to send training queries (set of XPath queries) on the target XML document in order to get on the statistics needed. For this purpose, we adapted and used the stream-querying algorithm LQ [Gou 2007] (layer 1 of figure 5.15). After that, statistics are used to build a mathematical model which consists of a set of linear regression functions that will be used to estimate the cost for a given XPath query (layer 2 of figure 5.15). Then, the moment the end user send an XPath query, the function *estimator* analyses it and estimates the values of the input parameters of the mathematical model by using certain prediction rules (layer 3 of figure 5.15) and a part of the statistics (e.g. searching range). *estimator* provides the end user with the estimated cost for his XPath query (which was calculated by the mathematical model). If the estimated cost fits the end user needs and resources, then, the stream-querying algorithms LQ is used to process the XPath query. If not, we use the second function of the cost model *optimizer* to allow further optimization for the XPath query processing as we mentioned above (layer 4 of figure 5.15).

Next, we will explain in details the whole process: how to get on the statistics needed, how to build the mathematical model, what are the prediction rules used and how to optimize the stream-querying process if the cost estimated does not fit the end user needs and resources.

5.3.1 Lazy Stream-querying Algorithm (LQ)

To enable our experimental study we implemented the stream-query algorithm LQ of [Gou 2007]. We chose to use it because LQ handles recursion in the XML document and repetition of node labels (same node-labels) in the XPath query, neither of which can be done using [Peng 2003] and [Chen 2006]. LQ was implemented by using the functional language OCaml release 3.11 [Leroy 2010b] which combines relatively high performance with strong typing and ML-language constructs for tree processing. The current version of LQ (figure 5.16) functions as follows: it takes two input parameters. The first one is the XPath query that will be transformed to a query table statically using our Forward XPath Parser. After that, the main function is called. It reads the data set line by line repeatedly, each time generating a tag. Based on that tag a corresponding startBlock (SB) or endBlock (EB) function is called to process it. Finally, the main function generates as output the statistics needed.

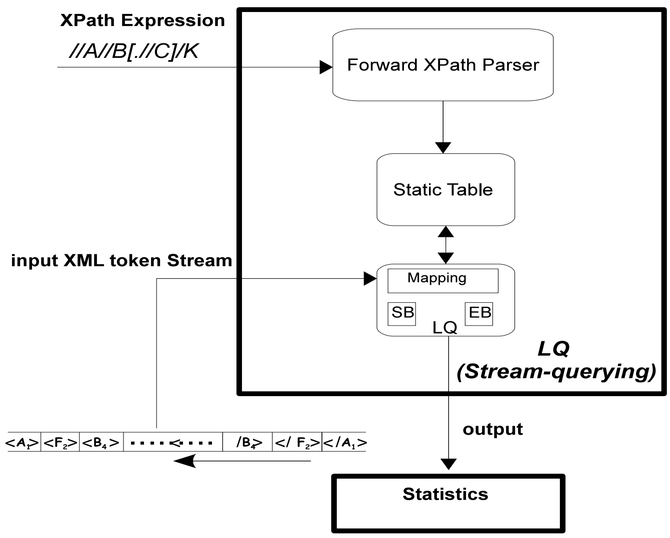


Figure 5.16: LQ (Lazy stream-querying algorithm)

Statistics consist of the following:

- *Cache*: is the number of elements cached in the running-time stacks during the processing of the XPath query Q on the XML document D. They correspond to the axis nodes of Q.
- *Buffer*: is the number of potential answer elements which are buffered during the processing of the XPath query Q on the XML document D. They correspond to the answer nodes of the XPath query Q.
- *NumberOfMatches*: is the number of answer elements found during the processing of the XPath query Q on the XML document D.
- *StartLT*: the location of the Start-Tag of the first element X in the XML document order that corresponds to the root node of the XPath query Q.

- *EndLT*: the location of the End-Tag of the last element Y in the XML document order that corresponds to the result node of the XPath query Q.
- *MinTime*: returns the processor time used by the program since the beginning of execution till finding the first answer.
- *AvgTime*: (sum (time for each match))/total *NumberOfMatches*.
- *MaxTime*: the processor time used by the program since the beginning of execution till the end of XML document processing.
- *MinMemory*: the total amount of memory allocated by the program since the beginning of execution till finding the first answer.
- *AvgMemory*: (sum (memory size for each match))/total *NumberOfMatches*.
- *MaxMemory*: the total amount of memory allocated by the program since the beginning of execution till the end of XML document processing.

5.3.2 Building the Mathematical Model

As illustrated in figure 5.17, the first step is to send training queries to collect the information needed (statistics) by using the stream-querying algorithm LQ. These statistics will be stored in a hash table. We call our technique for sending training queries and collecting the statistics by *exhaustive testing*: a comprehensive process to test all possible not repeated XPath queries existing in the data set and having the following forms: $//A/B$, where A and B can be any element name in the data set, and A is a parent of B.
The moment we have this information, we use them to build the mathematical model. Our mathematical model consists of a set of linear regressions, they are:

- *MinTime* vs (*Buffer*, *Cache*, *StartLT*, *EndLT*).
- *AvgTime* vs (*Buffer*, *Cache*, *StartLT*, *EndLT*).
- *MaxTime* vs (*Buffer*, *Cache*, *StartLT*, *EndLT*).
- *MinMemory* vs (*Buffer*, *Cache*, *StartLT*, *EndLT*).
- *AvgMemory* vs (*Buffer*, *Cache*, *StartLT*, *EndLT*).
- *MaxMemory* vs (*Buffer*, *Cache*, *StartLT*, *EndLT*).

To build a part of the mathematical model (the linear function) which will be used to estimate the value of *MaxMemory*, we linearize the *MaxMemory* vs (*Buffer*, *Cache*, *StartLT*, *EndLT*) relation *i.e.*we perform a linear regression on our initial measurements. The same process is applied on (*MinTime*, *AvgTime*, *MaxTime*, *MinMemory*, *AvgMemory*) to obtain the complete mathematical model.

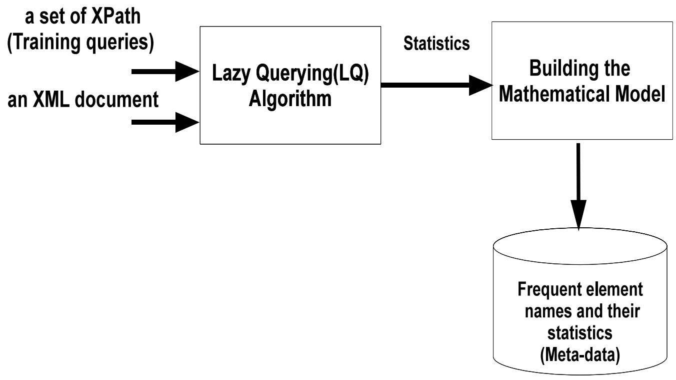


Figure 5.17: Building the Mathematical Model

For example: to linearize *MaxMemory* vs *Buffer*, we calculate the slope and intercept of this relation.

In the next section 5.3.3, we explain how the prediction rules use these linear regressions to predict the XPath query cost.

Once our mathematical model is built, all data (statistics) in the hash table will be discarded to free the memory. Then, we apply exhaustive testing, this time we use $//A$ instead of $//A/B$, where A can be any element name in the data set. The advantage of the this process is to store the frequent element names and a part of their statistics (*Buffer*, *Cache*, *NumberOfMatches*, *StartLT*, *EndLT*) in a hash table. This information is our ***metadata*** which helps us to estimate the cost for the queries sent by the end user.

The number of frequent element names of the data set TreeBank 64KiB is 32, where frequent element name means element name exists 3 times or more in the data set.

5.3.3 Building the Prediction Model

As illustrated in figure 5.18, the end user sends his/her XPath query to the function $estimator$. This function analyses the query and uses *metadata* in additional to certain prediction rules to estimate the values of the input parameters of the mathematical model. These input parameters are: (*Buffer*, *Cache*, *StartLT*, *EndLT*). Each value of an input parameter will be used by its corresponding linear regression function in the mathematical model. The average of the linear regressions results is calculated to estimate the cost for a given XPath query. The cost estimated for a given XPath query is: *MinTime*, *AvgTime*, *MaxTime*, *MinMemory*, *AvgMemory*, *MaxMemory*.

In certain cases, we need to send partial queries to enrich the metadata if some parameters values are missing.

Below we present the prediction rules of our performance prediction (cost)

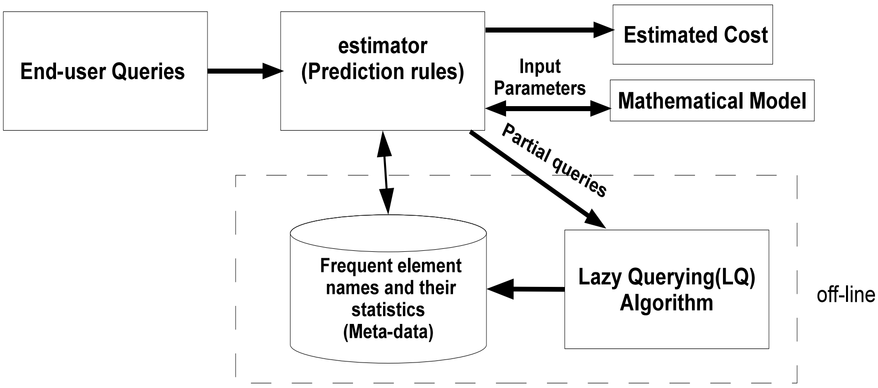


Figure 5.18: Building the Prediction Model

model.

5.3.3.1 Prediction Rules

We classify the prediction process into many cases. Each case may have several prediction rules. The purpose of these rules is to estimate the values of the input parameters of the mathematical model. For simplicity, we provide explaining examples for three cases. A detailed explanation of the prediction rules can be found at [Alrammal 2010].

Case (1) positive node tests of the end user query:

Table 5.6 represents our metadata (frequent element names and their statistics). Note that these element names are distinct e.g. $A <> B <> C$.

Element name	Cache	Buffer	NumberOfMatches	StartLT	EndLT
A	0	0	20	500	1500
B	0	0	15	700	2000
C	0	0	10	1000	2500

Table 5.6: Frequent element names and their metadata for Case (1)

The end user query is $//A/B/C$. According to the content of table 5.6, the node tests of the query are positive. In this case, the prediction rule is as follows:

- *NumberOfMatches* for element name A plus those for B will be the upper bound estimated value of *Cache* for XPath $//A/B/C$.
- *NumberOfMatches* for element name C will be the upper bound estimated value of *Buffer* for XPath $//A/B/C$.
- *StartLT* for element name A will be the lower bound value of *StartLT* for XPath $//A/B/C$.
- *EndLT* of element name C will be the upper bound value of *EndLT* for XPath $//A/B/C$.

XPath	Cache	Buffer	StartLT	EndLT
$//A/B/C$	35	10	500	2500

Table 5.7: Result of prediction rules for Case (1)

Table 5.7 shows the learning process from metadata for this case. The values of the parameters (*Buffer*, *Cache*, *StartLT*, *EndLT*) will be used by the mathematical model to estimate the cost of $//A/B/C$.
By symmetry this case is also valid for the following XPath: $//A/C/B$, $//B/A/C$, $//B/C/A$, $//C/A/B$, $//C/B/A$.

Case (2) metadata contains negative element name:

A negative element name in metadata table is an element name that does not belong to the XML data set. In certain cases, it can be a frequent one because its name has the same name of node tests which belong to repeated queries sent by the end users. In this case we add it to the hash table of metadata. (in a certain period, this node test was used in several queries more than 3 times).
Below, we provide an example with respect to metadata in table 5.8.

Element name	Cache	Buffer	NumberOfMatches	StartLT	EndLT
A	0	0	20	500	1500
B	0	0	0	0	0
C	0	0	10	1000	2500

Table 5.8: Frequent element names and their metadata for Case (2)

The query of the end user is $//A/C/B$. In this case, the prediction rules is as follows:

- *NumberOfMatches* for element name A plus those for C will be the upper bound estimated value of *Cache* for XPath $//A/C/B$.
- *NumberOfMatches* for element name B will be the upper bound estimated value of *Buffer* for XPath $//A/C/B$, unless *NumberOfMatches* for element name B is zero (as in table 5.8), then a message will be sent to the end user informing him in advance that this query is negative. This case is also valid for the following XPath: $//C/A/B$ by symmetry.

Case (3) no information about an element name in metadata:

As explained at the beginning of this document, It may be impractical to store information about all the element names. Here we suggest possible solution to this absence of information.
Below, we provide an example with respect to metadata in table 5.9 (we do not have any information about the element name C neither positive nor negative).

The end user query is $//A/B/C$. In this case, the prediction rule is as follows:

Element name	Cache	Buffer	NumberOfMatches	StartLT	EndLT
A	0	0	20	500	1500
B	0	0	15	700	2000

Table 5.9: Frequent element names and their metadata for Case (3)

- *NumberOfMatches* for element name *A* plus those for *B* will be the upper bound estimated value of Cache for XPath $//A/B/C$. We obtain the metadata of element name *C* as follows:
  We implicitly send the query $//C$ to get its metadata (in this case we obtained: *NumberOfMatches* = 7 and *EndLT* = 1900), then:

- *NumberOfMatches* for element name $//C$ will be the upper bound estimated value of *Buffer* for XPath $//A/B/C$.

- *StartLT* for element name *A* will be the lower bound estimated value of *StartLT* for XPath $//A/B/C$.

- *EndLT* for element name *C* will be the upper bound estimated value of *EndLT* for XPath $//A/B/C$.

XPath	Cache	Buffer	StartLT	EndLT
$//A/B/C$	35	7	500	1900

Table 5.10: Result of prediction rules for Case (3)

Table 5.10 shows the learning process from metadata for this case. Sending implicitly the XPath query $//C$ is recommended because we send it only once to help us to predict the cost of the following XPath: $//A/C/B$, $//B/A/C$, $//B/C/A$, $//C/A/B$, $//C/B/A$, $//A/C$, $//B/C$, $//C/A$, $//C/B$ and to know in advance whether the above mentioned queries are negative or not.

5.3.4 User Protocol

The *user protocol* is an interactive mode, it is used with our performance model to optimize the stream-querying process based on the end user needs and resources. For example, the end user can impose the total amount of memory allocated by the program to process an XPath query *Q*. Thus, our model performs implicitly several mathematical estimations to adapt the stream-querying process with the value imposed by the end users.

The interaction of the end user with our model can be summarized as follows:

- Imposing the maximum time: he can impose the value of the processor's time in seconds, used by the program to process an XPath query *Q*. For example: the time imposed by the end user to process query *Q* is 10s.

- Imposing the maximum memory: he can impose the value of the total amount of memory allocated by the program in KiB to process an XPath query Q. For example: the memory imposed by the end user to process query Q is 2048KiB.

- Imposing the maximum time and memory: he can impose the value of time and the value of memory allocated by the program to process an XPath query Q. For example: the time imposed by the end user to process query Q is 10s, and he memory imposed by the end user to process query Q is 2048KiB.

As mentioned above, the purpose of user protocol is to optimize the stream-querying process based on the needs and the resources of the end user. In this case, we search over a subset of the XML document as specified by the search range (an interval consists of *StartLT* and EndLT). This process yields a *stream-querying semi-algorithm* which is an algorithm returning correct but possibly incomplete results. In other words sub-documents outside the search range are only scanned but not processed as an attempt to improve performance. This strategy is based on our earlier measurements [Alrammal 2009c] showing important gains when replacing stream-querying with stream-scanning.

To optimize the stream-querying process based on the resources (values of time/memory) imposed by the end user, we need to determine the values of the searching range (*StartLT*, *EndLT*).

Deciding the searching range (*StartLT*, *EndLT*)

As we mentioned before, our mathematical model consists of many linear regression functions, $y = ax + b$. The terms used in this section *StartLT*, *EndLT*, *Buffer*, *Cache*, *MaxTime*, and *MaxMemory* were already defined in section 5.3.1.

To determine the searching range (*StartLT*, *EndLT*) based on the imposed value by the end user, we present the following scenarios:

1. Time: the end user imposes a limited time, e.g. the value of *MaxTime* is 15s. In this case, we calculate the values of *StartLT* and *EndLT* of the searching range as follows:

   - *EndLT*: in our mathematical model, the value of *MaxTime* is the absolute average value obtained from the following relations:

     ```
     ValueOfMaxTimeVsStartLT=((slopeMaxTimeVsStarLT)*.(StartLT)+.
     (interceptMaxTimeVsStartLT))
     ValueOfMaxTimeVsEndLT=((slopeMaxTimeVsEndLT)*.(EndLT)+.
     (interceptMaxTimeVsEndLT))
     ValueOfMaxTimeVsBuffer=((slopeMaxTimeVsBuffer)*.(Buffer)+.
     (interceptMaxTimeVsBuffer))
     ```

```
ValueOfMaxTimeVsCache=((slopeMaxTimeVsCache)*.(Cache)+.
(interceptMaxTimeVsCache))
```

Therefore, the value of *EndLT* is the absolute average integer value obtained from the following relations:

```
StartLT=(ValueOfMaxTimeVsStartLT/. slopeMaxTimeVsStartLT)-.
(interceptMaxTimeVsStartLT)
EndLT=(ValueOfMaxTimeVsEndLT/. slopeMaxTimeVsEndLT)-.
(interceptMaxTimeVsEndLT)
Buffer=(ValueOfMaxTimeVsBuffer/. slopeMaxTimeVsBuffer)-.
(interceptMaxTimeVsBuffer)
Cache=(ValueOfMaxTimeVsCache/. slopeMaxTimeVsCache)-.
(interceptMaxTimeVsCache)
```

- *StartLT*: to make sure that stream-querying process will start from the right position in the XML document, we get the value of *StartLT* from our metadata based on the XPath query sent by the end user.

```
val getStartLT->string->float
```

Once we have the values of searching range (*StartLT*, EndLT), we augment them to the end user's query to optimize the stream-querying process.

2. Memory: the end user imposes a limited memory, e.g. the value of *MaxMemory* is 15000KiB. In this case, we calculate the values of *StartLT* and *EndLT* of the searching range as follows:

   - *EndLT*: in our mathematical model, the value of *MaxMemory* is the absolute average value obtained from the following relations:

```
ValueOfMaxMemoryVsStartLT=((slopeMaxMemoryVsStarLT)*.(StartLT)+.
(interceptMaxMemoryVsStartLT))
ValueOfMaxMemoryVsEndLT=((slopeMaxMemoryVsEndLT)*.(EndLT)+.
(interceptMaxMemoryVsEndLT))
ValueOfMaxMemoryVsBuffer=((slopeMaxMemoryVsBuffer)*.(Buffer)+.
(interceptMaxMemoryVsBuffer))
ValueOfMaxMemoryVsCache=((slopeMaxMemoryVsCache)*.(Cache)+.
(interceptMaxMemoryVsCache))
```

Therefore, the value of *EndLT* is the absolute average integer value obtained from the following relations:

```
StartLT= (ValueOfMaxMemoryVsStartLT/. slopeMaxMemoryVsStartLT)-.
(interceptMaxMemoryVsStartLT)
EndLT= (ValueOfMaxMemoryVsEndLT/. slopeMaxMemoryVsEndLT)-.
(interceptMaxMemoryVsEndLT)
```

```
Buffer= (ValueOfMaxMemoryVsBuffer/. slopeMaxMemoryVsBuffer)-.
(interceptMaxMemoryVsBuffer)
Cache= (ValueOfMaxMemoryVsCache/. slopeMaxMemoryVsCache)-.
(interceptMaxMemoryVsCache)
```

- *StartLT*: to make sure that stream-querying process will start from the right position in the XML document, we get the value of *StartLT* from our metadata based on the XPath query sent by the end user.

```
val getStartLT->string->float
```

Once we have the values of searching range (*StartLT*, EndLT), we add them as metadata to the end user's query to optimize the stream-querying process.

3. Time and Memory: the end user imposes a limited time, e.g. $MaxTime$ is 15s, and he imposes a limited memory e.g. *MaxMemory* is 15000KiB. In this case, we calculate the values of *StartLT* and *EndLT* of the searching range as follows:

- *EndLT*: to find a solution for this case, we calculate the values of $MaxTime$ and $MaxTime$ in the same way that we explained before. Then, the new value of *EndLT* will the *min* value of *EndLT* ($MaxTime$) and *EndLT* ($MaxMemory$)

EndLT= min (EndLT($MaxTime$)) (EndLT($MaxMemory$))

- *StartLT*: to make sure that stream-querying process will start from the right position in the XML document, we get the value of *StartLT* from our metadata based on the XPath query sent by the end user.

```
val getStartLT->string->float
```

Below, we present the interactive mode with the end user.

Interactive mode with the end user

Once the mathematical model is built and the metadata is stored, the end user can interact with our model as it is explained in algorithm 11.

Algorithm 11: User Protocol (Interactive Mode)

```
while (true) do
    Enter: the XPath expression (query) and the name of the data set
    if (The predicted time/memory satisfy the end user) then
        The stream-querying algorithm will be used to process the query and to return a complete answer for
        the end user
    else
        Optimize you query by imposing further constraints
        if (Constrain = Time) then
            Press 1, then, enter the value of MaxTime in second.    /*Calculating the value of the
            searching range*/
        if (Constrain = Memory) then
            Press 2, then, enter the value of MaxMemory in KiB.    /*Calculating the value of the
            searching range*/
        if (Constrain = Time and Memory) then
            Press 3, then, enter the value of MaxMemory in KiB, and enter the value of MaxTime in second
            /*Calculating the value of the searching range*/
        The value of the searching range (StartLT and EndLT) is calculated, and it is augmented to the end
        user's query which will be processed by the stream-querying semi-algorithm.
```

An example of using our user protocol is as follows:

1. The end user inserts the name of data set to construct the mathematical model and to store the metadata, see figure 5.19 (this process occurs only one time).

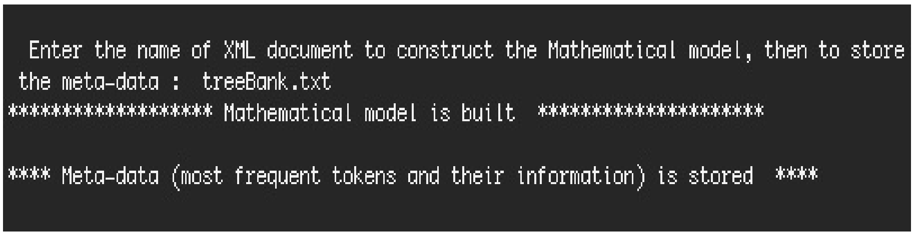


Figure 5.19: Building the mathematical Model

2. Then, he is asked to insert the name of data set and the XPath query concerned to execute his search. He will receive a massage from the system in the expected time/ space to process the XPath. He has the option to accept/refuse this cost, see figure 5.20.

3. In our case, we suppose that the end user did not accept this cost, therefore he chose the option *No* to optimize his search, see figure 5.21.

4. The end user decided to optimize the time, therefore he pressed 1, then he imposed a new value for the maximum time, see figure 5.22.

   The value of maximum time imposed by the end user is 0.004s. As we see in the figure above, the value of *MaxTime* was reduced from 0.009s to 0.004s. Also, *NumberOfMatches* reduced from 9 matches to 5 matches.

```
 *** Please enter the information needed as it is mentioned below ***
 Enter the name of XML document :  treeBank.txt
Enter the XPath expression  :   //PP/TO


The  predicted value of MinTIme is  0.0015 s
  The  predicted value of AvgTIme is  0.0048 s
  The  predicted value of MaxTIme is 0.0092 s
  The  predicted value of MinMemory is  406.80  KB
  The  predicted value of AvgMemory is  970.62  KB
  The  predicted value of MaxMemory is  1791.97  KB
  --> Do you accept this cost (Yes/No)
```

Figure 5.20: Predicting the cost of the XPath query sent

```
  --> Do you accept this cost (Yes/No)
No
--> You can impose your constraints (Time/Memory) to process //PP/TO
 ----> If your constraint is:
 ------> Time, enter 1
------> Memory, enter 2
------> Time and Memory, enter 3
```

Figure 5.21: Refusing the predicted query cost

```
  --> Do you accept this cost (Yes/No)
No
--> You can impose your constraints (Time/Memory) to process //PP/TO
 ----> If your constraint is:
 ------> Time, enter 1
------> Memory, enter 2
------> Time and Memory, enter 3
1
--> Please enter the value of maximum time in seconds
.004
 ****************************Measured query evaluation cost ******************************
  The measured value of MinTime is : 0.0004  s
 The measured value of AvgTime is : 0.0020  s
 The measured value of MaxTime is : 0.0041  s
 The measured value of MinMemory is : 53.77  KB
 The measured value of AvgMemory is : 282.32  KB
 The measured value of MaxMemory is : 603.92  KB
 The  measured number of matches  is : 5 token
```

Figure 5.22: Optimizing the query by imposing constraints

5.3.5 Experimental Results

In this section, we demonstrate the accuracy of our model by using variety of XML data sets. In addition, we examine its efficiency and the size of the training set and metadata that it requires. For example: the latter should not be too large in practice and we observe that our model behaves favorably in this respect. Metadata build from frequent (repeated 3 times or more) element names in the document occupies only 1/2000th of the document size, and this is confirmed for tests on documents differing in content, structure and size.

5.3.5.1 Experimental Setup

We performed experiments on a MacBook with the following technical specifications: Intel Core 2 Duo, 2.4 GHz, 4 GB RAM. Then, we checked the portability of the model to Red hat Linux with the following specifications: Intel Xeon 2.6 GHz, 8 GB RAM.
We used synthetically generated data sets and data sets from a real-world application [Suciu 1992]. See table 5.11. The functional language OCaml version 3.11 was used on both machines.

	Synthetic	Synthetic	TreeBank	TreeBank
Structure	wide and shallow	wide and shallow	narrow deep and recursive	narrow deep and recursive
Data Size	43MiB	1GiB	64KiB	146MiB
Max./Avg Depth	10/4.5	10/4.5	36/7.6	36/7.6

Table 5.11: Characteristics of the experimental data sets.

The average relative error was used to measure the quality of the model prediction, it is defined as follows: $\frac{1}{n}\sum_i^n |\frac{M_i - P_i}{M_i}|$, where M_i is the measured value of the $i-$th query in the workload and P_i is its predicted one.

In the first part of our experiments, we measured the quality of the model prediction (section 5.3.5.2). While in the second part (section 5.3.5.3), we presented the impact of using metadata in our model on the performance. Then we checked the portability (section 5.3.5.4).

5.3.5.2 Quality of Model Prediction

To test the quality of the model prediction we performed several experiments for measuring space prediction. We measured the space prediction of the model without the interaction of the end user, in this syntax, the end user accepts the predicted query evaluation cost (predicted allocated memory) and does not impose any constraint. Then, the quality of space prediction was measured once the end user imposes memory constraints to optimize his search.

Though the purpose of our model is to measure the space prediction, we also presented an attempt to measure the time prediction for an XPath given query.

Space Prediction

1. No interaction between the end user and the model:

   The quality of prediction (error percentage) of our model was measured for both real and synthetic data sets which have different sizes and structures (see table 5.11).

End-user query	Predicted MaxMemory MiB	Measured MaxMemory MiB	Percentage of error MaxMemory
//PP/TO	1.624	1.563	3.848
//NP/NNP	1.655	1.631	1.456
//NP/CC	1.641	1.604	2.287
//EMPTY/_PERIOD_	1.625	1.563	3.958
//VP/VB	1.626	1.574	3.263
//NP[./CC]	1.641	1.603	2.378
//VP[./VB]	1.626	1.573	3.349
//NP/SBAR/S	1.671	1.607	4.032
//VP/ADJP/JJ	1.639	1.565	4.709
//ADJP/PP/IN	1.650	1.556	6.006
//S/VP/NP	1.738	1.633	6.456
	Error average		3.795

(a) TreeBank 64KiB

End-user query	Predicted MaxMemory GiB	Measured MaxMemory GiB	Percentage of error MaxMemory
//ADVP-3/RB	4.367	4.006	9.017
//ADJP/FW	4.398	4.011	9.660
//NP-1/_DOLLAR_	4.423	4.008	10.36
//PNP/POSS	4.112	4.002	2.755
//PNP[./POSS]	4.112	4.120	0.195
//ADVP-3[./RB]	4.367	4.006	9.017
//SBAR-2/WHNP/WDT	5.092	4.003	27.20
//EMPTY/SIINV/_NL_	4.447	4.016	10.73
	Error average		9.866

(b) TreeBank 146MiB

End-user query	Predicted MaxMemory GiB	Measured MaxMemory GiB	Percentage of error MaxMemory
//A1//B3	20.51	19.22	6.708
//B3//C4	21.14	19.48	8.519
//E6//G3	20.43	19.20	6.401
//F1[.//H5]	20.54	19.08	7.633
//A2[.//C4]	21.09	19.09	10.53
//C8/D5/E5	20.00	19.12	4.628
//D5//E7//H4	20.69	19.15	8.077
//B4//E3//G7	20.44	19.10	7.009
//E5//F7/G3	20.52	19.10	7.435
	Error average		7.437

(c) Synthetic 1GiB

Figure 5.23: Percentage error for space prediction

In figure 5.23(a), we tested our model by using the real data set TreeBank which has a narrow and deep structure and a size of 64KiB. As can be seen, the error average for space is 3.80%. Our motivating scenario is to process large XML documents and considering the memory allocated to model the cost for a given XPath query. Therefore, in figure 5.23(b), we tested our model with the data set TreeBank which has the size of 146MiB. The error average of space is 9.87%. In addition, in figure 5.23(c) the model was tested by using a synthetic data set which is wide, shallow and it has a size of 1GiB. The error average of space is 7.44%.

2. Interaction between the end user and the model:

   As we mentioned above, the end user can impose certain constraints to optimize his queries by using the user protocol. We measured the quality of prediction of user protocol by using the data set TreeBank 146MiB.

End-user query	Imposed	Measured	Percentage of error
	MaxMemory GiB	MaxMemory GiB	MaxMemory
//SQ/AUX	2.222	1.907	16.54
//WHADVP/WHADVP	3.197	2.711	17.91
//WHNP/_LRB_	3.959	4.218	6.132
//X-2[./_COLON_]	1.311	1.080	21.47
//X[./CONJ-5]	1.610	1.309	22.94
//EMPTY/SIINV/_NL_	3.292	2.711	21.43
//SBAR/WHNP[./WDT]	4.921	4.002	22.97
	Error average		18.48

(a) TreeBank 146MiB

End-user query	Imposed	Measured	Percentage of error
	MaxMemory GiB	MaxMemory GiB	MaxMemory
//A1//B3	0.954	0.887	7.478
//B3//C4	11.99	10.92	9.786
//E6//G3	0.942	0.881	6.986
//F1[.//H5]	17.68	16.43	7.634
//A2[.//C4]	0.114	0.103	10.56
//C8/D5/E5	1.593	1.474	8.120
//D5//E7//H4	0.666	0.615	8.299
//B4//E3//G7	11.00	10.23	7.510
//E5//F7/G3	1.442	1.342	7.439
	Error average		8.201

(b) Synthetic 1GiB

Figure 5.24: User Protocol -
Percentage error for space prediction

   Figure 5.24(a) illustrates the percentage of error between the values imposed by the end user and the measured ones by the model for certain queries. As can be seen, the error average for space is 18.48%.

   We also tested the user protocol by using the synthetic data set 1GiB. Figure 5.24(b) illustrates the percentage of error for this test. The error average of space is 8.2%.

Time Prediction

To test the time prediction of our model, we used the same real data sets which we used in the previous experiments for space prediction. We first measured the

End-user query	Predicted	Measured	Percentage of error
	MaxTime Second	MaxTime Second	MaxTime
//PP/TO	0.038	0.035	8.857
//NP/NNP	0.039	0.040	2.500
//NP/CC	0.038	0.038	0.000
//EMPTY/_PERIOD_	0.039	0.036	8.333
//VP/VB	0.038	0.038	0.000
//NP[./CC]	0.039	0.039	0.000
//VP[./VB]	0.038	0.042	9.524
//NP/SBAR/S	0.041	0.040	2.500
//VP/ADJP/JJ	0.039	0.040	2.500
//ADJP/PP/IN	0.040	0.039	2.564
//S/VP/NP	0.045	0.040	12.22
	Error average		4.454

(a) TreeBank 64KiB

End-user query	Predicted	Measured	Percentage of error
	MaxTime Seconds	MaxTime Seconds	MaxMemory
//ADVP-3/RB	21.47	12.19	76.04
//ADJP/FW	22.16	12.36	79.22
//NP-1/_DOLLAR_	22.92	12.43	84.44
//PNP/POSS	14.42	12.20	18.23
//PNP[./POSS]	14.42	13.56	6.354
//ADVP-3[./RB]	21.47	13.46	59.48
//SBAR-2/WHNP/WDT	38.01	14.90	155.2
//EMPTY/SIINV/_NL_	23.41	13.14	78.18
	Error average		69.64

(b) TreeBank 146MiB

Figure 5.25: Percentage error for time prediction

quality of prediction by using the real data set TreeBank which has a narrow and deep structure and a size of 64KiB.

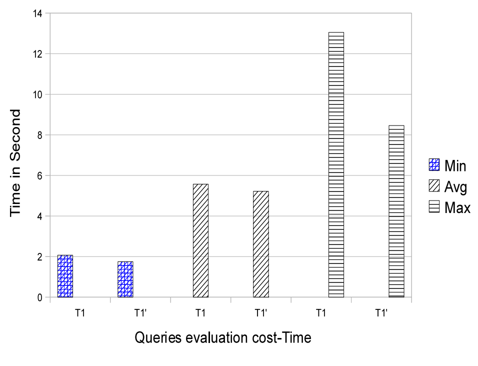
Figure 5.25(a) illustrates the percentage error for time prediction by using the data set TreeBank 64KiB. The error average of time for this experiment is 4.45%. In figure 5.25(b) we measured the error percentage for time prediction by using the data set TreeBank 146MiB. As can be seen, the error average of time is 69.64%.

5.3.5.3 Impact of Using Metadata in our Model on the Performance

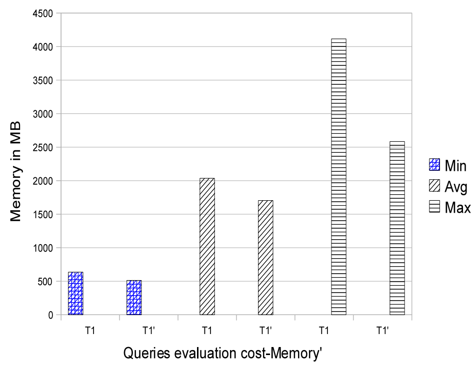
Improving the Performance by Using Searching range

To show the efficiency of a restricted searching range compared to the existing exhaustive stream-querying algorithm LQ, we performed two type of tests on the data set TreeBank 146MiB, these tests are:
– T1: queries were sent without searching range.
– T1′: same queries were sent with searching range obtained from T1 to demonstrate the time/memory gain possible.

Figure 5.26(a) shows the query evaluation costs (time spent) of T1 and T1′. The *MinTime* of T1 is 2.07s while for T1′ it is 1.75s. The 15% gain in time is due to *stream-scanning*: [3] until reaching the *StartLT* point, thus avoiding unnecessary buffering and caching processes. The *AvgTime* of T1 is 5.57s while for T1′ is 5.22s, the slight gain of time occurred because the gain of time of *MinTime* affects positively the value of *AvgTime*. The *MaxTime* for T1 is 13.05s while for T1′ it is 8.46s. The gain of 37% in time is due to both *StartLT* and *EndLT* restricting the searching range so that stream-querying process stops the moment *EndLT* is reached. This is correct because we know that there will be no any further possible matches in the XML document.



(a) T1 and T1′ -Time



(b) T1 and T1′-Memory

Figure 5.26: Query evaluation cost for T1 and T1′

Figure 5.26(b) shows the query evaluation cost (memory used) of T1 and T1′. The *MinMemory* of T1 is 635MiB while for T1′ it is 512MiB the gain of memory which is 20% was obtained because of the stream-scanning technique which scans

[3] to process the XML data stream with minimal resources, this process simply searches the position of a specific element in D without caching nor buffering.

the XML document until *StartLT* is reached. The *AvgMemory* of T1 is 2035MiB while for T1′ it is 1703MiB, a gain of 17% in memory is obtained because the gain on *MinMemory* affects positively the value of *AvgMemory*. The *MaxMemory* value for T1 is 4115MiB while for T1′ it is 2584MiB. The gain of 38% in memory is due to the use of both *StartLT* and *EndLT* to delimit the searching range. Thus we stop the stream-querying process the moment *EndLT* is reached because we know that there will be no any further possible matches in the document.

Negative Queries

In this section, we present the impact of using metadata on the measured time/memory for the negative queries. As we mentioned in 5.3.3.1 (prediction rules), frequent negative element names help us to decide in advance that certain queries are negative. This property which exists in the model improves the performance.

To show the efficiency of our model compared to the existing exhaustive stream-querying algorithm LQ, we performed two type of tests on the data set TreeBank 146MiB, these tests are:
– T2: negative queries were sent without metadata.
– T2′: same queries were sent with metadata obtained from T2 to demonstrate the possible gain of time/memory for negative queries.

Figure 5.27 shows the query evaluation costs (time spent) of T2 and T2′. The values of *MaxTime* of T2 and T2′ for the first 5 queries are equal 69.5s, because the model still did not detect any frequent negative element name. The values of *MaxTime* of T2 for the first 10 queries is 134.12s while for T2′ is 122s, the time improvement of T2′ occurred because the model detected in advance that one query is negative. The values of *MaxTime* of T2 for the first 15 queries is 207.5s while for T2′ is 160.2s, the time improvement of T2′ occurred because the model detected in advance that three queries are negative. The values of *MaxTime* of T2 for the all queries is 268.35s while for T2′ is 187s, the time improvement of T2′ which is 30% occurred because the model detected in advance that six queries are negative.

Figure 5.28 shows the query evaluation costs (memory used) of T2 and T2′. The values of *MaxMemory* of T2 and T2′ for the first 5 queries are equal 20.11GiB, because the model still did not detect any frequent negative element name. The values of *MaxMemory* of T2 for the first 10 queries is 40.13GiB while for T2′ is 36.13GiB, the memory improvement of T2′ occurred because the model detected in advance that one query is negative. The values of *MaxMemory* of T2 for the first 15 queries is 60.31GiB while for T2′ is 48.14GiB, the memory improvement of T2′ occurred because the model detected in advance that three queries are negative. The values of *MaxMemory* of T2 for the all queries is 80.42GiB while for T2′ is 56.21GiB, the memory improvement of T2′ which is 30% occurred because the model detected in advance that six queries are negative.

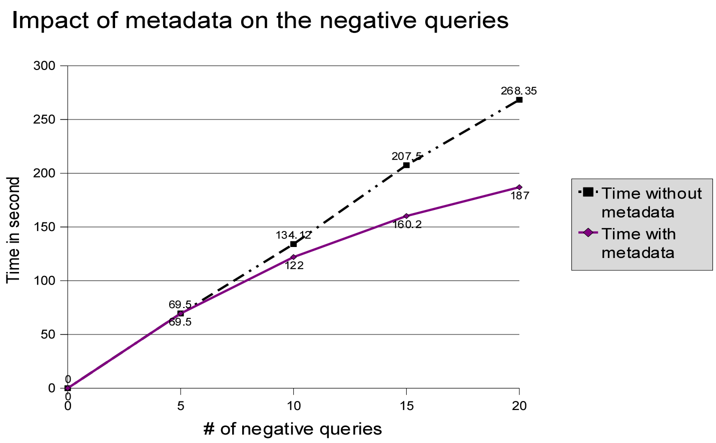


Figure 5.27: Impact of metadata on time for T2 and T2′

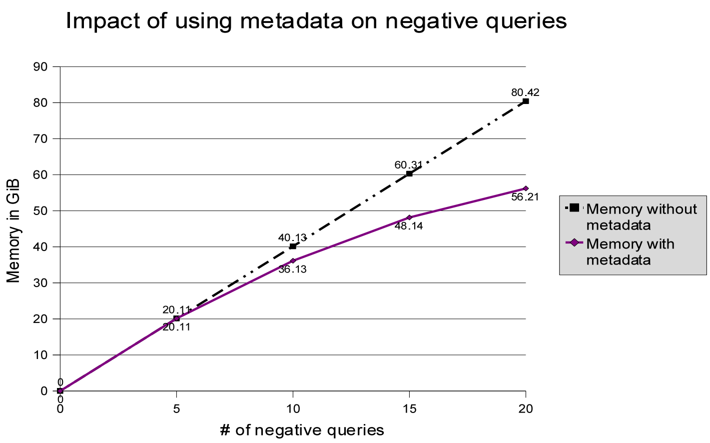


Figure 5.28: Impact of metadata on memory for T2 and T2′

5.3.5.4 Model Portability on Other Machines

To check the portability of our model, we rebuilt it on another machine: Red hat Linux with the following specifications: Intel Xeon 2.6 GHz, 8 GiB RAM.

We used the same byte-code object file, the data set TreeBank 64KiB and the same queries which we already used in the test of figure 5.23(a).

Figure 5.29 illustrates the error average for space that is 3.79%, it is the same result as the test in figure 5.23(a).

Hence:
–$\text{Memory}_{\text{linux}} = \text{Memory}_{\text{mac}}$
– $\text{Time}_{\text{linux}} = \text{Time}_{\text{mac}}/1.18$

which is a stable factor for porting our model on another machine *without rebuilding it*.

End-user query	Predicted MaxMemory MiB	Measured MaxMemory MiB	Percentage of error MaxMemory
//PP/TO	1.624	1.563	3.848
//NP/NNP	1.655	1.631	1.456
//NP/CC	1.641	1.604	2.287
//EMPTY/_PERIOD_	1.625	1.563	3.958
//VP/VB	1.626	1.574	3.263
//NP[./CC]	1.641	1.603	2.378
//VP[./VB]	1.626	1.573	3.349
//NP/SBAR/S	1.671	1.607	4.032
//VP/ADJP/JJ	1.639	1.565	4.709
//ADJP/PP/IN	1.650	1.556	6.006
//S/VP/NP	1.738	1.633	6.456
	Error average		3.795

Figure 5.29: Percentage error- TreeBank 64KiB(Linux)

5.3.6 Conclusion

In this section we presented our performance prediction model - simple path. The model allows static *a priori* prediction of time-space parameters on a given (variable) query for a given (fixed) XML data set. It proceeds by accumulating information from training queries whose node tests are those frequently found in the target document. Two specific objectives for our model were:

1. to obtain reliable and portable cost predictions for random queries on a fixed data set, while storing a small amount of metadata.

2. to use the predictions to improve performance and/or resource management.

Our objectives are attained for any structure/size of XML documents and over both time/memory. Two improvements over the computing approach COMET [Zhang 2005] have been achieved. However our current system covers a smaller fragment of XPath.

Our optimizations are also novel: they are obtained by using searching ranges to alternate between stream-scanning and stream-querying. The gain of *MaxTime* reached up to 38%, while the gain of *MaxMemory* reached 37%.

Our current non optimized model building processes from 100 to 1000 elements by second which is maybe slow for very large XML documents.

As future work, we aim to extend and improve the performance model by considering a larger fragment of XPath to include all of the Forward XPath defined in 1. To ensure accurate XML path selectivity estimation, our mathematical model and metadata must be updated once the underlying XML data change. To avoid reconstructing the mathematical model by using the off-line periodic scan, we will investigate how to automatically adapts to changing XML data by using the queries feedback (online algorithm for model construction). All these points are processed in the next section 5.4 which explains the performance prediction model - twig path.

5.4 Performance Prediction Model - Twig Path

The performance prediction model - twig path is a cost model which estimates the cost (in terms of space used and time spent) for any structural XPath query belongs to Forward XPath (defined in section 1.1.1.2).

Figure 5.15 illustrates our performance prediction (cost) model - twig path.

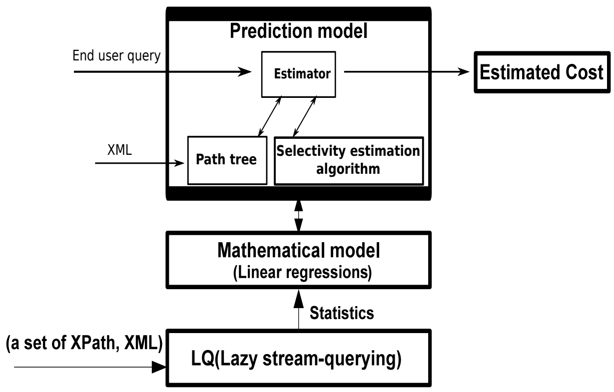


Figure 5.30: Layers of our performance prediction model - twig path

To built this model, we need a stream-querying algorithm to send training queries (a set of XPath queries) on the target XML document in order to get on the statistics needed. We therefore used our extended lazy stream-querying algorithm that is defined in chapter 4.2 (layer 1 of figure 5.30). After that, statistics are used to build a mathematical model which consists of a set of linear regression functions that will be used to estimate the cost for a given XPath query (layer 2 of figure 5.30).

The path tree is built for the target XML document by using our streaming algorithm (the details of the construction process of the path tree are explained in chapter 3). After that, the moment the end user send an XPath query, the function *estimator* analyses it and estimates the values of the input parameters of the mathematical model by using the path tree and the selectivity estimation algorithm that is defined in chapter 4.3 (layer 3 of figure 5.30). *estimator* provides the end user with the estimated cost for his query (which was calculated by the mathematical model).

Next, we will explain in details each layer of the model.

5.4.1 Lazy Stream-querying Algorithm (LQ)

We used our stream-querying algorithm that is defined in chapter 4.2 to get on the statistics needed to build the mathematical model.

The current extended version of LQ processes queries which belong to the fragment of Forward XPath. Our algorithm was implemented using the functional language OCaml release 3.11 [Leroy 2010b] which combines relatively high performance with strong typing and ML-language constructs for tree processing.

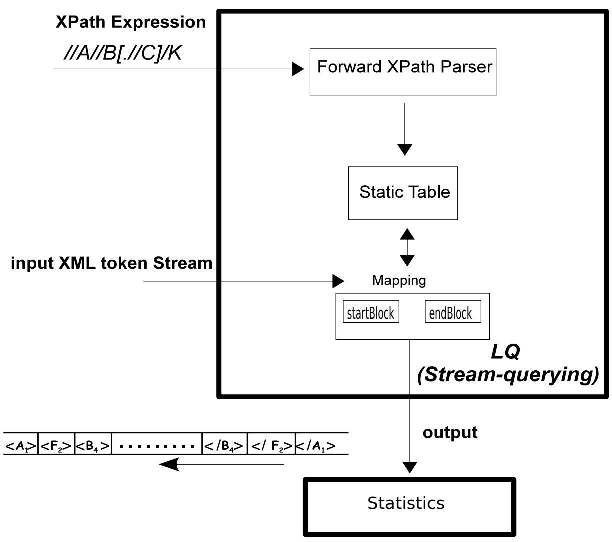


Figure 5.31: Extended LQ (Lazy stream-querying algorithm)

Our extended LQ takes two input parameters (see figure 5.31). The first one is the XPath query (which belongs to Forward XPath to allow stream-processing) that will be transformed to a query table statically using our Forward XPath Parser. After that, the main function is called. It reads the second parameter (XML document in SAX events syntax) line by line repeatedly, each time generating a tag. Based on that tag a corresponding *startBlock* or *endBlock* function is called to process it. Finally, the main function generates as output the result for the sent XPath (statistics).

Statistics consist of:

1. *NumberOfMatches*: is the number of answer elements found during processing of the XPath query Q on the XML document D.

2. *Cache*: is the number of elements cached in the run-time stacks during processing of the XPath query Q on the XML document D. They correspond to the axis nodes of Q.

3. *Buffer*: is the number of potential answer elements buffered during processing of the XPath query Q on the XML document D.

4. *OutputSize*: is the total size in MiB of the number of answer elements found during processing of the XPath query Q on the XML document D.

5. *WorkingSpace*: is the total size in MiB for the number of elements cached in the run-time stacks and the number of potential answer elements buffered during processing of the XPath query Q on the XML document D.

6. *NumberOfPredEvaluation*: is the number of times the query's predicates are evaluated (their values are changed or passed from an element to another).

In the next section, we explain the construction process of the mathematical model.

5.4.2 Building the Mathematical Model

As illustrated in figure 5.32, the first step is to send training queries to collect the information needed (statistics) by using our extended stream-querying algorithm LQ. These statistics will be stored in a hash table.

We call our technique for sending training queries and collecting the statistics by *partial testing*: a process to test some not-repeated XPath queries existing in the data set. In our model, the number of the training queries = p^2, where p is the number of the input parameters of the mathematical model which is 6. These parameters are: *NumberOfMatches*,*Cache*, *Buffer*, *OutputSize*, *WorkingSpace*, *WorkingSpace*, *NumberOfPredEvaluation*.

The moment we have this information, we use them to build the mathematical model. The model consists of a set of linear regressions, they are:

- *MaxTime* vs (*Buffer*, *Cache*, *NumberOfMatches*, *OutputSize*, *WorkingSpace*, *NumberOfPredEvlaution*).

- *MaxMemory* vs (*Buffer*, *Cache*, *NumberOfMatches*, *OutputSize*, *WorkingSpace*,*NumberOfPredEvlaution*).

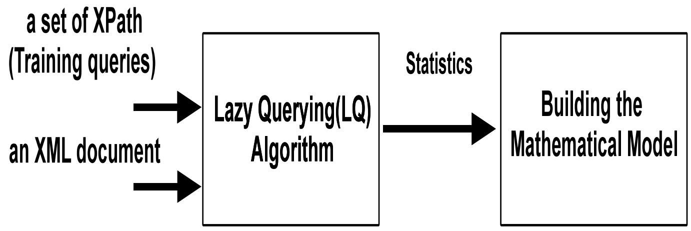


Figure 5.32: Building the Mathematical Model

To build a part of the mathematical model (the linear function) which will be used to estimate the value of *MaxMemory*, we linearize the *MaxMemory* vs (*Buffer*, *Cache*, *NumberOfMatches*, *OutputSize*, *WorkingSpace*, *NumberOfPredEvlaution*). The same process is applied on *MaxMemory* to obtain the complete mathematical model. For example: to linearize *MaxMemory* vs *Buffer*, we calculate the slope and intercept of this relation.

In the next section 5.4.3, we explain how the prediction model uses these linear functions to estimate the cost for a given XPath query.

5.4.3 Building the Prediction Model

As illustrated in figure 5.33, the end user sends his/her XPath query to the prediction model which was constructed for the XML document *D*. The *estimator* analyses the XPath query and uses the path tree of *D* (that is introduced in chapter 3) and the selectivity estimation algorithm (that is introduced in chapter 4) to estimate the values of the input parameters of the mathematical model. These input parameters are: *Buffer*, *Cache*, *NumberOfMatches*, OutputSize, *WorkingSpace* and *NumberOfPredEvlaution*. The last parameter used if the XPath query contains any predicates.

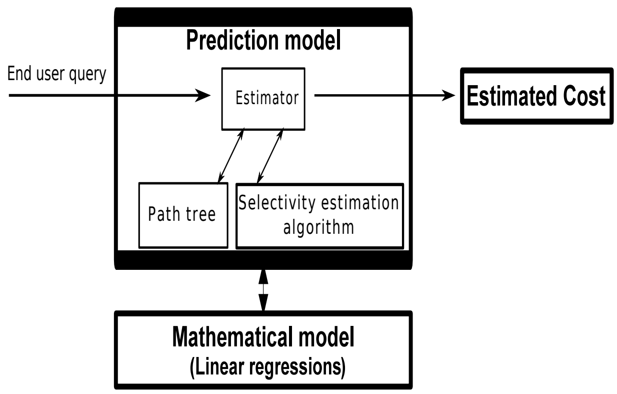
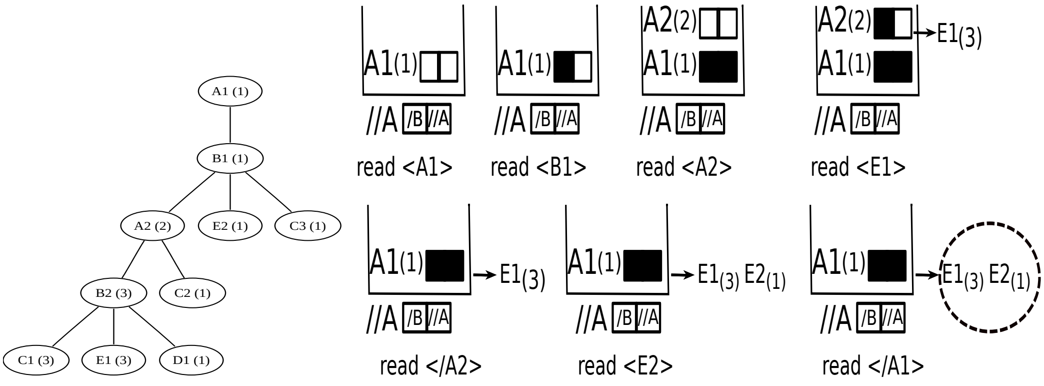


Figure 5.33: Building the Prediction Model

Each value of an input parameter will be used by its corresponding linear regression function in the mathematical model. The average of the linear regressions



(a) path tree of D (b) Snapshots of the run-time stacks for the evaluation process

Figure 5.34: Snapshots of the run-time stacks for the evaluation of the path tree of D on Q ($//A[./B \textit{ and } .//A]//E$)

results is calculated to estimate the cost for a given XPath query. The cost estimated for a given XPath query is: *MaxTime* and *MaxMemory*.

Though we explained in details the selectivity estimation process in chapter 4), below we introduce an example which explains how to get the values of the input parameters for the mathematical model.

5.4.3.1 Example of the Selectivity Estimation Process

Figure 5.34(b) illustrates different snapshots of the evaluation process of the path tree of D on the twig path $//A[./B \textit{ and } .//A]//E$ which returns $E1(3)$, $E2(1)$ as result nodes. For each non-leaf node, the algorithm creates a stack. Therefore, in this example, a stack is created for the root node A.

When $< A1 >$ is read, the function *startBlock* is called in the post order of A in Q, that is 3,1. The predicate node $A1$ with order 3 is not evaluated because its parent stack (stack A) is empty. While $A1$ with order 1 is pushed (with its information) in its corresponding stack A with false values for its both predicate nodes B and A. Moreover, the values of *Cache* and *WorkingSpace* are updated.

When $< B1 >$ is read, $B1$ is a direct child for node $A1$, therefore the value of the predicate B of the node $A1$ is changed from false to true. The value of the *NumberOfPredEvlaution* is updated.

When $< A2 >$ is read, the function *startBlock* is called in the post order of A in Q, that is 3, 1. The value of the predicate node A with order 3 for $A1$ is changed from false to true because its parent stack (stack A) is not empty, it contains the node $A1$. While $A2$ with order 1 is pushed (with its information) in its corresponding stack A with false values for its both predicate nodes B and A. Moreover, the values of *Cache*, *WorkingSpace* and *NumberOfPredEvlaution* are updated.

When $< E1 >$ is read, as long as it is a descendant of $A2$, the node $E1$ is buffered (with it information) to the potential answers list of its parent node $A2$.

When $< /A2 >$ is read, it is popped out from its stack. $A2$ is the root node, but its predicate node A is not satisfied, therefore, the function *appendOrDestroy* is called. The host stack of A is the stack A itself ($host[A] = A$), as long as this stack is not empty (it contains node $A1$), the potential answers list of $A2$ is appended to the same list of $A1$.

When $< E2 >$ is read, as long as it is a descendant of $A1$, $E2$ is buffered (with it information) to the potential answers list of its parent node $A1$.

Finally, when $< /A1 >$ is read, it is popped out from its stack. $A1$ is the root node, as long as the values of its predicates B and A are true, then, the content of its potential answers list ($E1(3)$ and $E2(1)$) is flushed as a final answer.

The result of the XPath query estimation is as follows (estimated values): $NumberOfMatches$: the value is 4 , they are $E1(3)$, $E2(1) = 3 + 1 = 4$. $Buffer$ is this example, the $Buffer$ has the same value as the the value of $NumberOfMatches$ that is 4. *Cache*: the value is 3, they are $A1(1)$, $A2(2)$ $= 1+2 = 3$. *WoringSpace*: its size was estimated to $= (22+44)+(66+22) = 154$ byte= 0.0001MiB. *OutputSize*: its size was estimated to 88 byte = 0.00008 MiB. *NumberOfPredEvlaution*: the value is 6, they are $B1(1)$, $A2(2)$, $B2(3)$ $= 1+2+3 = 6$.
Each value of an input parameter (an estimated value) will be used by its corresponding linear regression function in the mathematical model to estimate the cost for a given XPath query. The cost estimated for a given XPath query is: *MaxTime* and *MaxMemory*.

5.4.4 Experimental Results

In this section, we demonstrate the accuracy of our system by using variety of XML data sets and complex queries. Furthermore, we show the efficiency of our modified LQ algorithm. Finally, we compare our approach with other approaches.

5.4.4.1 Experimental Setup

We performed experiments on a MacBook with the following technical specifications: Intel Core 2 Duo, 2.4 GHz, 4 GB RAM. The well known XML data sets XMark [Schmidt 2001] and TreeBank [Suciu 1992] were selected for the experiments. XMark is a wide and shallow data set, its size is 116MiB and its maximum depth is 12. TreeBank is a deep and recursive data set, its size is 86MiB and its maximum depth is 36. The average relative error was used to measure the accuracy of our approach, it is defined as follows: $\frac{1}{n}\sum_{i=1}^{n}|\frac{M_i - P_i}{M_i}|$, where M_i is the measured value of the $i-$th query in the workload and P_i is its predicted one.

Extensive testing and complex queries were used in our experiments. Queries include: $'/'$, $'//'$, $'*'$, same node-labels, $'text()'$, predicates with *and*, *or*, *not* and nested predicates. An example for a complex XPath query taken from XMark $//item[.//payment\ or\ .//shipping]//mailbox//mail[./date]/to$ and from TreeBank $//EMPTY[./S//NP[./*]\ and\ .//VP]//*/NNS$.

5.4.4.2 Accuracy of the Selectivity Estimation

Data set	*NumberOfMatches*	*Buffer*	*Cache*	*OutputSize*	*WorkingSpace*	*MaxTime*
XMark	2.6%	0%	0%	2.5%	0%	7%
TreeBank	9.8%	8.5%	3%	9.5%	8%	19.8%

Table 5.12: Average relative error

Table 5.12 shows the estimation accuracy of path tree for complex queries. The estimation accuracy on both data sets XMark and TreeBank is remarkable (see table 5.12) due to the structure of the path tree which captures the recursions in the data set and due to the efficiency our modified LQ algorithm which supports the complete Forward XPath fragment.

5.4.4.3 Efficiency of the Selectivity Estimation Algorithm

To evaluate the efficiency of our modified LQ algorithm, we calculated the average time spent on estimating the selectivity and the average time spent on actually evaluating the XPath queries. The average ratio of the estimation time to the actual querying time on TreeBank (86MiB) and XMark (116MiB) are 13% and 0.00007% respectively.

5.4.4.4 Comparing our Approach with the other Approaches

Criteria		Synopsis		
		TreeSketch	XSeed	Path tree
Construction Time	XMark(116MiB)	681min	1min	1min
	TreeBank(86MiB)	>4	>4	244Min
Download Bandwidth	XMark(116MiB)	0.003MiB/s	1.93MiB/s	1.93MiB/s
	TreeBank(86MiB)	0.00004MiB/s	0.00004MiB/s	0.006MiB/s
Recursion in XML		No	Yes	Yes
Incremental Update		No	No	Possible

Table 5.13: Comparison of the selectivity estimation techniques

- **Construction time**: TreeSketch builds its synopsis in two steps. First, It creates an intermediate count-stability (C-stability) synopsis that preserves all the information of the original XML data set in a compact format. After that, the Tree-Sketch synopsis is built on top of the C-stability synopsis by merging similar structures.
  The XSeed synopsis consists of two parts, an XSeed kernel and a hyper-edge table (HET). The construction of HET is performed by gradually extracting

irregular structures out of the data set. The HET construction stops when it determines that no further improvement can be made. On the contrary of the above structural synopses, path tree is built in one step by one pass of the data set (in streaming). Table 5.13 shows the total construction time of TreeSktech, XSeed and path tree synopses. We do not show the construction time of the Subtree sampling synopsis because it is not a structural one, while for XCLUSTER it is unknown. The construction time of the structural synopses largely depends on the structure of the data set. Our streaming algorithm for building path tree outperforms considerably the other approaches. The construction time for each of TreeSktech and XSeed for TreeBank 86MiB (depth 36) took more than 4 days, this result was confirmed in [Luo 2009]. While for path tree, the construction time for the same data set took 244 minutes.

- **Selectivity of structural queries and synopsis size**: TreeSketch and XSeed can estimate the accuracy for the number of matches (*NumberOfMatches*), while our approach estimates the accuracy for: *NumberOfMatches*, *Buffer*, *Cache*, *OutputSize*, *WorkingSpace* and *MaxTime*. The accuracy of our approach outperforms the accuracy of TreeSketch and XSeed due to the structure of the path tree which captures the recursions in the data set and due to the efficiency our modified LQ algorithm which supports the complete Forward XPath fragment. The size of the path tree varies according to structure of the data set. It is a 10% of the size of TreeBank and a 0.00006% of the size of XMark. In all cases, an efficient streaming algorithm is used to traverse the path tree to avoid any computational overhead. Note that to control the space budget (synopsis size), it is possible to use a very partial, hence small, path tree, to use no more space than competing approaches, but the accuracy of selectivity estimation will then be much lower.

- **Recursion in the data set** : the path tree and the XSeed synopses are more general than the TreeSketch synopsis because the latter does not support the recursion in the data sets as it is explained in [Zhang 2006b].

- **The fragment of XPath**: the XPath fragment covered by our approach is more general than the one used by XSeed and TreeSketch. The TreeSketch does not support queries with Ancestor-Descendant relationships neither queries with $'text()'$ [Luo 2009]. While the XSeed does not support queries with $'text()'$ neither queries with nested predicates.

- **Download bandwidth**: for the XMark data set which has a light degree of recursion the *download bandwidth* (size of data set/construction time) of both XSeed and path tree outperform TreeSktech. As a data set become more complex, the path tree outperforms XSeed. For the data set TreeBank, the download bandwidth of path tree is 150 times faster than XSeed (see table 5.13). A study was performed in 2009 by FH SARL [FH 2009] shows that the average bandwidth for download in France is 1MiB/s. By using this average, the expected time to download the data set XMark (116MiB) is 1.93 minutes. While the expected time for the same process by using our approach is 1 minute. This means that the download bandwidth of our approach can be up to twice the average download bandwidth in France.

- **Incremental update**: minimal synopsis size seems desirable but won′t be the best because incremental maintenance would be difficult [Goldman 1997]. This is the case of both TreeSketch and XSeed. While in our approach, incremental update is possible by using the patch operations as we explained in section 3.4.1.

5.4.5 Use Case: Online Stream-querying System

In this section, we introduce the structure of the online stream-querying system through a use case.

5.4.5.1 Online Stream-querying System

Figure 5.35 illustrates the structure of the stream-querying system. Innov-Lacl: is an intermediate company between the publishers of XML data (documents) and its clients. It uses its online stream-querying system to satisfy the queries of its clients. Clients: are the clients of Innov-lacl which search for specific information. Publishers: are the providers of XML data (documents) which cooperate with Innovi-Lacl to sell access to their data.

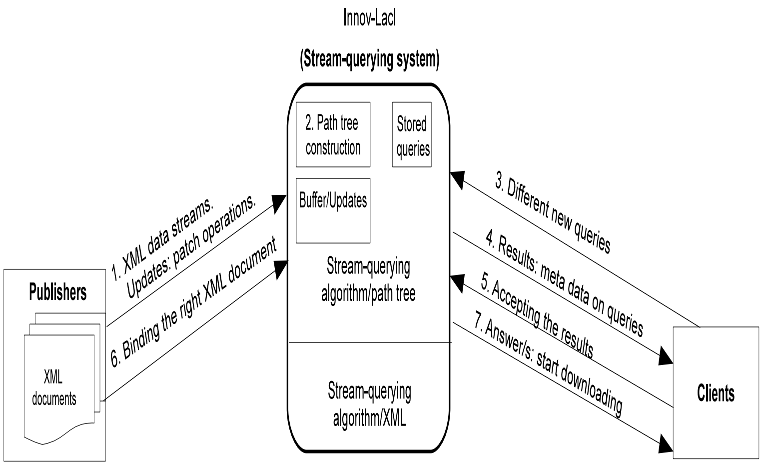


Figure 5.35: Use case - stream-querying system

The stream-querying system receives different XML documents from its publishers in streaming mode. It constructs incrementally a path tree for each XML document. It stores XPath queries from the clients of the company. It matches each stored XPath query Q with each complete/incomplete path tree using the stream-querying (for path tree) algorithm. If any matches are found in a complete/incomplete path tree with an XPath query, metadata is sent to the query's

sender informing him in: the number of matches found, the size of the output, the expected time to get the answer, and whether he accepts of refuses to get the result. In case of acceptance, our system binds the right XML document and uses our stream-querying algorithm (for XML) to provide the client in the final answers that are ready to be downloaded.
If our system is provided with the model of the XML document received, the partition process of the document to construct incomplete path trees will be more precise.

Our stream-querying system supports three scenarios for the matching process:

1. **Scenario-1** matching Q with a complete path tree:
   in this syntax, if any matches are found, the metadata sent to the query's sender will contain complete information about the answer.

2. **Scenario-2** matching Q with incomplete path tree:
   if any matches are found, the metadata sent to the query's sender will contain information about the partial answer found. e.g. if the number of matches found is 2, then, the number of matches in metadata will have the form 2+ which means that there are two matches or more.

3. **Scenario-3** path tree not yet built (worst case for our pre-processing method *i.e.* pre-processing is part of the actual XPath query processing):
   in this scenario, a message will be sent to the query's sender informing him that: – the path tree is not yet constructed – the approximated time to construct it – and whether he would to like to continue (wait for the construction process) or not.

To construct a path tree for a document D, the whole document is received (downloaded) in streaming mode and the path tree is incrementally built until the end of the stream (end of D). During the contraction process of the path tree of D, if any updates occurs on D, these updates are received as a stream of patch operations [Urpalainen 2008] and a buffer is allocated to store them. Once the construction process finishes, the updates are applied on D as we explained in section 3.4.1.

5.4.6 Conclusion and Future Work

In this section, we presented our performance prediction model - twig path. The model uses the path tree synopsis structure and the selectivity estimation algorithm for accurate XPath query selectivity estimates. Furthermore, we proposed an online stream-querying system through a use case. The system estimates the cost for a given XPath query time/memory and provides an accurate answer. Extensive experiments were performed. We considered the accuracy of the estimations, the types of queries and data sets that this synopsis can cover, the cost of the synopsis to be created, and the estimated vs measured time/memory. Experiments demonstrated that our performance prediction model is both accurate and efficient.

Probabilistic guarantees are an open problem mentioned in [Bonifati 2007]. In the future we will consider a probabilistic version of our performance prediction model.

In the next, chapter we conclude our work and present our perspectives for future research.

CHAPTER 6

Conclusion and Perspectives

Contents

6.1 Conclusion

In this thesis, we reviewed the literature and pinpointed the critical areas where we developed our research work. We justified the need for and use of a new selectivity estimation technique that is based on streaming, then, its application to performance (cost) prediction. We have identified factors that can make this process inaccurate or inefficient.

To obtain such a performance model which estimates the cost for any XPath query belonging to the fragment of Forward XPath:

- We performed an experimental study to confirm the linear relationship between the stream-processing and the data-access resources [Alrammal 2009b]. We concluded that (1) a linear regression approach can be used (in the performance prediction model) to model the cost for a given XPath query over a stream of XML data. (2) the complexity of the selectivity estimation algorithm used in a performance prediction model should not be more than linear in the size of the XML data set. Our selectivity estimation algorithm (introduced in chapter 4) does have a linear complexity.

- Then, we searched for an efficient, capable and accurate selectivity estimation technique for XPath queries, and having the following advantages: (1) fast construction for the structure synopsis, (2) to function with any data set (Size/Structure), (3) to allow the incremental update and maintenance for the structure synopsis, (4) well suited for a cost-based model, (5) and finally, accurate and time/space efficient.
In doing so we:

1. Studied in detail the path tree, a synopsis structure for XML documents that is used for accurate selectivity estimates. To the best of our knowledge, the path tree was not formally defined in the literature, but was used before in more limited ways. We formally defined it and introduced two algorithms to construct it.

2. Extended and optimized the lazy stream-querying algorithm LQ which was introduced by [Gou 2007]. The current version of the algorithm processes any XPath query belonging to the fragment of Forward XPath (that is explained in section 1.1.1.2).

3. Presented a new selectivity estimation algorithm which was inspired by our extended stream-querying algorithm LQ. Our estimation algorithm is efficient for traversing the path tree structure synopsis to calculate the estimates. The algorithm is well suited to be embedded in a cost-based optimizer, and it has a linear time cost.

   After exhaustive testing on real and synthetic data sets (e.g., TreeBank [Suciu 1992] and XMark [Schmidt 2001]), we noticed that the accuracy of our selectivity estimation technique for any path expression p is 100% correct due to the complete structure (information) of the path tree synopsis. Moreover, the selectivity estimation for twig expressions with our technique is very accurate due to the complete structure of the path tree synopsis and the efficiency of our selectivity estimation algorithm. This property is new compared to previous work.

- We presented the performance prediction model - twig path, an accurate model for stream-processing of any structural XPath query which belongs to Forward XPath. The model uses our selectivity estimation technique to measure the values of the cost-parameters which determine the cost for a given XPath query. These values are used by a mathematical model (linear regression functions) to estimate the cost for a given XPath query in terms of time spent /memory used.
  Extensive experiments were performed to evaluate our model. We considered the accuracy of estimations, the types of queries and data sets that the selectivity estimation technique can cover, the cost of the synopsis to be created, and the estimated vs measured time/memory. Experiments demonstrated that our technique is accurate.

- Finally, we presented a use case for an online stream-querying system. The system uses our performance predicate model - twig path to estimate the cost for a given XPath query in terms of time/memory. Moreover, it provides an accurate answer for the query's sender. This use case illustrates the practical advantages of performance management with our techniques.

The novel aspects of our work are:

- **Construction time**: the construction time of the structural synopses largely depends on the structure of the data set. Our streaming algorithm for building path tree outperforms considerably the other approaches. The construction time for each of TreeSktech and XSeed for TreeBank 86MiB (depth 36) took more than 4 days, this result was confirmed in [Luo 2009]. While for path tree, the construction time for the same data set took 4 hours.

- **Selectivity of structural queries and synopsis size**: some approaches as TreeSketch and XSeed can only estimate the accuracy for the number of matches (*NumberOfMatches*), while our approach estimates the accuracy for: *NumberOfMatches*, *Buffer*, *Cache*, *OutputSize*, *WorkingSpace* and *MaxTime*. The accuracy of our approach outperforms the accuracy of TreeSketch and XSeed due to the structure of the path tree which captures the recursions in the data set and due to the efficiency our modified LQ algorithm which supports the complete Forward XPath fragment.

- **Recursion in the data set** : the path tree and the XSeed synopses are more general than the TreeSketch synopsis because the latter does not support recursion in the data sets as explained in [Zhang 2006b].

- **XPath fragment**: the XPath fragment covered by our approach is more general than the one used by XSeed and TreeSketch. The TreeSketch system does not support queries with Ancestor-Descendant relationships nor queries with $'text()'$ [Luo 2009]. While the XSeed does not support queries with $'text()'$ nor queries with nested predicates.

- **Incremental update**:[Goldman 1997] estimates that *minimal synopsis size seems desirable but won't be the best because incremental maintenance would be difficult*. This is the case of both TreeSketch and XSeed. While in our approach, incremental update is possible by using the patch operations as we explained in section 3.4.1.

6.2 Future Work

There is much research to be conducted that can enhance the applicability and efficiency of the methods described in this thesis. We classify it into the following domains: stream-processing, selectivity estimation and parallel processing.

6.2.1 Stream-processing

In this domain, we would like to study the following:

- Proposing an algorithm to process a fragment of XPath larger than Forward XPath in (|D|.|Q|) time. Then comparing this algorithm with some existing work like [Nizar 2009a] which handles backward XPath axes in streaming.

The approach presented in [Nizar 2009a] processes an XPath fragment that is larger than our fragment of Forward XPath, but its complexity is unknown. We could cover more general axes than $'/'$, $'//'$ by using rewrite rules as shown in [Olteanu 2002] to reduce more general axes to forward ones when possible.

6.2.2 Selectivity Estimation Technique

In this domain, we would like to study the following:

- Compute a synopsis for a given XML document by summarizing both the structure and the content of document: a recommended way of doing this is to apply the XMill approach [Liefke 2000] in separating the structural part of the XML document from the data part and then group the related data values according to their path and data types into homogeneous sets. Then, introducing an efficient stream-querying algorithm to traverse this synopsis to obtain the measures or the estimates needed efficiently.

  The XMill approach was used in XCLUSTER [Polyzotis 2006]. Unfortunately, different questions about its efficiency still have no answers, for example: what is the construction time needed for the summary (structure and content of document)? what is the size/structure of the XML document that can be summarized? In XCLUSTER [Polyzotis 2006], the maximum size of XML document used was 10MiB.

- Interval arithmetic guarantees: a new goal of our research will be obtaining a priori interval of error for the time needed to answer a given XPath query. To obtain this goal, we should have accurate intervals for the values of the cost-parameters of our performance prediction mode, and propagate them by interval arithmetic [Hickey 2001] or some method specific to our mathematical model.

6.2.3 Parallel Processing

Another complementary technique for processing XPath queries on very large data sets is parallel processing. Speeding-up multi-query processing by treating each one in parallel is practically useful and requires sharing/copying of the data set, but it poses no fundamental algorithmic problem: duplication of stream-processing algorithms can support it. Its speed-up factor is also limited to the number of independent queries.

Genuine data-parallel processing of a single XPath query, on the other hand, holds the promise of unlimited speedups proportional to the size of the data set and number of processing units. But this poses real complexity- and algorithm-design problems. The former have been identified [Gottlob 2005] and many research groups have studied practical methods since. A modest set of experiments has been started in our group of [ANDRIESCU 2010] and we will study its improvements and generalization. A deeper study will come from ANR project CODEX (2009-2011) and its results will be applied in our group at LACL (Laboratoire d'Algorithmique, Complexité et Logique).

The goal is to have parallel systems on cloud-computing platforms to store and process in parallel, data sets that have been pre-processed on pre-filtered by more economical and pervasive system-processing methods.

Bibliography

[Aboulnaga 2001] A. Aboulnaga, A. R. Alameldeen and J. F. Naughton. *Estimating the Selectivity of XML Path Expressions for Internet Scale Applications*. In Proceedings of the 27th International Conference on Very Large Data Bases (VLDB), pages 591 – 600, 2001. 18, 19, 20, 30, 32, 44, 46

[Aggarwal 2007] C. C. Aggarwal and P. S. Yu. Data Streams Models and Algorithms. Chapter 9: A Survey of Synopsis Construction in Data Streams. Springer, January 2007. 16, 17

[Alrammal 2009a] M. Alrammal, G. Hains and M. Zergaoui. *Intelligent Ordered XPath for Processing Data Streams*. In the Spring Symposium (SSS'09) of the AAAI: Event Processing Stream, Stanford-USA, pages 6–13, 2009. 81

[Alrammal 2009b] M. Alrammal, G. Hains and M. Zergaoui. *Performance Measurements towards the Optimization of Stream Processing for XML Data*. In Proceedings of the 2009 International Conference on Internet Computing (ICOMP'09), July 2009. 96, 145

[Alrammal 2009c] M. Alrammal, G. Hains and M. Zergaoui. *Realistic Performance Gain Measurements for XML Data Streaming with Meta Data*. Technical report, Université Paris-Est. Laboratoire d'Algorithmique, Complexité et Logique., 2009. *http://lacl.univ-paris12.fr/Rapports/TR/TR-LACL-2009-4.pdf*. 121

[Alrammal 2010] M. Alrammal, G. Hains and M. Zergaoui. *A Portable and Extensible Performance Model for Stream-processing of XPath Queries*. Technical report, Université Paris-Est. Laboratoire d'Algorithmique, Complexité et Logique., 2010. *http://lacl.univ-paris12.fr/Rapports/TR/TR-LACL-2010-4.pdf*. 118

[Altinel 2002] M. Altinel and M. J. Franklin. *Efficient Filtering of XML Documents for Selective Dissemination of Information*. In Proceedings of 26th International Conference on Very Large Data Bases (VLDB), pages 53–64, 2002. 33, 35, 36

[ANDRIESCU 2010] E-M. ANDRIESCU, A. AZZABI and G. Hains. *Parallel processing of Forward XPath queries: an experiment with BSML.* Technical report TR-LACL-2010-11, Université Paris-Est. Laboratoire d'Algorithmique, Complexité et Logique., 2010. *http://lacl.fr/Rapports/TR/TR-LACL-2010-11.pdf*. 149

[Bar-Yossef 2004] Z. Bar-Yossef, M. Fontoura and V. Josifovski. *On the Memory Requirements XPath Evaluation over XML Streams.* In Proceedings of the 23rd ACM SIGMOD Symposium on Principles of Database Systems, pages 177–188, June 2004. 5, 36

[Barton 2003] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura and V. Josifovski. *Streaming XPath Processing with Forward and Backward Axes*. In Proceedings of the International Conference on Data Engineering (ICDE), pages 455–466, 2003. 39

[Berglund 2010] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie and J. Siméon. *XML Path Language (XPath) 2.0*. 14 December 2010. *http://www.w3.org/TR/2010/REC-xpath20-20101214/*. 2, 4, 9, 17, 94, 113

[Boag 2010] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie and J. Siméon. *XQuery 1.0: An XML Query Language (Second Edition)*. 14 December 2010. *http://www.w3.org/TR/2010/REC-xquery-20101214/*. 2, 4, 17, 94

[Bohannon 2002] P. Bohannon, J. Freire, P. Roy and J. Siméon. *From XML Schema to Relations: A Cost-based Approach to XML Storage*. In Proceedings of the 18th International Conference on Data Engineering (ICDE), pages 64–75, 2002. 26

[Bonifati 2007] A. Bonifati and A. Cuzzocrea. *Synopsis Data Structures for XML Databases: Models, Issues, and Research Perspectives*. 18th International Workshop on Database and Expert Systems Applications., pages 20–24, 2007. 143

[Böttcher 2007] S. Böttcher and R. Steinmetz. *Evaluating XPath Queries on XML Data Streams*. In Proceedings of the 24th British Inational Conference on Databases (BNCOD), pages 101–113, 2007. 34

[Bray 2008] T. Bray, J. Paoli, C. M. Sperberg-McQueen and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. 26 November 2008. *http://www.w3.org/TR/REC-xml/*. 1, 2, 8, 17, 94, 95

[Brownell 2002] D. Brownell. SAX2. O'Reilly Media, January 2002. 3, 46

[Bruno 2002] N. Bruno, N. Koudas and D. Srivastava. *Holistic Twig Joins: Optimal XML Pattern Matching*. In Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, pages 310–321, 2002. 26, 38, 39

[Bry 2005] F. Bry, F. Coskun, S. Durmaz, T. Furche, D. Olteanu and M. Spannagel. *The XML Stream Query Processor SPEX*. Proceedings of the International Conference on Data Engineering (ICDE), pages 1120 –1121, 2005. 40

[Chan 2002] C. Chan, P. Felber, M. Garofalakis and R. Rastogi. *Efficient Filtering of XML Documents with XPath Expressions*. In Proceedings of the 18th International Conference on Data Engineering, pages 235 – 244, 2002. 32, 35

[Chaudhuri 2004] S. Chaudhuri, V. Ganti and L. Gravano. *Selectivity Estimation for String Predicates: Overcoming the Underestimation Problem*. In Proceedings of the 20th International Conference on Data Engineering (ICDE), page 227, 2004. 22

[Chen 2001] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. T. Ng and D. Srivastava. *Counting Twig Matches in a Tree*. In Proceedings of the 17th International Conference on Data Engineering (ICDE), pages 595 – 604, 2001. 20, 31, 32

[Chen 2004] Y. Chen, S. B. Davidson, G. A. Mihaila and S. Padmanabhan. *Expedite: A System for Encoded XML Processing*. In Proceedings of the thirteenth ACM international conference on Information and knowledge management (CIKM), pages 108–117, 2004. 42

[Chen 2005] T. Chen, J. Lu and T. W. Ling. *On Boosting Holism in XML Twig Pattern Matching using Structural Indexing Techniques*. In Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, 2005. 38

[Chen 2006] Y. Chen, S. B. Davidson and Y. Zheng. *An Efficient XPath Query Processor for XML Streams*. In Proceedings of the 22nd International Conference on Data Engineering (ICDE), 2006. 39, 42, 44, 115

[Choi 2002] B. Choi. *What are real DTDs like?* In Proceedings of the 5th International Workshop on the Web and Databases (WebDB), pages 43–48, 2002. 5, 43

[Clark 1999] J. Clark and S. DeRose. *XML Path Language (XPath),* http://www.w3.org/TR/xpath . November 1999. 4

[DeHaan 2003] D. DeHaan, D. Toman, M. P. Consens and M. T. Ozsu. *A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding*. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pages 623–634, 2003. 42

[Diao 2002] Y. Diao, M. Altinel, M. Franklin, H. Zhang and P. Fischer. *Efficient and Scalable Filtering of XML Documents*. In Proceedings of the 18th International Conference on Data Engineering, pages 341 – 342, 2002. 34, 35, 36, 38

[Diaz 1999] A. L. Diaz and D. Lovell. *Book :IBM's XML Generator*, 1999. *http://www.alphaworks.ibm.com/tech/xmlgenerator*. 9

[Fernández 2010] M. Fernández, J. Siméon, C. Chen, B. Choi, V. Gapeyev, A. Marian, P. Michiels, N. Onose, D. Petkanics, C. Rath, C. Ré, M. Stark, G. Sur, A. Vyas and P. Wadler. *Galax:An Implementation of XQuery*. 2010. *http://www.galaxquery.org/* . 43

[FH 2009] FH. *FH SARL: Barometer of Fixed and Mobile Connections*. 2009. *http://www.freenews.fr/spip.php?article8044*. 140

[Fisher 2007] D. K. Fisher and S. Maneth. *Structural Selectivity Estimation for XML Documents*. In Proceedings of the 23rd International Conference on Data Engineering (ICDE), pages 626–635, 2007. 23, 30

[Florescu 2003] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan and G. Agrawal. *The BEA/XQRL Streaming XQuery Processor*. In Proceedings of the 2003 International Conference on Very Large Data Bases (VLDB), pages 997–1008, 2003. 41, 43

[Freire 2002] J. Freire, J.R. Haritsa, M. Ramanath, P. Roy and J. Siméon. *StatiX: Making XML Count*. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 181–191, 2002. 26, 27

[Goldman 1997] R. Goldman and J. Widom. *DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases*. In Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB), pages 436–445, August 1997. 31, 62, 141, 147

[Gottlob 2002] G. Gottlob, C. Koch and R. Pichler. *Efficient Algorithms for Processing XPath Queries*. In Proceedings of the 2002 International Conference on Very Large Data Bases (VLDB), pages 95–106, 2002. 43

[Gottlob 2005] G. Gottlob, Ch. Koch, R. Pichler and L. Segoufin. *The Parallel Complexity of XML Typing and XPath Query Evaluation*. Journal of the ACM, vol. 52, no. 2, pages 284–335, 2005. 2, 94, 95, 149

[Gou 2007] G. Gou and R. Chirkova. *Efficient Algorithms for Evaluating XPath over Streams*. In Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pages 269–280, 2007. 11, 12, 39, 42, 44, 62, 65, 72, 114, 115, 146

[Green 2003] T. J. Green, G. Miklau, M. Onizuka and D. Suciu. *Processing XML Streams with Deterministic Automata*. In Proceedings of the 9th International Conference on Database Theory (ICDT), pages 173–189, 2003. 44

[Grust 2002] T. Grust. *Accelerating XPath Location Steps*. In Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, pages 109 –120, 2002. 42

[Grust 2004] T. Grust, S. Sakr and J. Teubner. *XQuery on SQL Hosts*. In Proceedings of the 29th International Conference on Very Large Data Bases (VLDB), pages 252–263, 2004. 25

[Grust 2005] T. Grust. *Purely Relational FLWORs*. In Proceedings of the 2nd International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P), in cooperation with ACM SIGMOD, 2005. 25

[Gupta 2003] A. Gupta and D. Suciu. *Stream Processing of XPath Queries with Predicates*. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Ddata, pages 219 – 430, 2003. 36, 39

[Han 2008] W-S. Han, H. Jiang, H. Ho, and Q. Li. *StreamTX: Extracting Tuples from Streaming XML Data*. In Proceedings of the VLDB Endowment, pages 289–300, 2008. 41

[Harary 1960] F. Harary and R.Z. Norman. *Some Properties of Line Graphs*. Wikipedia, 1960. *http://en.wikipedia.org/wiki/Line$_g$raph.* 52

[He 2004] Z. He, B. Lee and R. Snapp. *Self-tuning UDF Cost Modeling using the Memory-Limited Quadtree*. In Proceedings of the 9th International Conference on Extending Database Technology (EDBT), 2004. 20

[Hickey 2001] T. Hickey, Q. Ju and M. H. V. Emden. *Interval Arithmetic: From Principles to Implementation*. In Journal of the ACM, vol. 48, pages 1038 – 1068, 2001. 148

[Hickson 2011] I. Hickson. *HTML5 (W3C Editor's Draft)*. January 2011. http://dev.w3.org/html5/spec/Overview.html. 2

[Hopcroft 1971] J. E. Hopcroft. *An n log n Algorithm for Minimizing the States in a Finite Automaton*. In The Theory of Machines and Computations - Academic Press, pages 189 –196, 1971. 54

[Hopcroft 1979] J. Hopcroft and J. Ullman. Introduction to Automata Theory, Language, and Computation. 1979. 36, 48, 53, 54, 58

[ICE 2007] ICE. *Prefixes for Binary Multiples*. 2007. *http://en.wikipedia.org/wiki/Mebibyte*. 9

[Jiang 2003] H. Jiang, W. Wang, H. Lu and J. X. Yu. *Holistic Twig Joins on Indexed XML Documents*. In Proceedings of the 29th International Conference on Very Large Data Bases (VLDB), vol. 29, pages 273–284, 2003. 38, 41

[Josifovski 2005] V. Josifovski, M. Fontoura and A. Barta. *Querying XML Streams*. VLDB Journal, vol. 14, pages 197 – 210, 2005. 20, 40, 41, 43

[Kay 2007] M. Kay. *XSL Transformations XSLT Version 2.0*. 23 January 2007. *http://www.w3.org/TR/xslt20/*. 4

[Kay 2010] M. Kay. *Saxon: the XSLT and XQuery Processors*. October 2010. *http://saxon.sourceforge.net/* . 44

[Koch 2004] C. Koch, S. Scherzinger, N. Schweikardt and B. Stegmaier. *Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams*. In Proceedings of the 2004 International Conference on Very Large Data Bases (VLDB), pages 228–239, 2004. 41

[Lee 2004] B. Lee, L. Chen, J. Buzas and V. Kannoth. *Regression-based Self-tuning Modeling of Smooth User-defined Function Costs for an Object-Relational Database Management System Query Optimizer*. The Computer Journal, pages 673–693, 2004. 20

[Leroy 2010a] X. Leroy, D. Doligez, J. Garrigue, D. Rémy and J. Vouillon. *Module Gc*. Institute National de Recherche en Informatique et en Automatique (INRIA), June 2010. *http://caml.inria.fr/pub/docs/manual-ocaml/libref/Gc.html*. 104

[Leroy 2010b] X. Leroy, D. Doligez, J. Garrigue, D. Rémy and J. Vouillon. *Objective Caml Language - release 3.12*. Institute National de Recherche en Informatique et en Automatique (INRIA), June 2010. *http://caml.inria.fr/pub/distrib/ocaml-3.12/*. 65, 97, 113, 115, 134

[Ley 2011] M. Ley. *DBLP bibliography*. January 2011. http://dblp.uni-trier.de/xml/. 25

[Li 2006] H. Li, M. L. Lee, W. Hsu and G. Cong. *An Estimation System for XPath Expressions*. In Proceedings of the 22nd International Conference on Data Engineering (ICDE), page 54, 2006. 29, 30

[Liefke 2000] H. Liefke and D. Suciu. *XMILL: An Efficient Compressor for XML Data*. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 153–164, 2000. 16, 22, 148

[Lim 2002] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter and R. Parr. *XPath-Learner: An On-line Self-tuning Markov Histogram for XML Path Selectivity Estimation*. In Proceedings of 28th International Conference on Very Large Data Bases (VLDB), pages 442 – 453, 2002. 19

[Ludascher 2002] B. Ludascher, P. Mukhopadhyay and Y. Papakonstantinou. *A Transducer-based XML Query Processor*. In Proceedings of the 2002 International Conference on Very Large Data Bases (VLDB), pages 227–238, 2002. 41, 43, 44

[Luo 2009] C. Luo, Z. Jiang, W-C Hou, F. Yu and Q. Zhu. *A Sampling Approach for XML Query Selectivity Estimation*. In Proceedings of the International Conference on Extending Database Technology (EDBT), pages 335–344, 2009. 21, 25, 26, 31, 140, 147

[Nizar 2008] A. Nizar and S. Kumar. *Efficient Evaluation of Forward XPath Axes over XML Streams*. In Proceedings of the 14th International Conference on Management of Data (COMAD), pages 222–233, 2008. 42, 44

[Nizar 2009a] A. Nizar and S. Kumar. *Ordered Backward XPath Axis Processing against XML Streams.* Proceedings of the 6th International XML Database Symposium (XSym), pages 1–16, 2009. 43, 147, 148

[Nizar 2009b] M. A. Nizar, G. S. Babu and P. S. Kumar. *SFilter: A Simple and Scalable Filter for XML Streams*. In Proceedings of the 15th International Conference on Management of Data (COMAD), 2009. 37

[Olteanu 2002] D. Olteanu, H. Meuss, T. Furche and F. Bry. *XPath: Looking Forward*. In Proceedings of the 2002 XML-Based Data Management and Multimedia Engineering (EDBT) Workshops, pages 109–127, 2002. 4, 40, 148

[Olteanu 2007] D. Olteanu. *SPEX: Streamed and Progressive Evaluation of XPath*. IEEE Transactions on Knowledge and Data Engineering, pages 934–949, 2007. 39, 40

[Parberry 1987] I. Parberry. Parallel Complexity Theory. 1987. 95

[Peng 2003] F. Peng and S. S. Chawathe. *XPath Queries on Streaming Data*. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pages 431–442, 2003. 32, 39, 42, 44, 115

[Polyzotis 2002a] N. Polyzotis and M. Garofalakis. *Statistical Synopses for Graph-structured XML Databases*. In Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, pages 358–369, 2002. 20

[Polyzotis 2002b] N. Polyzotis and M. Garofalakis. *Structure and Value Synopses for XML Data Graphs*. In Proceedings of the 28th international conference on Very Large Data Bases (VLDB), pages 466–477, 2002. 21, 22

[Polyzotis 2004a] N. Polyzotis, M. N. Garofalakis and Y. Ioannidis. *Approximate XML Query Answers*. In Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pages 263–274, 2004. 20, 30, 31, 62

[Polyzotis 2004b] N. Polyzotis, M. N. Garofalakis and Y. Ioannidis. *Selectivity Estimation for XML Twigs*. In Proceedings of the International Conference on Data Engineering (ICDE), 2004. 20, 31

[Polyzotis 2006] N. Polyzotis and M. N. Garofalakis. *XCluster Synopses for Structured XML Content*. In Proceedings of the International Conference on Data Engineering (ICDE), 2006. 22, 31, 148

[Poosala 1996] V. Poosala, Y. E. Ioannidis, P. J. Haas and E. J. Shekita. *Improved Histograms for Selectivity Estimation of Range Predicates*. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 294–305, 1996. 22

[Ramanan 2005] P. Ramanan. *Evaluating an XPath Query on a Streaming XML Document*. International Conference on Management of Data (COMAD), 2005. 41

[Saker 2007] S. Saker. *Cardinality-aware and Purely Relational Implementation of an XQuery Processor*. PhD thesis-University of Konstanz, pages 58–82, 2007. 24, 31

[Saker 2008] S. Saker. *Algebra-based XQuery Cardinality Estimation*. International Journal of Web Information Systems, pages 7–46, 2008. 24, 31

[Sakr 2010] S. Sakr. *Towards a Comprehensive Assessment for Selectivity Estimation Approaches of XML Queries*. In Proceedings of the International Journal of Web Engineering and Technology, vol. 6, pages 58–82, 2010. 23, 24, 30, 31

[Schmidt 2001] A. Schmidt, R. Busse, M. Carey, M. Kersten D. Florescu, I. Manolescu and F. Waas. *XMark: An XML Benchmark Project.* Technical report, 2001. *http://www.xml-benchmark.org/*. 9, 20, 26, 92, 138, 146

[Suciu 1992] D. Suciu. *TreeBank: XML Data Repository.* Rapport technique, University of Pennsylvania Treebank Project, Novmber 1992. *http://www.cs.washington.edu/research/xmldatasets*. 9, 92, 126, 138, 146

[Takekawa 2007] H. Takekawa and H. Ishikawa. *Incrementally-Updatable Stream Processors for XPath Queries based on Merging Automata via Ordered Hash-keys*. In Proceedings of the 18th International Conference on Database and Expert Systems Applications (DEXA), pages 40–44, 2007. 37

[Teevan 2005] J. Teevan, E. Adar, R. Jones and M. Potts. *History Repeats Itself: Repeat Queries in Yahoo's Query Logs*. In Proceedings of the 29th Annual ACM Conference on Research and Development in Information Retrieval (SIGIR), pages 703–704, 2005. 96

[TenCate 2009] B. TenCate and M. Marx. *Axiomatizing the Logical Core of XPath 2.0*. Theoretical Computer Science, vol. 44, pages 561–589, 2009. 2, 94, 95

[Teubner 2008] J. Teubner, T. Grust, S. Maneth and S. Sakr. *Dependable Cardinality Forecasts for XQuery*. In Proceedings of the VLDB Endowment (PVLDB), pages 463–477, 2008. 25

[Urpalainen 2008] J. Urpalainen. *XML Patch Operations Framework Utilizing XPath Selectors*. Network Working Group, 2008. *http://datatracker.ietf.org/doc/rfc5261/*. 61, 142

[Wang 2003] W. Wang, H. Jiang, H. Lu and J. X.Yu. *Containment Join Size Estimation: Models and Methods*. In Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, pages 145–156, 2003. 28

[Wang 2004] W. Wang, H. Jiang, H. Lu and J. X. Yu. *Bloom Histogram: Path Selectivity Estimation for XML Data with Updates*. In Proceedings of the 13th International Conference on Very Large Data Bases (VLDB), pages 240 – 251, 2004. 28, 30

[Wu 2002] Y. Wu, J.M. Patel and H.V. Jagadish. *Estimating Answer Sizes for XML Queries*. In Proceedings of the 8th International Conference on Extending Database Technology (EDBT), pages 590–608, 2002. 27, 28

[Zhang 2005] N. Zhang, P. Haas, V. Josifovski, G. Lohman and C. Zhang. *Statistical Learning Techniques for Costing XML Queries*. In Proceedings of the 31st VLDB Conference on Very Large Data Bases (VLDB), pages 289–300, 2005. 19, 32, 44, 46, 132

[Zhang 2006a] G. Zhang and Q. Zou. *QuickXScan: Efficient Streaming XPath Evaluation*. In Proceedings of the International Conference on Internet Computing, pages 249–255, 2006. 41

[Zhang 2006b] N. Zhang, M. T. Ozsu, A. Aboulnaga and I. F. Ilyas. *XSeed: Accurate and Fast Cardinality Estimation for XPath Queries*. In Proceedings of the 20th International Conference on Data Engineering, 2006. 21, 26, 30, 31, 62, 140, 147