Intelligent Ordered XPath for Processing Data Streams

Muath Alrammal, Gaétan Hains, Mohamed Zergaoui

Intelligent Ordered XPath for Processing Data Streams

Muath ALRAMMAL **Gaétan HAINS** **Mohamed ZERGAOUI**

July 2008

TR–LACL–2008–4

Laboratoire d'Algorithmique, Complexité et Logique (LACL)
Département d'Informatique
Université Paris 12 – Val de Marne, Faculté des Science et Technologie
61, Avenue du Général de Gaulle, 94010 Créteil cedex, France
Tel.: (33)(1) 45 17 16 47, Fax: (33)(1) 45 17 66 01

Laboratory of Algorithmics, Complexity and Logic (LACL)
University Paris 12 (Paris Est)

Technical Report **TR–LACL–2008–4**

M. ALRAMMAL, G. HAINS, M. ZERGAOUI

Intelligent Ordered XPath for Processing Data Streams

Abstract

Data Streaming is a necessary and useful technique to process very large XML documents but poses serious algorithmic problems, various approaches and algorithms were implemented to fulfill this purpose. In this article we highlight and categorize the different important works in this domain. Furthermore, we present the critical parameters which affect the complexity of the streaming algorithms accompanied with examples. In the end, we propose a new approach for processing XML data streams using intelligent Ordered XPath that account for these critical parameters .

1 Introduction

The extensible markup language (XML) [30], 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. It provides a much needed non-proprietary universal format for sharing hierarchical data among different software systems, and does so in a user-friendly manner. It is a verbose plain-text format, making it robust, platform-independent and legible to humans without additional tools. This popularity of XML has created several important applications like information dissemination, processing of the Scientific data, and real time news. Query Languages like XPath [14] and XQuery [4] have been proposed for accessing the XML data, which provide a syntax for specifying which elements and attributes are sought to retrieve specific pieces of a document.
Often, data is very large to fit into limited internal memory and in many cases it needs 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, in this case streaming fashion is the best approach to process the XML Data. The term data streaming is used to describe data items that are available for reading only once and that are provided in a fixed order determined by the data source. In the streaming model queries must be known before any data arrive, so queries can be preprocessed by building machines for query evaluation . Figure 1 illustrates a data stream processor.

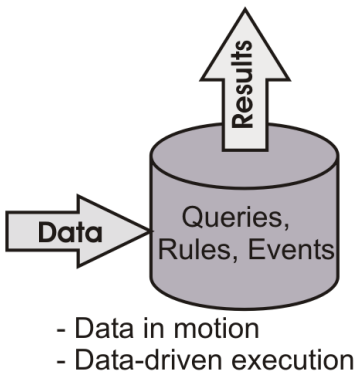

Figure 1: Data Stream Processor.

Due to the recursive nature of XML data, the single sequential forward scan of an XML streams, also the presence of descendant axes and the predicates in an XPath query, query evaluation process on XML streams raises different challenges compared to non-streaming environments, some of them are :

- During evaluation of XPath queries that contain predicates in streaming, we may encounter data that can be query solutions, before we reach the required data to evaluate the predicates[1] in order to decide its 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 that we can determine the predicate satisfaction and deciding their membership.

[1]XML data may be processed in several different models, which require different query processing strategies like LAZY and EAGER

- The descendant axis traversal in a query and recursive structure of the XML document may cause an exponential number of pattern matches of sub-query from a single initial node.
- The property of the existing XPath expression whose are unordered (predicates and axes) will increase the number of buffered potential answer nodes before we encounter the required data to evaluate the predicate to decide its membership. **In the worst case the buffering size will reach the document size**.

1.1 Preliminaries

- **Data Model of XML Streams**
  An XML document can be seen 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. The streaming XML data is modeled as a sequence of SAX [5] events extended with the depth of the event. An XML streaming algorithm accepts input XML document as SAX events, these events are: startElement(X) and endElement(X) which are activated respectively when the opening or closing tag of a streaming element is encountered and accept the name of the element X as input parameter.

- **XPath**
  XPath [14] is a language that describes how to locate specific elements (and attributes, processing instructions, etc.) in a document, it treats an XML document as a logical ordered tree. XPath has a particular importance for XML applications since it is a core component of many XML processing standards such as XSLT [13] or XQuery [4].

  textitForward XPath [2] is a fragment of XPath consists of queries that have only child, attribute, or descendant axes, it has different subfragments based on the manner of computing the predicates. Symmetric predicate is a predicate P if its evaluation on any document node X is independent of the order of $X's$ children, also it is a predicate that does not contain positional function operators such as *position*() or *last*(). *Symmetric XPath* is a fragment of Forward XPath consisting, if any, only symmetric predicates, for example $F[N]$, $F[N < B]$ are symmetric, while $F/G[last()]$ is not. Atomic Predicate can be called Univariate if it depends on the value of at most one node in the query tree. Based on that *Univariate XPath* is a fragment of Forward XPath consisting, if any, only Univariate predicates, for example, $F[S > 9]$ is a Univariate Atomic XPath, while $F[S > N]$ is not. Furthermore, *SubSumption-free XPath* are queries that do not contain redundancies. Some queries can be rewritten to be SubSumption-free by eliminating redundant portions. They can be considered as subset of queries of Univariate XPath.
  In this article we focus on Forward XPath. Figure 1.1 illustrates its grammar. A location path is a structural pattern composed of sub expressions called step. 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 './/'.

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

Figure 1.1 Grammar of the Intelligent Forward XPath

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: Nodes of Structural Matching of Q in D

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 paper [24].

Concerning query evaluation process, usually two principle questions are asked. First, does our document D match the query Q?, in this case we determine whether there exists at least one match of Q in D. Second, which part or parts of the D match the Q?, it means outputting all nodes in an XML data tree D (answer nodes) whose satisfy a specified twig pattern Q at its result node. Different strategies and streaming algorithms were implemented for this purpose, we can categorize some of them based on the time of predicates evaluation in queries. **Lazy** streaming algorithm evaluates the predicates only when the closing tags of the streaming elements are encountered. While the **Eager** algorithm does that as soon an atom in a predicate is evaluated to true.

- **Recursion In XML Data**
  Recursion occurs frequently in XML data in practice [11], where some elements with the same name are nested on the same path in the data tree. In [2], they define it as the recursion depth of an XML data tree D w.r.t the query node q in Q, denoted by r_q, as the length of the longest sequence of nodes $e_1, ..., e_r$ 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 2 illustrates the depth of document D w.r.t the query Q //A//B[.//C]/K

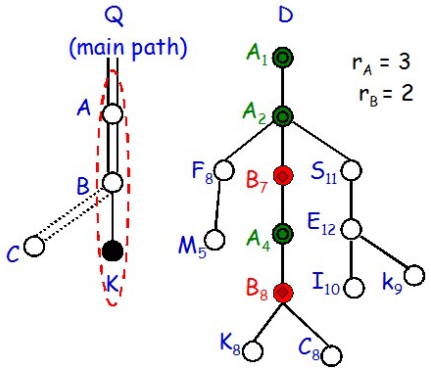

Figure 2: Recursion depth of D w.r.t Q.

where the single line edges represents child ('/'), the double line edges represents descendant('//'), single dashed line represents [./node()], double dashed line represents [.//node()] and the result node which is in this example the shaded node K. It is obvious from figure 2 that node C is not on the main path, this is why we do not consider it as a r_q. If we have a look at table 1 clearly, both nodes A and B satisfy the definition of r_q. Actually, $r_A = 3$ represented by (A_1, A_2, A_4), while $r_B = 2$ represented by (B_7, B_8) In our example, figure 2 the used XPath expression can be considered a **simple one**, for more complex XPath expression, see figure 3, it benefits from the grammar of figure 1.1. Moreover our Query is unordered (axes and predicate position), because node C in Q can be **before** node K or **after**.

- **Document Depth**
  The depth of a document is the length of the longest root to leaf path in the tree representing the document d_D. In our example in figure 2, document depth is the path from root node A_1 to the leaf node K_8

2 Related Work

Large amount of work has been conducted to process XML documents in streaming fashion. The different approaches to evaluate XPath queries on XML data streams can be categorized by the subset of XPath they support. In fact, most of them are *automata based* (XPush [19], XSQ [25], SPEX [23], XMLTK [1]) or *parse trees* ([10],[3],[2],[17]). All of them support a particular fragment of XPath called Forward XPath as it is defined in section 1.1 , other works use a sub-fragment of Forward XPath. YFilter [19], XMLTK [1] and X-Scan [20] support XPath queries containing the child and descendant-or-self axes and wildcards using finite state automata.

In [19] authors seek to process efficiently large number of XPath expressions with many predicates per query on a stream XML data. Therefore, they propose to lazily construct a single deterministic pushdown automata (a special deterministic stack machine) which is called "XPush Machine" for a given XPath filters, the input is a series of SAX events and the output is a set of filters that match the processed document. The used fragment of XPath is Forward XPath. Note that, the memory requirement for this technique is manageable and the cost of laziness is recovered later. An application example for this work is XML Message Brokers.
AFilter[7] is an adaptable XPath query evaluation approach that needs a base memory requirement which is linear in query and data size. If more memory is provided to AFilter, it uses the remaining main memory for a caching approach to evaluate queries faster than with only the base memory. AFilter is mainly based on a lazy DFA and it supports wildcard, but does not support predicate filters. Similar to XPush [19], it is designed to evaluate a large set of queries.

The approach presented in TurboXPath[22] discusses how to handle the child and descendant or- self axes, predicates (including functions and arithmetics) and wildcards in XQuery using TurboXPath. The input query is translated into a set of parse trees. Whenever a matching of a parse tree is found within the data stream, 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 the conditions of which have been evaluated to true. The used XPath fragment is a Forward XPath (child, descendant, self, axes, any node test (*), functions, arithmetic and structural predicates). Furthermore, it supports parent and ancestor axes. Important feature of TurboXpath, it avoids the translation of query into a finite state automata like [23] which might degrade the query performance in scenarios where several independent queries are executed in parallel over different XML Stream. Another distinguishing feature its capability to process query trees constructed of several concatenated path returning tuples as result.

[2] builds a parse tree as well, it supports the parent and the ancestor axis in addition to that Forward XPath axis. This parse tree is used to 'predict' the next matching nodes and the level in which they have to occur, for example, consider the query //C/D and a matching of C in level 3. Then the next interesting matching would be a node D in level 4.

XSQ [25] and SPEX [23] propose a method for evaluating XPath queries over streaming data to handle closers, aggregation and multiple predicates. Their method is designed based on hierarchical arrangement of push-down transducers augmented with buffers. Automata is extended by actions attached to states, extended by a buffer to evaluate XPath queries. The main idea is that a nondeterministic push-down transducer (PDT) is generated for each location step in an XPath query, and these PDTs are combined into a hierarchical pushdown transducer in the form of a binary tree. In fact XSQ might buffer multiple physical copies of a potential answer nodes, because buffering space size is measured by the maximum of physical copies of a potential answer nodes buffered at the same time during the running time, therefore, it might has a higher cost, particularly with very deep recursion. The Considered fragment of XPath is XPath 1.0 including(closures, aggregations, and multiple predicates) except reverse axes and position functions (such as *position*() and *last*()). This technique eagerly evaluates queries and it is fully implemented thus give us the possibility to compare our algorithms with it.

The approach presented in [10] uses a structure which resembles a parse tree with stacks attached

to each node. These stacks are used to store XML nodes that are solutions to the parse tree nodes subquery (or to store XML nodes that are candidates for a solution in case of predicate filters).

In [9] authors aim to achieve polynomial time complexity in both data and query size for evaluating XPath queries on streaming data, based on that, they propose ViteX System that is composed of four modules: 1. XPath Parser, it takes an XPath query Q as input and generates a tree representation of the query. 2. The TwigM builder, which constructs a TwigM machine according to the input query tree. 3. The SAX parser that takes an XML stream and outputs a sequence of SAX events. 4. As SAX events stream in, TwigM changes its state according to the current state and the input event, then it computes a set of XML fragments as solutions to Q. The idea is using an Encoding Technique, TwigM 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. Notice that it supports queries with AND operator only. The used fragment of XPath is a Sub-fragment of Forward XPath $\{/, //, *, []\}$. TwigM achieves a complexity of Polynomial time $O(|D||Q|(|Q| + d_D.B)$ where B is the size of the candidate solutions. While for non-recursive dataset[2] TwigM has the complexity of $O(|D||Q|)$. It has the same case like XSQ, because sometimes it might buffer multiple physical copies of a potential answer node at the same running time.

[31] introduce a streaming XPath algorithm (QuickXScan), it evaluates XPath expression with predicates by one sequential Scan of XML data, it is based on the principles similar to that of attribute grammars. QuickXScan support the following forward axes: child, descendant, attribute, self and self-or descendant and the parent axes by transformation into forward axis [24]. Authors, model an XPath query with a query tree. In structural join based algorithms, there is a nice solution of using compact stacks to represent a possibly combinatorial explosive number of matching path instantiations with linear complexity like [21], therefore, QuickXScan extends the idea of compact stacks in a technique called matching grid, which is used also in [27]. Again query representation can be considered as the best feature of this technique because it represents complex queries using a query tree, together with a set of variables and evaluation rules associated with each query node. Notice that in their paper they do not address the issue of streaming output.

In [8] they present a model of data processing in an information systems exchange environment, it consists of a simple and general encoding scheme for servers, and streaming query processing algorithms on encoded XML stream for data receivers with constrained computing abilities "binary encoding". The target XPath is a subset of Forward XPath that include: child, descendant, predicate and name tests. The EXPedite query processor takes an encoded XML stream and an encoded XPath query as input, and outputs the encoded fragment in the XML stream that matches the query. The idea of the query processing algorithm is taken from different proposed techniques [15], [18] for efficient query evaluation based on XML node labels for XML data stored in the database , in other words, the used algorithm evaluates queries based on XML labels ¡start, end, depth¿. For the time being, EXPedite encoder support only XML elements and character data, and the query processor support Twig queries

In [6] a SAX Based approach is introduced to evaluate the XPath queries that support all axes of *Core XPath*. Starting from the XPath query, this approach generates a stack of automata that uses the SAX stream as input and generates the result of the query as an output SAX stream. Each input query is translated into an automaton that consists of only four different types of transitions. The small size of the generated automata allows for a fast evaluation of the input XML data stream within a small amount of memory. Note that their system supports the sibling axes, whereas other approaches like (XMLTK [1], AFilter [7], XPush [19], XSQ [25], SPEX [23]) are limited to the parent-child and ancestor-descendant axes. They have implemented a prototype called XPA. The query processor decomposes and normalizes each XPath query, such

[2]There are no nodes of a certain type can be nested in another nodes of the same type.

	XMark	Book	TreeBank
Structure	Wide and Shallow	Semi deep+ and recursive	narrow deep and recursive
Data Size	113M	12M	82M
Nbr. of nodes	1666K	114K	2437K
Max./Avg Depth	12/5.5	22/19.4	36/7.6

Table 2: Different Dataset Structures

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 [17] authors propose two Algorithms to evaluate XPath over streams, algorithms accept XML Document as a stream of SAX events. Start and end element events are activated when the opening and closing tag of the streaming elements are arrived. Streaming algorithms are categorized into two classes, based on when they evaluate the predicate in the queries: 1. Lazy Streaming Algorithm (LQ), evaluation is occurred at the closing tag of the streaming element. 2. Eager Streaming Algorithm(EQ), evaluation is occurred if an atom in the predicate is evaluated to true. The used fragment of XPath called *Univariate XPath*, see section (1.1). 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. Based on their experiments, both LQ and EQ algorithms show very similar time performance in practice. In non recursive cases, LQ and TwigM has the same buffering space costs, as well as, EQ and XSQ has the same case. EQ achieve the optimal buffering space performance, therefore it can be considered as the best performance representation between algorithms in the state of the art of their paper.
In comparison to all these approaches, we introduce the notion of an ordered and oriented Forward XPath taking in our consideration that attribute axis it can not be handled in a way similar to the child axis.

3 Motivations and Objectives

3.1 Complexity

Usually, the caching space costs of stream-querying algorithms depend on the number of elements cached in the run-time stack(s). It is bounded by the maximum document depth d_D when queries do not involve *-nodes or the same name nodes and does not exceed $(|Q|.d_D)$ in the worst case. Also time performance of stream querying algorithms can be measured by the following $T_p = t_{all} - t_{input} - t_{output}$, where t_{all} is the total running time, t_{input} is the reading time usually from the disk into memory also parsing the XML document, and t_{output} is the taken time to output the result nodes from the memory to the disk. In practice complexity depends on the following :

- **Structure of XML Document data**
  Documents may have different structures, for instance, shallow XML documents (Wide) that does not include recursive elements. In this case the cashing space costs of the stream-querying algorithm is almost negligible. An example for this type of XML document is **XMark** [28] which is a famous benchmark dataset that allows users and developers to gain insights into the characteristics of their XML repositories. See table 2 which indicates the maximum depth of this dataset that reaches 12(not deep), and it has a large data size 113M.

  While others are semi deep and recursive like **Book** dataset [16], actually it is a synthetic dataset, generated using IBM's XML generator, based on real DTD from W3C XQuery use case. As we can see from table 2, it has a size of 12 M which is not enormous and a maximum depth that reaches 22 which is quite deep enough comparison 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 tree, therefore this kind of dataset(semi deep and recursive) increase the buffering space and processing time. We can find also document with narrow deep structure, an example **TreeBank** [29], here one can recognize the structure of the document from the Maximum depth in table 2, it is 36, moreover, the average depth which is 7.8 indicates that this document is narrow. The existence of these properties in the document will increase of costs of the stream-querying algorithms significantly .

- **The XPath Expression (recursion)**.
  Simple XPath expressions may not require huge buffering space size, see figure 2, while the existence of descendant axis '//', the wildcard * or the same name node in the expression will enormously increase the buffering space size and processing time particularly with deep and recursion XML document because we will be forced to buffer large number of potential answer nodes, an example of this XPath Expression type is //A[./B//*]//*[./A]/K see figure 3. We call this extension **Complex XPath**.

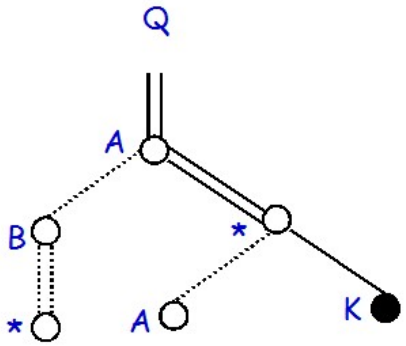

Figure 3: Complex XPath

- **Query Evaluation Strategy**.
  The used methodology to evaluate the XPath expression may affect the buffering space size and processing time. For example, let us consider that we have the document D and the query Q //A[./F]/C figure 4. In 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. While in 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.

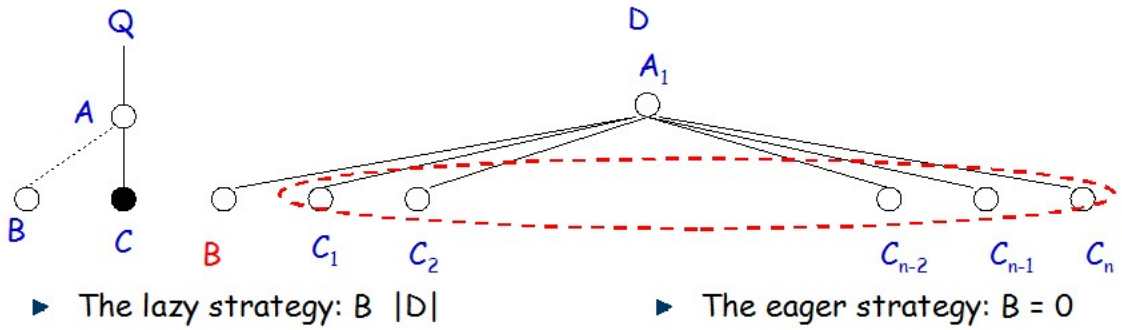

Figure 4: Lazy and Eager Approaches.

Note that B is the size of buffering space, which is measured by the maximum number of potential answer nodes buffered at a time during the running time. It is clear that Eager

strategy has a better buffering space strategy, but both of them do not handle the problem of descendant axis and the complexity of *unordered* XPath expressions. Note that figure 4 is an example of a wide dataset.

3.2 Motivation Examples

- **Descendant axis Problem**
  In parse tree approach, an XPath query specifies a twig pattern Q to navigate an XML Document D. Document size $|D|$ is the number of elements in the D, while query size is the number of nodes in Q. Query nodes are three types, we have the **result node** which specifies the output of the XPath, also **axis nodes** which are all non-result nodes on the main path of Q, that means on the path from the root to the result node. Finally, **predicate nodes** which are all other nodes. For example let us suppose that we have the following XPath //A[./E]/B[./F]//C[./G/I and //H]/D[./J], which is illustrated in figure 5.

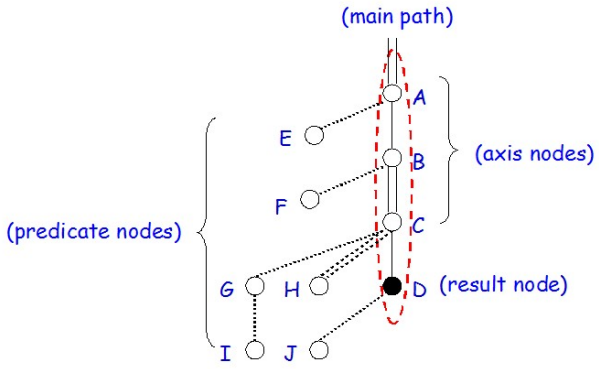

Figure 5: Twig Pattern of an XPath Query Q.

An important question is the order of the node H, actually in our example node H can be in different orders, might be before nodes G, I or after, and in a worst case it can be after node D ,which will increase the complexity (buffer space and processing time) remarkably, due to the fact that our XPath expression is unordered (axes and predicates position).

- **Attributes Order Problem**
  Another important point is the child and attribute axes. Both axes can not be handled in a similar way. The reason again relates to elements order. A simple example that prove the difference between both axes is the following: let us consider that we have the following XML Documents (f,g), figure 3.2 .

```
<A m='male' f='female'>          <A f='female' m='male'>
  <B> </B>                          <C></C>
  <C> </C>                          <B></B>
</A>                             </A>

   XML Document f                    XML Document g
```

Figure 3.2 XML Documents

Obviously, in both documents attributes are unordered, just because attribute specification markup is unordered, which means, the order of attribute specifications in a start-tag or empty-element tag is not significant [30]. While it is not the same matter for elements. The order of elements in an XML document as determined by XML syntax is called **document order**. One can have other types of orders such as alphabetical and numerical, but document order is determined solely by XML syntax. Both [2] and [17] do not consider this point, they do not treat explicitly the attribute axis @, considering that it can be handled in a way Similar to the Child Axis. This consideration affects the evaluation

process of the XPath over the XML data stream.

- **Buffering space size explosion**
  Launching a complex unordered XPath expression that contain *-nodes and the same name nodes to search a narrow deep recursive XML Document affects the complexity by increasing both processing time and buffering space incredibly. In the worst case, it may result in buffering space size explosion. An example for that is launching complex XPath expressions to search the dataset **Treebank** in table 2 using *XSQ* [25] . Moreover, in our case, XSQ may report too many path combinations and terminate the searching process. While using an ordered XPath expression will reduce remarkably the potential answer nodes which is a very good complexity investment for both time processing and buffering space size.

4 Aware-Metadata Schema and Ordered XPath

In this paper, we propose the idea of a new approach for processing XML data streams in order to prepare data to a reasoning task. Our processing relies on the structure of the document, the information that it contains, and the ordered XPath.

Metadata is data associated with objects which relieves their potential users of having full advance knowledge of their existence or characteristics. The term "meta" derives from the Greek word denoting a nature of a *higher order*. Information resources must be made visible in a way that allows people to tell whether the resources are likely to be useful to them. This is no less important in the online world, and in particular, the World Wide Web. Metadata is a systematic method for describing resources and thereby improving access to them. If a resource is worth making available, then it is worth describing it with metadata, so as to maximize the ability to locate it.

Ordered XPath has been investigated before in relational database. [12] Presents a novel approach to efficiency evaluate ordered XPath queries in a relational databases. A scheme extends SUCXENT++ [26] was proposed to support the processing of ordered axes and predicates while maintaining its original properties, their main focus is on the evaluation of the following, preceding, following-sibling, and preceding-sibling axes as well as position-based and range predicates.

In this article we define the **ordered XPath** as the following : ***it is a language that treats an XML document as logical ordered tree to locate specific elements (and attributes, processing instruction, etc... .) considering that it has ordered axes and predicate position.***
To support ordered XPath Queries, the order information of nodes must be captured by the schema to process these queries efficiently. Figure 6 illustrates the order (ordered axes) of document D.

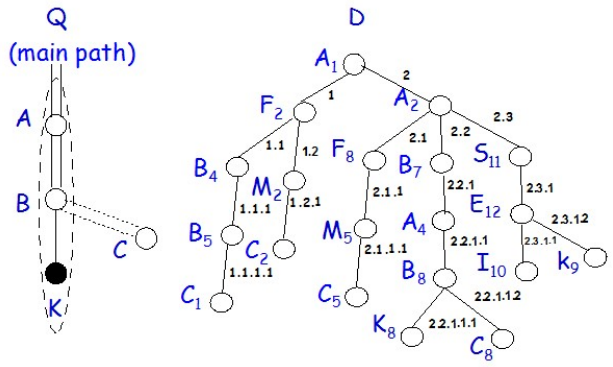

Figure 6: Document Information Order

In our state of the art we found many techniques and algorithms to process and evaluate XPath Expressions over XML Data streams. These techniques vary in the used fragment of XPath, but all of them use ***unordered*** XPath Expression. In section 3.2 we introduced very important motivations that explain the idea and complexity investment of ordered XPath to evaluate efficiently the axes and position predicates. Next we will present few examples to highlight the difference between using an ordered and unordered XPath expressions .

4.1 Ordered vs. Unordered XPath

The purpose of aware-metadata schema is to help us locating the part of XML document we search by providing information about the structure and the content of this document. Thus we can profit from the ordered XPath property to efficiently retrieve the required data that correspond to the XPath expression. ***Comparison examples*** between both ordered and unordered XPaths are as follows:
Example 1: let us suppose that we use the dataset **TreeBank** which is narrow and deep, see table 2 and the Lazy strategy of figure 4. Though Eager strategy has a better performance on this example, we use the Lazy one in our example to illustrate the importance of ordered XPath. By using unordered XPath, the buffering space will be in the worst case $B = |D| = 2437$. Now using ordered XPath which is provided in information by user as follows (start after node order 1000 or interesting potential answer nodes are after node order 1000), this will reduce the worst case of buffering size to be B=1437, such information may be available for a user who can grossly locate the target of the query, for example "the section we search in is somewhere after page number 1000 in the book". In the same example imagine that ordered XPath was provided in information as follows (always search to right of the context node), here buffering size can be reduced remarkably because we will avoid buffering any potential answer node lie to the left of the context node.
Example 2: in figure 6 if the XPath is provided in information by user as follows(start at the axis order 2 of node A_1, then we do not need to buffer the following nodes: B_4, B_5, C_1 though they would otherwise match the query. Note that the position of the predicate node C is to the right(after) of the result node K(predicate position). Thus, we avoid to evaluate the predicate node C_5.

4.2 Advantages of ordered XPath

It is true that sometimes by restricting the query to ordered XPath, our document may not satisfy the query. In this case relaunching the query with new information is necessary. In the same time, there are many advantages of ordered XPath that account for the critical complexity parameters. They are summarized as the following :
(1) *XPath complexity*: in both cases, **simple XPath**, see figure 2 or **complex XPath** see figure 3, the buffering size performance will be improved due to the property of the ordered (axes and predicates position) of XPath, while it is not the same case for unordered XPath.
(2) *XML document structures*: ordered XPath can be efficient with the different XML document structures. For example, the wide dataset like **XMark**, or the narrow recursive like **TreeBank**, as we saw in the examples of section 4.1.
(3) *Query evaluation approach*: it is as well, efficient with the various used approach of query evaluation like **lazy** and **eager**. In the same example of section 4.1, it is explained how to avoid the worst case of buffering size with Lazy. Note that Eager approach may has a better performance. So using ordered XPath with the Eager approach will also result in a better performance than using it with the Lazy one. Space saving come as we have shown, from reduced buffering, and for a given strategy Eager/Lazy ordering will decrease time simply by avoiding unnecessary buffering operations.
We can conclude based on the previous examples, that the efficiency of ordered XPath does not only improve the buffering space, but also the processing time.

5 Conclusion

In this article we presented different approaches related to XPath expression evaluation process over XML data Stream, we categorized the different works by the sub-set of XPath they support into main two parts. First, Automata based, like (XPush [19], XSQ [25], SPEX [23], XMLTK [1]). Second , parse trees like ([10],[3],[2],[17]). Variant examples were introduced to explain the complexity of streaming algorithms, like query evaluation strategy, the XPath expression (recursion) and the structure of XML document data. Based on that the notion of the intelligent ordered XPath for processing efficiently large XML data streams was introduced.

6 Future Work

The process of writing an algorithm for the intelligent ordered XPath is under going, after that, experiments will be done to compare the performance of our algorithm with stream-querying systems like [17], XSQ [25] and [9]

7 Acknowledgments

The first author is funded by a doctoral scholarship from INNOVIMAX and a CIFRE scheme from the French government.

References

[1] I. Avila-Campillo, T. J. Green, A. Gupta, M. Onizuka, D. Raven, and D. Suciu. XMLTK:An XML Toolkit for Scalable XML Stream Processing. *Proceedings of PLANX*, (10), October 2002.

[2] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. On the Memory Requirements XPath Evaluation over XML Streams. *PODS.*, pages 177 – 188, June 2004.

[3] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming XPath Processing with Forward and Backward Axes. *ICDE*, pages 455–466, 2003.

[4] S. Boag, M. F. Fernndez, D. Florescu, J. Robie, and J. Simon. XQuery 1.0: An XML Query Language. January 2007, http://www.w3.org/TR/xquery.

[5] D. Brownell. *SAX2*. January 2002.

[6] S. Bttcher and R. Steinmetz. Evaluating XPath Queries on XML Data Streams . *BNCOD*, pages 101–113, 2007.

[7] K. S. Candan, W. Hsiung, S. Chen, J. Tatemura, and D. Agrawal. AFilter: Adaptable XML Filtering with Prefix-Caching and Suffix-Clustering. *VLDB*, pages 559 – 570, 2006.

[8] Y. Chen, S. B. Davidson, G. A. Mihaila, and S. Padmanabhan. Expedite : A System for Encoded XML Processing . *CIKM*, pages 108–117, 2004.

[9] Y. Chen, S. B. Davidson, and Y. Zheng. ViteX : a Streaming Xpath Processing System. *Proceeding of the 21St international conference on Data Engineering ICDE*, pages 1118 – 1119, 2005.

[10] Y. Chen, S. B. Davidson, and Y. Zheng. An Efficient XPath Query Processor for XML Streams. *Proceedings of the 22nd International Conference on Data Engineering ICDE* , 2006.

[11] B. Choi. What are real DTDs like? *International Workshop on the Web and Databases*, 2002.

[12] B. Choi, E. Leonardi, B.-S. Seah, K. G. Widjanarko, and S. S. Bhowmick. Efficient Support for Ordered XPath Processing in Tree-Unaware Commercial Relational Databases. *Springer*, 2007.

[13] J. Clark. XSL Transformations XSLT. November 1999, http://www.w3.org/TR/xslt.html.

[14] J. Clark and S. DeRose. XML path language (XPath). November 1991, http://www.w3.org/TR/xpath.

[15] D. DeHaan, D. Toman, M. P. Consens, and M. T. Ozsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding . *SIGMOD*, 2003.

[16] A. Diaz and D. Lovell. Book : IBM XML Dataset, http://www.alphaworks.ibm.com/tech/xmlgenerator.

[17] G. Gou and R. Chirkova. Efficient Algorithms for Evaluating Xpath over Streams. *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 269 – 280, 2007.

[18] T. Grust. Accelerating XPath Location Steps . *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 109 –120, 2002.

[19] A. K. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. *SIGMOD*, pages 219 – 430, 2003.

[20] Z. Ives, A. Halevy, and D. Weld. An XML Query Engine for Network-Bound Data. *VLDB*, pages 380 – 402, 2002.

[21] H. Jiang, W. Wang, H. Lu, and J. Yu. Holistic Twig Joins on Indexed XML Documents*. *Proceedings of the 29th VLDB Conference, Berlin, Germany*, 2003.

[22] V. Josifovski, M. Fontoura, and A. Barta. 'Querying XML Streams . pages 197 – 210, 2004.

[23] D. Olteanu, T. Kiesling, and F. Bry. An Evaluation of Regular Path Expressions with Qualifiers against XML Streams . *ICDE*, pages 702 – 704.

[24] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward, 2002.

[25] F. Peng and S. S. Chawathe. XPath Queries on Streaming Data . *SIGMOD*, 2003.

[26] S. Prakash, S. S. Bhowmick, and S. Madria. Efficient Recursive XML Query Processing Using Relational Database Systems. *Data Knowledge Engineering*, 2006.

[27] P. Ramanan. Evaluating an XPath Query on a Streaming XML Document. *COMAD, India*, 2005.

[28] A. Schmidt. Xmark:An XML Benchmark Project, http://monetdb.cwi.nl/xml.

[29] D. Suciu. Treebank:XML data repository, http://www.cs.washington.edu/research/xmldatasets.

[30] W3C. Extensible markup language (xml) 1.0 (third edition). November 2004, http://www.w3.org/TR/2004/REC-xml.

[31] G. Zhang and Q. Zou. QuickXScan: Efficient Streaming XPath Evaluation. *International Conference on Internet Computing*, pages 249–255, 2006.