Intelligent Ordered XPath for Processing Data Streams

Muath ALRAMMAL
Laboratory of Algorithmics, Complexity and Logic (LACL).
Université Paris Est.
muath.alrammal@univ-paris12.fr
malrammal@innovimax.fr

Gaëtan HAINS
Laboratory of Algorithmics, Complexity and Logic (LACL).
Université Paris Est.
gaetan.hains@univ-paris12.fr

Mohamed ZERGAOUI
Consulting, Training and XML Development.
Innovimax SARL.
mzergaoui@innovimax.fr

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.

Terms: Algorithms, Performance, Design.
Keywords: XML, XPath, Query Processing, Streams.

Introduction

The extensible markup language XML (W3C 2006) 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 (J. Clark and S. DeRose 1999) and XQuery (Boag et al. 2007) 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. 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.

Query evaluation process of XML data streams raises many challenges compared to non streaming environment (the recursive nature of XML document, the single sequential forward scan of an XML streams, also the presence of descendant axes and the predicates in the XPath query). Better explanation for these challenges as follows:

• During the evaluation process of XPath queries which include predicates, we may encounter potential solutions before we reach the required data to 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.
• 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 current Unordered XPath expressions (unordered predicates and axes) will increase the number of the buffered potential answer nodes before we encounter the required data to evaluate the predicates to decide their satisfaction. In the worst case the buffer size will reach the document size.

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 streamed XML data is modeled as a sequence of SAX (Brownell 2002) 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 (J. Clark and S. DeRose 1999) 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 known as the data model. XPath has a particular importance for XML applications since it is a core component of many XML processing standards such as XSLT (Clark 1999) or XQuery (Boag et al. 2007). It has a natural subset (fragment) that can be used for matching (testing whether or not a node matches a pattern) (Berglund et al. 2007). Therefore, we classify XPath based on its fragment as the following:

- XPath 2.0: it is the largest fragment of XPath, for precise information about its grammar see (Berglund et al. 2007).
- XPath 1.0: it is a sub fragment of XPath 2.0, for precise information about its grammar see (J. Clark and S. DeRose 1999).
- Forward XPath: we define Forward XPath as the following:

  a sub fragment of XPath 1.0 consists of queries that have: child, descendant axis. nodeTest which is either element, wild card. Predicate with (or, not, and). And arithmetic.

For a precise understanding of Forward XPath, we illustrate its grammar in figure 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 ‘.’/.

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

**Figure 2: Grammar of Forward XPath.**

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 (Olteanu et al. 2002) to reduce more general axis operations to forward ones.

Concerning query evaluation process, 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 as follows: 1) streaming algorithm evaluates the predicates only when the closing tags of the streaming elements are encountered. 2) Eager streaming algorithm evaluates the predicates as soon an atom in a predicate is evaluated to true.

**Recursion in XML Data** occurs frequently in XML data in practice (Choi 2002), where some elements with the same name are nested on the same path in the data tree. In (Bar-Yossef, Fontoura, and Josifovski 2004), they define 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 3 illustrates the depth of document $D$ w.r.t the query $Q$ $//A//B//C//K$.

**Figure 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 represents [j/node()] and the result node which is in this example the shaded node $K$.

It is obvious from figure 3 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, 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 3 the used XPath expression can be considered a Simple XPath, for more Complex XPath expression, see figure 5, it benefits from the grammar of figure 2. Moreover, our query is Unordered (axes and predicate position) (note that, further explanations about Ordered and Unordered XPath can be found in sections 4 and 5.) because node $C$ in $Q$ can be before node $K$ or after.

### Document Depth $d_D$

$d_D$ is the length of the longest root to leaf path in the tree. In our example in figure 3, document depth is the path from root node $A_1$ to the leaf node $K_8$.

The rest of the paper is organized as follows: In section 2, we present the related work, a survey about the different streaming algorithms and their streamable XPath fragments. In section 3, we examine and compare the XPath fragments of the streaming algorithms in the related work (section 2) with the fragment of Forward XPath. Next, in section 4 we explain the critical parameters that affect the complexity of the current streaming algorithms, furthermore, we present our motivations for a new approach. We propose a new approach for processing XML data streams in section 5. After that, we conclude in section 6.

### Related Work

A 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 processing approach they use. Most of them are automata based [XPush (Gupta and Suciu 2003), XSQ (Peng and Chawathe 2003), SPEX (Olteanu, Kiesling, and Bry 2003), XMLTK (Avila-Campillo et al. 2002)] or Parse tree based [(Chen, Davidson, and Zheng 2006), (Barton et al. 2003), (Bar-Yossef, Fontoura, and Josifovski 2004), (Gou and Chirkova 2007)].

In (Gupta and Suciu 2003) authors 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. An application example for this work is XML message brokers.

AFilter (Candan et al. 2006) 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 wildcards, but does not support predicate filters. Similar to XPush (Gupta and Suciu 2003), it is designed to evaluate a large set of queries.

In TurboXPath (Josifovski, Fontoura, and Barta 2004) 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 of which have been evaluated to true.

(Bar-Yossef, Fontoura, and Josifovski 2004) supports the parent and the ancestor axes in addition to that Forward XPath axes. It builds parse tree which is used to 'predict' the next matching nodes and the level in which they have to occur, e.g. consider the query //CD and a matching of C in level 3. Then the next interesting matching would be a node D in level 4.

XSQ (Peng and Chawathe 2003) and SPEX (Olteanu, Kiesling, and Bry 2003) propose a method for evaluating XPath queries over streaming data to handle closures, 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 a non-deterministic push-down transducer (PDT) that is generated for each location step in an XPath query, these PDTs are combined into a hierarchical pushdown transducer in the form of a binary tree. XSQ eagerly evaluates queries and it is fully implemented thus give us the possibility to compare our algorithms with it.

The approach presented in (Chen, Davidson, and Zheng 2006) 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 (Chen, Davidson, and Zheng 2005) authors aim to achieve polynomial time complexity in both data and query size for evaluating XPath queries on XML data stream, based on that, they propose ViteX System that is composed of four modules: 1) XPath parser. 2) TwigM builder. 3) SAX parser. 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 to use an encoding technique, therefore, 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.

(Zhang and Zou 2006) introduces a streaming XPath algorithm (QuickXScan). It is based on the principles similar to that of attribute grammars. Authors, model an XPath query with a query tree. In the 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 (Jiang et al. 2003), therefore, QuickXScan extends the idea of compact stacks in a technique called matching grid, which is used also in (Ramanan 2005).

The authors of (Chen et al. 2004) present 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 XML stream for data receivers with constrained computing abilities "binary encoding". The EXPedited 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 (DeHaan et al. 2003), (Grust 2002) for efficient query evaluation based on XML node labels for XML data stored in the database.

In (Stefan Böttcher and Rita Steinmetz 2007) a SAX based approach is introduced to evaluate XPath queries that support all axes of Core XPath. 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. They have 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 (Gou and Chirkova 2007) authors propose two algorithms to evaluate XPath over streams, they are 1) Lazy streaming algorithm (LQ), 2) Eager streaming algorithm (EQ). Algorithms accept XML document as a stream of SAX events. The used fragment of XPath called Univariate XPath. The goal of both algorithms is to prove that Univariate XPath
Comparison Process between Forward XPath and the XPath Fragments of the Streaming Algorithms.

In the related work that is mentioned above, we have examined the different fragments used of the algorithms (see table 2) and compared them with the fragment of Forward XPath (figure 2). The purposes of this comparison are as follows:

- Knowing the relationships between the fragments of the different algorithms.
- Estimate the boundaries or the frontiers area of the streamable fragment of XPath.

For further explanation, figure 4 illustrates the intersection between the different algorithms, and the boundaries of the streamable XPath fragment.

The red dashed line represents the boundary of the streamable XPath fragment. After analysis the different XPath fragments in table 2, we have noticed the following:

- 6/11 of the streamable XPath fragments are sub fragments of Forward XPath e.g. Univariate XPath (Gou and Chirkova 2007), AFilter (Candan et al. 2006).
- 3/11 of the streamable XPath Fragments can support specific parts of XPath 1.0 grammar (reverse axis, or functions). In the same time they do not support all the grammar of Forward XPath (predicate with or, not) e.g. XSQ (Peng and Chawathe 2003), XPA (Stefan Böttcher and Rita Steinmetz 2007).
- 1/11 of the streamable XPath fragments support specific grammar of XPath 2.0 (XQuery features for, let, where) but they do not support all the grammar of XPath 1.0 and Forward XPath, e.g. TurboXPath (Josifovski, Fontoura, and Barta 2004).
- 7/11 of the streamable XPath fragments, do not support predicates with (or, not).
- All algorithms use Unordered XPath.

Therefore, in comparison to all these approaches, we propose the notion of Intelligent Ordered and Oriented Forward XPath (section 5) taking in consideration the critical complexity parameters that are explained in the next section.

Critical Complexity Parameters

Usually, the caching space costs of stream-querying algorithms depends 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( Figure 5 is an example of a query that involves *-nodes or the same name nodes which increases the buffer size and processing time) and does not exceed $(|Q|d_D)$ in the worst case. Also, we can measure the CPU time performance of stream-querying algorithms by the following equation:

$$T_p = t_{all} - t_{input} - t_{output}$$

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 many parameters, we explain them as follows :

Structure of XML Document Data

Documents may have varied structures, for instance, shallow XML documents (Wide) that do not include recursive elements (figure 6). In this case the caching space costs of the stream-querying algorithm is almost negligible. An example for this
Table 3: Different Dataset Structures.

<table>
<thead>
<tr>
<th>Structure</th>
<th>XMark</th>
<th>Book</th>
<th>TreeBank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Size</td>
<td>113MB</td>
<td>12MB</td>
<td>82MB</td>
</tr>
<tr>
<td>Nbr. of nodes</td>
<td>1868K</td>
<td>114K</td>
<td>2437K</td>
</tr>
<tr>
<td>Max/Avg depth</td>
<td>125/6</td>
<td>22/19.4</td>
<td>36/7.6</td>
</tr>
</tbody>
</table>

Type of XML document is XMark (Schmidt) which is a famous benchmark dataset that allows users and developers to gain insights into the characteristics of their XML repositories. See table 3 which indicates the maximum depth of this dataset that reaches 12 (not deep), and it has a large data size 113MB.

Others documents are semi deep and recursive e.g. Book dataset (Diaz and Lovell), 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 3, it has a size of 12 MB which is not enormous and a maximum depth that reaches 22 which is quite deep 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) increases the buffering space and processing time. We can find also documents with narrow deep structure, e.g. TreeBank (Suciu), here one can recognize the structure of the document from the maximum depth in table 3, it is 36, moreover, the average depth which is 7.8. The existence of these properties in the document are strongly related to the algorithmic complexity of stream processing.

**The XPath Expression (Recursion)** Simple XPath expressions may not require huge buffering space size, see figure 3. On the other hand, the existence of descendant axis ‘//’, the wildcard * or the same name node in the expression that is used to search deep and recursive XML document will increase the buffering size and processing time enormously, because we will be forced to buffer large number of potential answer nodes. An example of this XPath expression type is 

\[
//A//B//C
\]

see figure 5. We call this expression Complex XPath.

![Complex XPath Expression](image)

![Figure 5: Complex XPath Expression.](image)

**Query Evaluation Strategy** The used strategy to evaluate the XPath expression may affect the size of the buffering space and processing time. Note that B is measured by the maximum number of potential answer nodes buffered at a time during the running 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 \[//A//F//C\] as it is shown in figure 6. In the 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.

![Lazy and Eager Approaches](image)

**Motivation Examples for New Approach**

**Descendant Axis Problem** In the 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. 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 \(//A//B//C//D//E//F//G//I//H//J\), which is illustrated in figure 7.

![Twig Pattern of an XPath Query Q](image)

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. In the worst case node H can be after node D, thus 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 8.

![Figure 8: XML Documents](image)
Obviously, in both documents attributes are unordered, just because attribute specification markup is unordered, which implies, the order of attribute specifications in a start-tag or empty-element tag is not significant (W3C 2006). While it is not the same case 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 (Bar-Yossef, Fontoura, and Josifovski 2004) and (Gou and Chirkova 2007) 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 explosion of the buffering space size. An example for that is launching Complex XPath expressions to search the dataset Treebank (table 3) using XSQ (Peng and Chawathe 2003). 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.

**Metadata Schema and Intelligent Ordered XPath**

The previous observations encouraged us to search a new, preemptive approach to optimizing query processing in streaming mode. The idea consists in providing metadata with structural data and search-orienting information. This information is either provided by the author of the query or accumulated over successive queries. The concept is supported by our first experiments in (Alrammal, Hains, and Zergaoui 2009).

- **Metadata** is data associated with objects which relieves their potential users of having full advance knowledge of their existence or characteristics. We can define metadata as 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.

In our case, the syntax of metadata can be as follows:

\[\langle A_4 \rangle[@order="2.2.1.1"]\subTreeSize="4" \ldots \rangle\] in this example, the attribute order provides us with the path from the root to node \(A_4\) (Dewey Path), while the attribute subTreeSize gives size of the subtree of the node \(A_4\). See figure 9.

- **Ordered XPath** has been investigated before in the relational databases. (Choi et al. 2007) Presents a novel approach to efficiently evaluate Ordered XPath queries in relational databases. A scheme extending SUCXENT++ (Prakash, Bhowmick, and Madria 2006) was proposed to support the processing of ordered axes and predicates while maintaining its original properties.

In this article we define Ordered XPath as the following: a language that treats an XML document as an ordered tree to locate specific elements (and attributes, processing instruction, etc...) considering that it has ordered axes and predicate position.

The grammar of Ordered XPath is almost the same as the grammar of Forward XPath in figure 2. The only difference is ordered axes and predicate position. Concerning ordered axes we use a Dewey Path as shown in figure 9. The predicate positions can be constrained as follows: 

\[\langle//\text{node()}[@\text{order}]\rangle > \langle//\text{node()}[@\text{order}]\rangle\] , it means the Dewey path of the first node is larger than the Dewey path of the second node. In other words the first node is after the second node in the document information order. For better understanding please see the examples of section (5.1).

- **Metadata Schema** To support Ordered XPath queries, the order information of nodes must be captured by the schema to process these queries efficiently. Figure 9 illustrates the information order of document \(D\) (ordered axes).

![Figure 9: Document Information Order](image)

The purpose of our metadata schema is to help us locate the part of XML documents we search by providing information about the structure and the content of this document. Thus we can profit from Ordered XPath property to efficiently retrieve the required data that correspond to the XPath expression.

- **Intelligent Ordered XPath.** 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 we introduced very important motivations that explain the idea and complexity cost of Ordered XPath. Based on that observation, we define Intelligent Ordered XPath as the following: an XPath language that profits from two main features:

  - Ordered XPath (ordered axes, predicate position).
  - Metadata Schema (metadata, document structure),

in order to improve the buffering space and the processing time.

Next we will present examples to highlight the difference between using Intelligent Ordered XPath and Unordered XPath.

**Intelligent Ordered XPath vs. Unordered XPath**

Comparison examples between Intelligent Ordered XPath and Unordered one are as follows:

**Example 1:** let us suppose that we use the dataset TreeBank which is narrow and deep (table 3) and Lazy strategy of figure 6. Though Eager has a better performance on this
example, we use Lazy 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 Intelligent Ordered XPath which is provided in information as follows (start after node order 1000 because 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 should search is somewhere after page number 1000 in the book".

In the same example imagine that Intelligent Ordered XPath was provided in information as follows (always search to the 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:** figure 9 represents an XPath expression

\[
//A//B//C\big[//D\big]\big[\big[//E\big]\big]\big[\big[//F\big]\big]\big[\big[//G\big]\big]
\]

and an XML document with (information order)

**Case(1):** if XPath is provided in information as follows (start at the axis order 2 of node $A$) then, the syntax of XPath expression that respects the grammar of Ordered XPath will be:

\[
//A//B[\text{order} = "2"]//C//D//E//F//G
\]

in this case, based on figure 9 we do not need to buffer the following nodes: $B_4, B_5, C_1$ though they would otherwise match the query.

**Case(2):** XPath is provided in further information as follows: (start at the axis order 2 of node $A$, and node $C$ should be after node $K$) then, the syntax will be:

\[
//A//B[\text{order} = "2"]//C//D//E//F//G//K
\]

in Case(2) we know that the position of the predicate node $C$ is to the right(after) of the result node $K$. Thus, we avoid to evaluate the predicate node $C_5$ which is time investment. Note that Case(2) has the same (ordered axes) as Case(1), while Case(1) has not (ordered predicate position) as in Case(2).

**Example 3:** Figure 10 is an example application of XML streaming quoted in (Gou and Chirkova 2005). It concerns Selective Dissemination of Information (SDI) whereby publishers and subscribers exchange flat XML documents through a filtering system in charge of realizing queries. SDI is an instance of event processing because stream querying algorithm is triggered by the arrival of a new data stream. Our proposed scheme is applicable to such a system in the following manner: Metadata helps to improve the algorithm performance, and conversely collected performance data is fed back into the system to enrich the metadata thus narrowing future queries as explained in our technical report (Alrammal, Hains, and Zergaoui 2009).

**Advantages of Intelligent Ordered XPath**

It is true that sometimes by restricting the query to Intelligent 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 Intelligent Ordered XPath that account for the critical complexity parameters. They are summarized as the following:

- **XPath complexity:** in both cases Simple XPath see figure 3 or Complex XPath see figure 5, the buffering size performance will be improved due to the property of the ordered (axes and predicates position) of Intelligent Ordered XPath, while it is not the same case for Unordered XPath.
- **XML document structures:** Intelligent 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 .
- **Query evaluation approach:** it is no less efficient as the various used approaches of query evaluation like Lazy and Eager. In the same example of section , we explained how to avoid the worst case of buffering size with Lazy. Note that Eager may has a better performance. Therefore, using Intelligent Ordered XPath with the Eager will also result in a better performance than using it with the Lazy. Moreover, saving the space comes as we have shown by reducing the buffering process. Thus, for a given strategy Eager/Lazy, ordered (axis and predicate position) will decrease time by avoiding unnecessary buffering operations.

We can conclude based on the previous examples, that the efficiency of Intelligent Ordered XPath does not only improve the buffering space, but also the processing time.

**Conclusion**

In this article we introduced a survey about different approaches used to evaluate XPath expressions over XML data Streams, we categorized the different works by the processing approach they use into two categories. First, *Automata based* like [XPush (Gupta and Suciu 2003), XSQ (Peng and Chawathe 2003), SPEX (Olteanu, Kiesling, and Bry 2003), XMLTK (Avila-Campillo et al. 2002)]. Second, *Parse tree based* like [(Chen, Davidson, and Zheng 2006),(Barton et al. 2003),(Bar-Yossef, Fontoura, and Josifovski 2004),(Gou and Chirkova 2007) ]. Furthermore, we estimated the boundary area of the streamable XPath fragments of the streaming algorithms, to the best of our knowledge we are the first who did this process. In addition, we introduced variant examples to explain the critical complexity parameters of the streaming algorithms, like query evaluation strategy, the XPath expression (recursion) and the structure of XML document. Based on that, the notion of the intelligent Ordered XPath for processing efficiently large XML data streams was proposed to accounts for these critical parameters.
Future Work
We are currently designing an algorithm for processing Ordered XPath, and when implemented its performance will be compared with stream-querying algorithms like (Gou and Chirkova 2007), XSQ (Peng and Chawathe 2003) and (Chen, Davidson, and Zheng 2005).

In (Alrammal, Hains, and Zergaoui 2009) we show performance measurements for token search depending on document structure and size. The results are analyzed to estimate possible gains from the scheme we suggested above. To minimize the effort in providing ordering information in the metadata we will investigate evolving query algorithms that accumulate the statistics of query ordering for a given document, thus gradually improving performance over successive queries.

Acknowledgments
The first author thanks Innovimax for doctoral funding under a CIFRE scheme of the French government.

References
Chen, Y.; Davidson, S. B.; and Zheng, Y. 2006. An Efficient XPath Query Processor for XML Streams. Proc. 22nd ICDE.
Peng, F., and Chawathe, S. S. 2003. XPath Queries on Streaming Data. SIGMOD.