Realistic Performance Gain Measurements for XML Data Streaming with Meta Data

Muath Alrammal, Gaétan Hains, Mohamed Zergaoui

Realistic Performance Gain Measurements for XML Data Streaming with Meta Data

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

August 2009

TR–LACL–2009–4

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–2009–4**

M. ALRAMMAL, G. HAINS, M. ZERGAOUI

Realistic Performance Gain Measurements for XML Data Streaming with Meta Data

Abstract

This report is part of our ongoing project on the optimization of stream processing for XPath queries on XML documents. The fact that XML documents can be large relative to query-processing memory is one of the reasons to favour streaming over in-core processing. But even in streaming mode query processing uses storage and hence time proportional to the depth of sub-documents. With the goal of querying large documents on mobile devices, or very large ones on normal machines, we have designed a scheme whereby exhaustive searching can be traded against streaming performance.

Our scheme uses query meta-data which restricts the search to a subset of the document. In an earlier report we have measured the maximal theoretical gains possible with perfect "chance" on choosing the meta-data for synthetic documents, and have found them attractive. In this report we verify two further necessary properties of our scheme, namely 1. the maximal theoretical gains are confirmed on a realistic data set 2. randomly-chosen meta-data also leads to substantial performance gains, which we quantify against the percentage of actual solutions found for the query.

1 Introduction

XML [23] has become the standard for data exchange and representation between applications such as GML [6] for spatial data, ebXML [8] for e-commerce, and OWL [19] for the next generation web. When more and more systems started using XML, processing XML documents became a critical and integral part of those applications. As a result, a lot of research concerning indexes [17], [16], [25], labeling [24], and structural joins [15] have been conducted for XML data.

Often, data is too 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 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 automata and customized algorithms for query evaluation.

Recently, research on processing XML data stream has been conducted [2], [7], [14], [5], [20]. Among various XML query languages, XPath [13] is generally used as the description language for the users since it is simple, but powerful enough to address any part of an XML document.

Data streaming is a necessary and useful technique to process very large XML documents but poses serious algorithmic problems. The challenges it poses are how to efficiently handle large amounts of XML documents or documents too large to be processed in memory. The critical complexity parameters of the streaming algorithms are: the query evaluation strategy, the XPath expression (path pattern/twig pattern with) and the structure of the XML document (depth/width/size).

The rest of this report is organized as follows : in section two we present the related work. Section 3 contains our motivations and objective. In section 4 we present our proposal searching range. Section 5 contains the experiments performed and their analyze. In section 6 we summarize our work and conclude. Finally, in section 6 we present the future work. This article presents a new approach a new approach to the optimization of XML data stream querying. Our concept is to rely on the structure of the document, the information it contains, and Oriented XPath: forward XPath augmented with meta-data for orienting the search. A prototype OXPath was implemented to account for the challenges of querying XML data streams and the critical complexity parameters of streaming algorithms. OXPath consists of two models 1) Perfomance Model which is a mathematical model of performance (Time/Space) prediction. This model provide us in advance with the expected Time/space to satisfy the XPath Expression. 2) Data Oriented Model to predicate future queries and the location of their results from past queries on a fixed XML dataset. An example of past queries importance is a 2005 study of Yahoo's query logs revealed 33% of the queries from the same user were repeat queries and that 87% of the time the user would click on the same result. This suggests that many users use repeat queries to revisit or re-find information 1.

Space and time measurements are applied to state the possible performance gains from OXPath to orient searching in XML data streams with respect to different structures of XML

documents (width/depth/size). Our experiments presented here come from a realistic XPath query-processing algorithm.

2 Related work

A large amount of research 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 as follows (1) Stream-filtering : determining whether there exists at least one match of the query Q in the XML document D (boolean), for example XTire [4]. (2) Stream-querying: which part or parts of the document D match the query Q? It implies outputting all nodes in an XML document D (answer nodes) which satisfy a query Q at its result node. for example XSQ [20].

Various stream-filtering systems have been proposed. XFilter [1], a filtering system, is the first one that addresses the processing of streaming XML data. In XFilter, each XPath query is transformed into a separate deterministic finite automaton (DFA). In YFilter [7] , a set of XPath expressions is transformed into a single non-deterministic finite automaton (NFA) by sharing the prex. Another filtering system is XTier [4] which supports tree shaped XPath expressions containing predicates. In XTier, a path expression is decomposed into several substrings each of which is a sequence of labels, then each substring is recorded in a table, called Substring-Table. The information in the table is used to check for partial matchings.

Recently, to improve the performance of stream-filtering, XPush machine [10] was proposed. It process large number of XPath expressions with many predicates per query on a stream of XML data. XPush machine transforms each XPath expression into an NFA and by grouping the states in a set of NFAs, a single deterministic pushdown automaton (DPA) is constructed lazily. This is similar to the algorithm for converting an NFA to a DFA [11].

However, stream-filtering systems 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.

Also, different stream-quyering systems have been proposed. In TurboXPath [14] which is a **lazy** system for evaluating XQuery-like , 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. Other systems are XSQ [20] and SPEX [18], they 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. In [5] authors propose 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 [3].

Recently [9] propose a lazy system for stream-querying called LQ. In this system the input query is translated into a table throughout stream processing and statically stored on the memory. Whenever a matching of the xpath expression is found within the data stream, the evaluation process is occurred only at the closing tag of the streaming element(root element of the XPath expression). LQ was proposed to handle two challenges of stream-querying were not solved by [20] and [5]. These challenges are : the recursion in the XML document and the existence of the same label-node in the XPath expression. LQ is the algorithm used in our experiments, in section 4 we justify our choice of using this algorithm.

In the next section we present our motivations then our objective to optimize the stream-querying process, which is based on our motivation.

3 Motivations and Objective

As we mentioned before, data streaming is a necessary and useful technique to process very large XML documents but poses serious algorithmic problems. Below we discuss few of these challenges :

- Unlimited XML document Size : an XML document is often large, sometimes very large, thus using the data streaming systems in the related work still requires resources (memory, disk storage, and network bandwidth). In our situation these resources might do not fit the capacity of the end-user pc.
- User resources (Machine): in certain situations processing the XML document occurs through portable devices with small memory sizes, thus we need to optimize the streaming process to fit our resources.
- Importance of past queries : an example of past queries importance is a 2005 study of Yahoo's query logs revealed 33 % of the queries from the same user were repeat queries and that 87 % of the time the user would click on the same result. This suggests that many users use repeat queries to revisit or re-find information [22].

Based on the above mentioned, further optimization for processing XML data stream is required. Our objective is to improve the stream-querying process even thought this improvement may decrease the possibility of finding all matches in the XML document.

To optimize the stream-querying process we propose *searching range* which augment the XPath expression with mete data to orient searching process. In this syntax, we search over a subset of the XML document as specified by the searching interval of searching range. Searching range can be provided to our algorithm implicitly by predicting future queries and the location of their results from past queries on a fixed XML dataset or explicitly based on either the knowledge of the user about the searching range or a decision taken by him to chose searching range randomly.

This process yields a *semi-stream-querying* algorithm which is an algorithm returns true (correct) if it finds the result(s) , in the same time, it does not guarantee to return the result(s) if it exists in the XML document (incomplete). This is why we call it correct but incomplete.

In the next section, we introduce our proposal.

4 Oriented XPath and searching range

4.1 Concept definition

As we mentioned before, stream-filtering systems 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. While stream-querying systems imply outputting all matches in an XML document D which satisfy a query Q, thus poses algorithmic challenges in a particular situation like using a portable devices with limited resources(small memory size) with very large XML documents.

In this report we define *stream-scanning*: to process XML data stream with minimal resources, this process searches the position of a specific element in XML document D and it does not require caching nor buffering. Where *caching* : is to cache specific nodes of the XML document D in the run-time stacks, these nodes correspond to the axis node of the query Q. And *buffering* : is to buffer the potential answer nodes of XML Document D. These nodes correspond to the answer node of Q.

Our concept is to rely on the structure of the document, the information it contains, and *Oriented XPath*: an XPath augmented with searching range (meta data) for orienting the search as described below. *Searching range* is a meta data augmented to XPath expression, it consist of two points : start-search (lt) and end-search (lt). *start-search (lt)* is the moment we stop stream-scanning and start stream-querying based on its lt (Location tag) value. It provides the algorithm in the location of the tag in the XML document, more precisely, the location of the

tag of the first node X in the XML document order that corresponds to the root node of XPath sent. While, *end-search(lt)* is the moment we decide to stop processing the XML document based on its lt (Location tag) value. It provides the algorithm in the location of the tag(lt) of the last node Y in the XML document order that corresponds the answer node of XPath sent.

To explain the idea of the location tag, we present the example below :

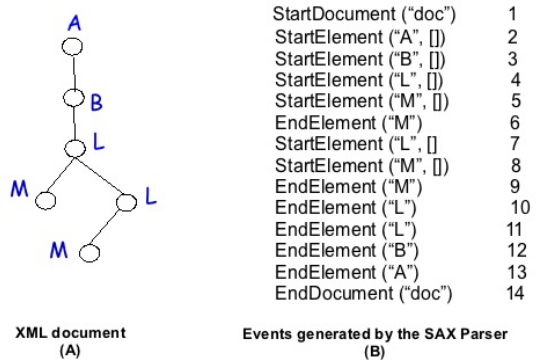

Figure 1: An XML document and it SAX parser events

Figure 1 represents an XML document and its events generated by the SAX parser. In figure 1 (B) the integer number to right of each event of the SAX parser represents the location of the tag (lt) of the corresponding event(element in the XML document). Based on this example, the values of start-search(lt) and end-search(lt) of the XPath //L/M are 4 and 8 receptively.

The idea is to first learn from past queries to "guess" in advance those parts of the document that contain the information to be found. This information may also be explicitly provided by the user of a querying interface, for example one may want to search a very large documents knowing that the information sought is located in the second half of the document. Whatever the source of metadata (and this will be the subject of future research) we plan to use it to direct search closer/faster *stream-scanning* to those parts of the document where *stream-querying* is to be applied using Lazy query evaluation.

The purpose of learning from past queries could be to provide us in the metadata with the exact element e in document D that matches the query root. The oriented XPath expression including the new meta data: could be for example a query together with a Dewey path for orientation of search.

4.2 Prototype

To enable the empirical study, we build the stream-query system LQ [9], our choice for this system returns to the following raisons: (1) the authors in [9] explain the algorithm in details thus facilitate the implementation process. (2) As we mentioned before, the query evaluation technique of LQ is lazy, thus it is easier to implement than Eegar one. (3) Finally and the most important, LQ was proposed to handle two challenges of stream-querying were not solved by [20] and [5]. These challenges are : the recursion in the XML document and the existence of the same label-node in the XPath expression. Based on that the algorithm complexity (Time/Memory) of LQ is the best between the stream-qyerying systems mentioned in our related work.Our prototype was implemented using the functional language Ocaml [12].

To augment the XPath expression with searching range, LQ was modified and called OXPath. The architecture of OXPath is shown in figure 2.

OXPath takes three input parameters. The first one is the XPath expression that will be transformed to a query table statically using our Forward XPath Parser. The second is the searching range which contain the start-search and end-search points . After that, the main function will be called, it reads the tree data set line by line repeatedly, each time it generates a token. The value of lt (location tag) of the generated token is matched with the lt value of the start-search point of searching range. If the matching condition is true, stream-scanning will stop then we start stream-querying process. Based on the token generated during the stream-querying process, the matching condition of end-search point is checked, if it is false, then a corresponding startBlock (SB) or endBlock (EB) function is called to process this token,

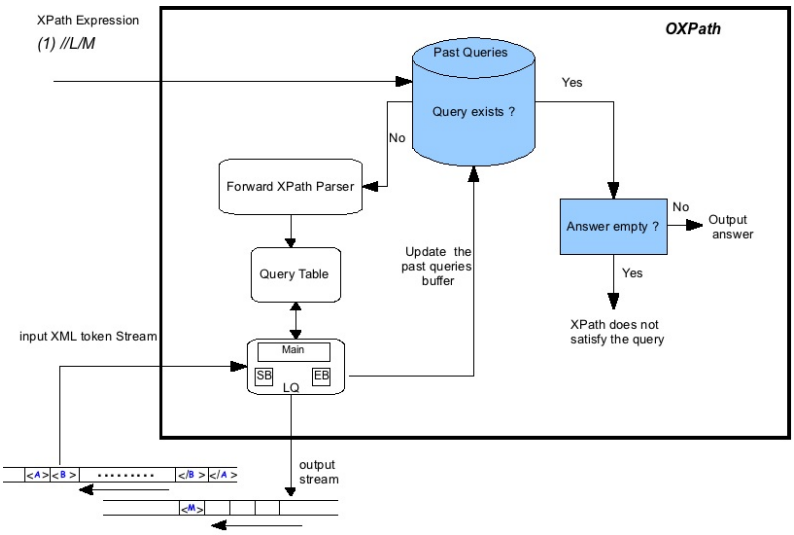

Figure 2: Architecture of OXPath

otherwise the processing the XML document will stop. Finally,during the stream-querying process, if tree data set satisfies the XPath expression, then the main function generates an output stream lazily. Notice that, we stop stream-querying process the moment our condition is true.

The automation process for learning from past queries is not implemented yet, but in order to perform our experiments, the algorithm was modified in a way that it calculates the searching range of the XPath sent for the first time, thus the moment we send the same XPath another time, it augments this XPath with searching range. This means that searching range is implicitly calculated by the algorithm. Furthermore, it possible to provide the algorithm in the searching range explicitly. As we explained before, explicit searching range is either based on the user knowledge about the location of the answer or a decision taken by him to chose searching range randmoly.

5 Experiments

In this section, we demonstrate the efficiency and scalability of `Searching_range`.

5.1 Experiments environment

Experiments were performed using a MacBook with the following technical specifications: Intel Core 2 Duo, 2.4 GHz clock speed , 4 GB RAM.

5.1.1 Data sets

We performed the experiments using the data set treeBank [21] . It is a real data set which has a narrow and deep structure and it has a size of 2500 tokens.

5.1.2 XPath Queries

The XPath queries used for the experiments were generated using Exhaustive testing. Exhaustive testing is a comprehensive testing process to test all XPath queries possible in the data set Treebank. They have the following forms : //A, //A/B, //A/[./B] where A and B can be any element in the data set TreeBank. The purpose of this process is to have a complete statistics related to query evaluation cost.

In this test we measure for each query the best case of the possible time/memory gain i.e. the difference between the cost of a normal streaming query over the whole document vs one where the algorithm simply scans tokens until reaching the answers in an interval indicated by the meta-data. This is of course only possible if the meta-data has been computed from a previous occurrence of the same query over the same document. But our intention is to use

this measurement as a target for performance improvement with more realistic e.g. random meta-data. This last observation will be briefly expanded in our second set of tests below: T4, T5, T6.

Query evaluation cost (Time/Memory) of one XPath is : - Min. Time: Returns the processor time in seconds, used by the program since the beginning of execution till finding the first answer if any, or reaching the end of the document of not - Avg. Time : (Σ (time for each match))/ total number of matches.- Max. Time :returns the processor time in seconds, used by the program since the beginning of execution till the end of XML document processing. - Min. Memory: the total amount of memory allocated by the program size in KB since the beginning of execution till finding the first answer if any, or reaching the end of the document of not. - Avg. Memory: (Σ (memory size in KB for each match))/ total number of matches.- Max. Memory: the total amount of memory allocated by the program size in KB since the beginning of execution till the end of XML document processing.

We classify the tests of our experiments as follows : (1) Test with implicit searching range, these tests are : T1 : XPath with the shape //A , this XPath is sent without searching range. T1′ : XPath with the shape //A, This XPath is sent with ideal searching range obtained from T1 to demonstrate the best time/memory gain possible. T2 : XPath with the shape //A/B , this XPath is sent without searching range. T2′ : XPath with the shape//A/B, this XPath is sent with ideal searching range obtained from T2 to demonstrate the best time/memory gain possible. T3 : XPath with the shape //A[./B], this XPath is sent without searching range. T3′ : XPath with the shape //A[./B], this XPath is sent with ideal searching range obtained from T3 to demonstrate the best time/memory gain possible.

(2) Test with explicit searching range (random choice). In each test, the same XPath was sent 25 times, each time XPath was augmented in searching range as follows : – *size*: is the size of our data set which contains 2500 tokens. *len* is the length of the searching range(interval), it equals size/10. Also, end-seach = start-search/len, and the difference between start-search points each time is 100 tokens. for example : searching range(start-search, end-search) searching range(1, (1+len) . searching range(100, (100+len). The purpose of this type of tests is to demonstrate the possible time/memory gain. Our demonstration will be represented as a histogram for time gain and another one for the memory gain for each test. The tests of this type are : T4 : XPath with the shape //A , this XPath is sent without searching range, then with a varying searching range. T5 : XPath with the shape //A/B , this XPath is sent without searching range, then with a varying searching range. T6 : XPath with the shape //A[./B] , this XPath is sent without searching range, then with a varying searching range.

5.2 Experiments results

5.2.1 Exhaustive testing on TreeBank

5.2.2 T1 and T1′

In this experiment, we show the efficiency of using searching range compared to the existing stream-querying algorithm LQ. figure 3 shows the queries evaluation cost of T1 and T1′ . In this experiment, we calculated the query evaluation cost for 45 positive queries.

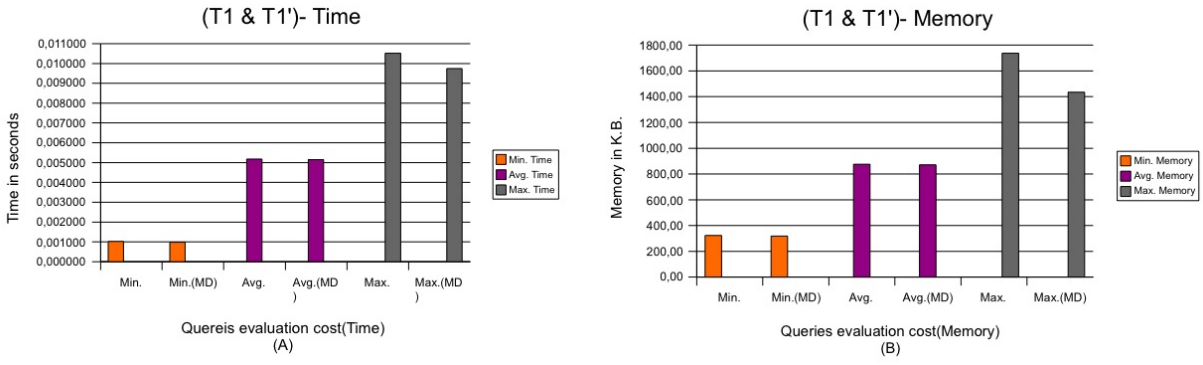

Figure 3: Query evaluation cost for T1 and T1′

Figure 3 (A) shows the queries evaluation cost (time) of T1 and T1′. The Min Time of T1 is 0.001027 s while for T1′ is 0.000985 s, the slight gain of time returns to the start-search point of the searching range. The Avg Time of T1 is 0.005176 s while for T1′ is 0.005146 s , the slight gain of time returns to start-search and end-search points of the searching range. The Max Time for T1 is 0.010512 s while for T1′ is 0.009745 s, the slight gain of time returns to end-search point of searching range. As we see from the results of the experiment in the figure 3 (A) , our idea does not affect the query evaluation cost negatively, on the contrary in specific situations where the interval of the searching range is small, we can have dramatic gain of time. Figure 3 (B) shows the queries evaluation cost (Memory) of T1 and T1′. The Min Memory of T1 is 323.47 K.B. while for T1′ is 317.55 K.B., the slight gain of memory returns to the start-search point of the searching range. The Avg Memory of T1 is 876.73 K.B. while for T1′ is 870.81 K.B. , the slight gain of memory returns to start-search and end-search points of the searching range. The Max Memory for T1 is 1736.96 K.B. while for T1′ is 1435.12 K.B. , the obvious gain of time returns to end-search point of searching range.

5.2.3 T2 and T2′

In this experiment, we show the efficiency of using searching range compared to the existing stream-querying algorithm LQ. We calculated the query evaluation cost for 75 positive queries, 1861 negative queries and for 1936 queries as a total number. figure **??** (A) shows the queries evaluation cost(Time) of T2 and T2′ .

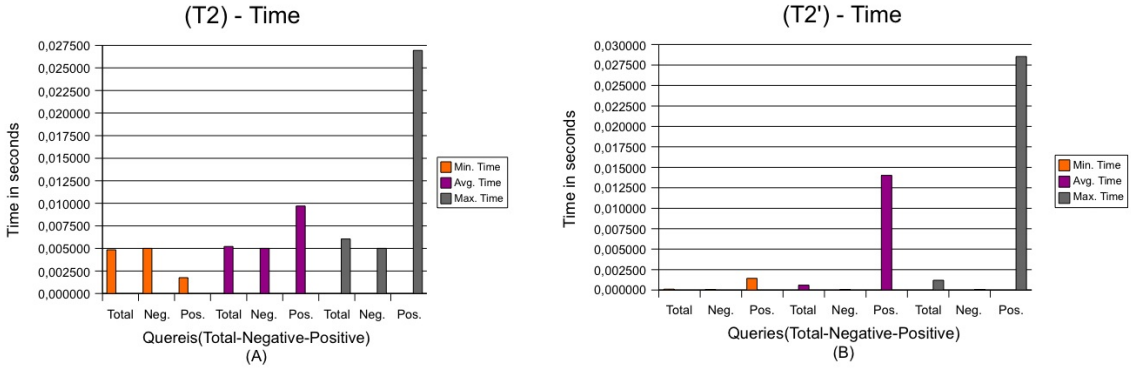

Figure 4: Query evaluation cost for T2 and T2′-(Time)

The Min Time of total, negative and positive queries in figure 4 (A) are 0.004850 s, 0.005004s, and 0.001768 s respectively, while in figure 4 (B) they are 0.0000115 s, 0.000061s and 0.001438 s, the gain of the time returns to start-search point of the searching range and because the number of negative queries is large thus decreases the time. The Avg Time of total, negative and positive queries in figure 4 (A) are 005228 s, 0.005004s, and 0.009708 s respectively, while in figure 4 (B) they are 0.000602 s, 0.000061s and 0.0014010 s, the gain of the time for the total and negative queries returns to start-search and search points of the searching range and because the number of negative queries is large thus decreases the time. Concerning the Avg Time of positive queries , it increases from 0.009708 s to 0.014010 s because as we explained before the evaluation technique used is lazy, thus in specific cases we can not return the last match of the positive query which respects the searching range interval. As a result the average time increases. The Max Time of total, negative and positive queries in figure 4 (A) are 0.006047 s, 0.005004s, and 0.026947 s respectively, while in figure 4 (B) they are 0.001191 s, 0.000061s and 0.028217 s, the gain of the time for the total and negative queries returns to start-search and search points of the searching range and because the number of negative queries is large thus decreases the time. Concerning the Max Time of positive queries , it increases from 0.026947 s to 0.028217 s because of the same raison as it is explained above for the Avg Time of positive queries.

Figure 5 shows the queries evaluation cost (Memory) of T2 and T2′. The Min Memory of total, negative and positive queries in figure 5 (A) are 1675.67 K.B , 1731.32 K.B, and 566.81 K.B respectively, while in figure 5 (B) they are 24.79 K.B, 9.14 K.B and 413.22 K.B, the gain of the memory returns to start-search point of the searching range and because the number of

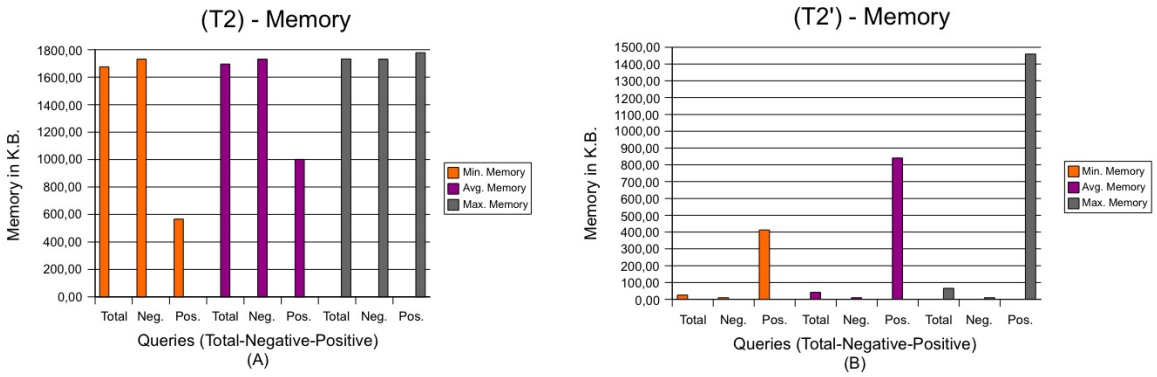

Figure 5: Query evaluation cost for T2 and T2′-(Memory)

negative queries is large thus decreases the allocated memory. The Avg Time of total, negative and positive queries in figure 5 (A) are 1696.32 K.B, 1731.00 K.B, and 1001.25 K.B respectively, while in figure 5 (B) they are 41.36 K.B, 9.14 K.B and 840.75 K.B, the gain of the memory returns to start-search and search points of the searching range and because the number of negative queries is large thus decreases the allocated memory. The Max Memory of total, negative and positive queries in figure 5 (A) are 1733.26 K.B, 1731.00 K.B, and 1778.73 K.B respectively, while in figure 5 (B) they are 65.33 K.B, 9.14 K.B and 1459.50 K.B, the gain of the memory returns to start-search and search points of the searching range and because the number of negative queries is large thus decreases the allocated memory.

5.2.4 T3 and T3′

In this experiment, we show the efficiency of using searching range compared to the existing stream-querying algorithm LQ. We calculated the query evaluation cost for 75 positive queries, 1861 negative queries and for 1936 queries as a total number. Figure 6 (A) shows the queries evaluation cost (Time) of T3 and T3′.

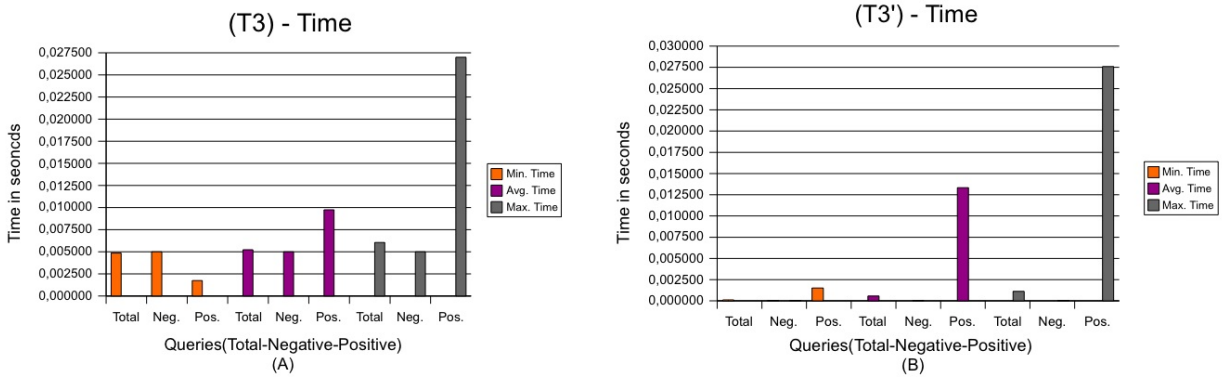

Figure 6: Query evaluation cost for T3 and T3′-(Time)

The Min Time of total, negative and positive queries in figure 6(A) are 0.004842 s, 0.004997s, and 0.001736 s respectively, while in figure 6 (B) they are 0.0000117 s, 0.000060s and 0.001530 s, the gain of the time returns to start-search point of the searching range and because the number of negative queries is large thus decreases the time. The Avg Time of total, negative and positive queries in figure 6 (A) are 005223 s, 0.004997s, and 0.009750 s respectively, while in figure 6 (B) they are 0.000574 s, 0.000061s and 0.000060 s, the gain of the time for the total and negative queries returns to start-search and search points of the searching range and because the number of negative queries is large thus decreases the time. Concerning the Avg Time of positive queries , it increases from 0.009750 s to 0.013338 s because as we explained before the evaluation technique used is lazy, thus in specific cases we can not return the last match of the positive query which respects the searching range interval. As a result the average time increases. The Max Time of total, negative and positive queries in figure 6 (A) are 0.006043 s, 0.0049974 s, and 0.027001 s respectively, while in figure 6 (B) they are ,001126 s, 0.000060s and

0.027589 s, the gain of the time for the total and negative queries returns to start-search and search points of the searching range and because the number of negative queries is large thus decreases the time. Concerning the Max Time of positive queries , it increases slightly from 0.027001 s to 0.027589 s because of the same raison as it is explained above for the Avg Time of positive queries.

Figure 7 (B) shows the queries evaluation cost (Memory) of T3 and T3′.

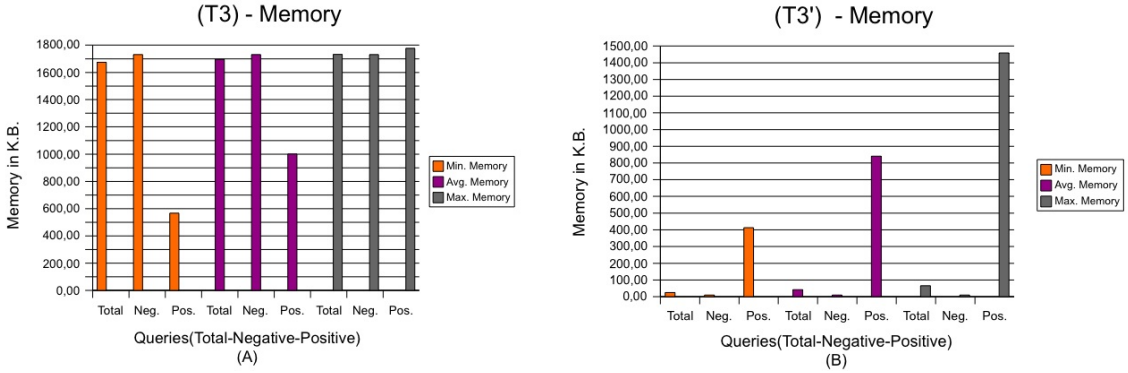

Figure 7: Query evaluation cost for T3 and T3′-(Memory)

The Min Memory of total, negative and positive queries in figure 7 (A) are 1675.06 K.B , 1730.37 K.B, and 566.43 K.B respectively, while in figure 7 (B) they are 24.79 K.B, 9.14 K.B and 413.29 K.B, the gain of the memory returns to start-search point of the searching range and because the number of negative queries is large thus decreases the allocated memory. The Avg Time of total, negative and positive queries in figure 7 (A) are 1695.70 K.B, 1730.37 K.B, and 1000.70 K.B respectively, while in figure 7 (B) they are 41.35 K.B, 9.14 K.B and 840.69 K.B, the gain of the memory returns to start-search and search points of the searching range and because the number of negative queries is large thus decreases the allocated memory. The Max Memory of total, negative and positive queries in figure 7 (A) are 1732.62 K.B, 1730.37 K.B, and 1777.57 K.B respectively, while in figure 7 (B) they are 65.31 K.B, 9.14 K.B and 1459.18 K.B, the gain of the memory returns to start-search and search points of the searching range and because the number of negative queries is large thus decreases the allocated memory.

5.2.5 T4

T4 is a query of the form //A for which we measure the performance gain for meta-data indicating a search interval of 1/10th the size of the document i.e. 250 over 2500 tokens. The test measures time- and space gains over the possible location of this interval in the document. Because the query's answers are distributed across the document, each query has obtained a proprotional fraction of matches. The intersting parameter to observe here is thus the performance gain.

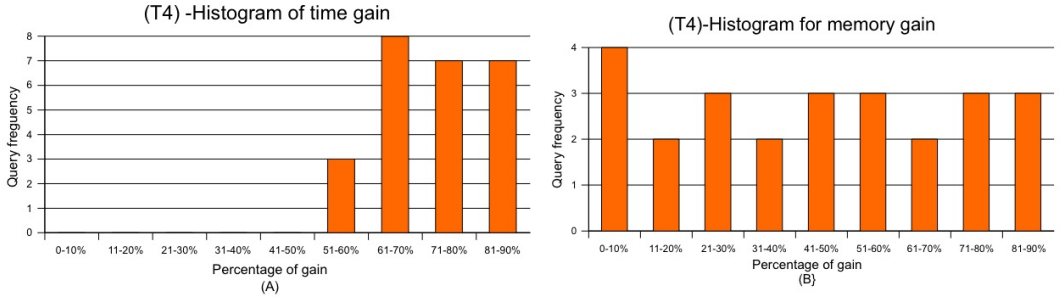

Figure 8: T4 time/memory gain distribution against choice of streaming interval i.e. random meta-data

The results show up to 100% speedup with an average speedup around 75% and never less than a 50% speedup. Space gains are somewhere at 50% on average and are uniformly distributed

between 0% and 90%. There is approximately a 1 on 2 chance of obtaining a 50% to 90% space gain.

5.2.6 T5

T5 is query of the form //A/B for which we apply the same test as T4.

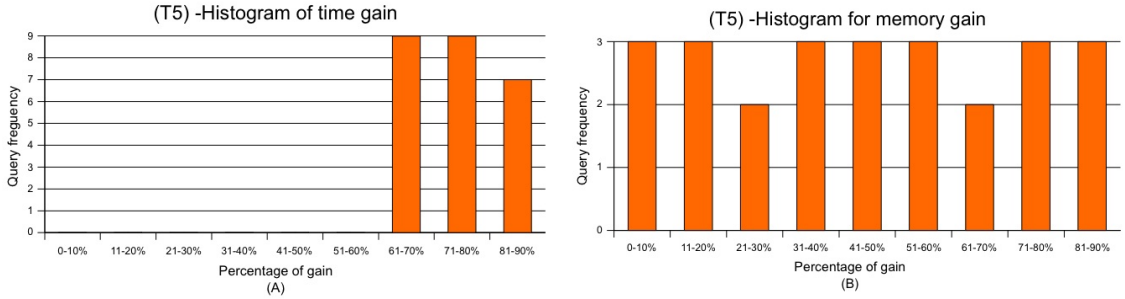

Figure 9: T5 time/memory gain distribution against choice of streaming interval i.e. random meta-data

The observations for T5 are very similar to those for T4 except that the distribution of performance gains is a little more uniform.

5.2.7 T6

T6 is query of the form //A[.B] for which we apply the same test as T4 and T5.

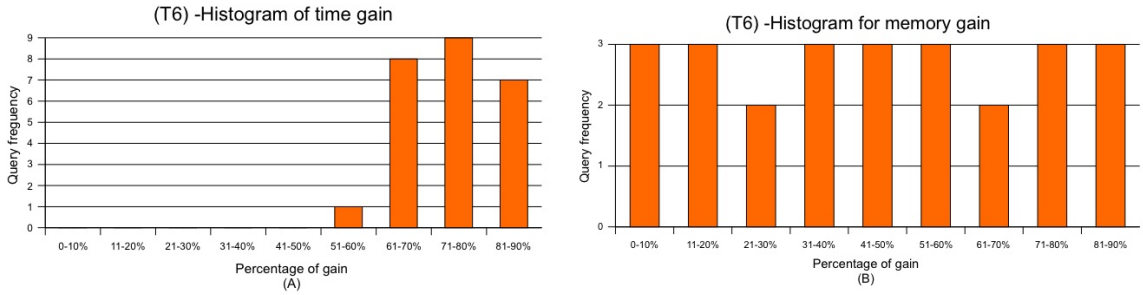

Figure 10: T6 time/memory gain distribution against choice of streaming interval i.e. random meta-data

The results for T6 are almost identical to those for T5.

Overall the results for tests T4, T5 and T6 support our view that even a random choice of interval for the algorithm's searching range will yield substantial time and space gains. It will remain to:

1. confirm these results by varying the test queries and the document;
2. observe and model the ratio between the subset of matches found and the performance gains.

This last value may prove elusive in general, but any empirical or statistical progress on its prediction will allow users of our algorithm to balance the cost of a query with its potential to return most interesting matches to a query.

6 Conclusion

In this report we presented the idea of Oriented XPath that is augmented with meta-data to specify a searching range. A core prototype called OXPath was implemented to have a concrete understanding for the complexity of stream-querying process and to demonstrate the possible time/memory gain of our proposal. Experiments were performed using searching range implicitly and explicitly. We observed from the experiments with the implicit searching range

the following : (1) in all tests Min Time was improved, this improvement returns to stream-scanning. (2) Memory gain depends on interval of searching range, if this interval is small a dramatic memory gain can be achieved, (3) negative queries in streaming is a critical point and very expensive, it affects the performance of stream-querying algorithms dramatically as we stated in our results.(4)

Our current prototype uses our own implementation of Gou and Chirkova's query evaluation algorithm in its *lazy* version. This is relatively clean to implement but it delays outputting the result to the end user and in some cases it decrease the number of matches in the XML document which satisfy the XPath query. An implementation of the eager streaming algorithm is more complex but should yield more predictable performance and improve the space consumption in absolute terms while remaining the same processing speed. Finally, our experiments with random explicit meta-data (T4, T5, T6) have confirmed our implicit assumption that it will be possible to find scanning ranges that produce a reasonable balance of matches with performance gains.

Our future work will first design a quantitative performance model to relate the key parameters in our scheme: match-percentage, space and time gain. Once this model is designed and tested it we will develop a machine-learning algorithm whereby the performance model and hence the best choices of meta-data will be computed from past experience for streaming queries on a given XML document.

References

[1] M. Altinel and M. Franklin. Efficient filtering of xml documents for selective dissemination of information. *Proceedings of 26th International Conference on Very Large Data Bases*, page 5364, 2002.

[2] S. Bose and L. Fegaras. Data stream management for historical xml data. *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, page 239250, 2004.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. *SIGMOD*, page 310321, 2002.

[4] 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*, page 235244, 2002.

[5] Y. Chen, S. B. Davidson, and Y. Zheng. An Efficient XPath Query Processor for XML Streams. *Proc. 22nd ICDE* , 2006.

[6] S. Cox, P. Daisey, R. Lake, C. Portele, and A.Whiteside. Geography markup language (gml) implementation specication. opengis, ¡http://www.opengis.org/docs/02-023r4.pdf¿. (accessed 29 January 2003).

[7] Y. Diao, M. Altinel, M. Franklin, H. Zhang, and P. Fischer. Ecient and scalable filtering of xml documents. *Proceedings of the 18th International Conference on Data Engineering*, page 341342, 2002.

[8] ebXML RequirementsTeam. ebXML Requirements Specication Version 1.06. UN/CEFACT and O. T. Specication. ¡http://www.ebxml.org/specs/ebreq.pdf¿. 8 May 2001.

[9] 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.

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

[11] J. Hopcraft and J. Ullman. Introduction to automata theory, language, and computation. *Addison-Wesley Publishing Company, Massachusetts*, 1979.

[12] INRIA. Objective Caml Language. *http://caml.inria.fr*, 1985.

[13] J. Clark and S. DeRose. XML Path Language (XPath), *http://www.w3.org/TR/xpath* . November 1999.

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

[15] J. Kim. Advanced structural joins using element distribution. *Information Sciences*, page 3300 3331, 2006.

[16] K. Leela and J. Haritsa. Haritsa, schema-conscious xml indexing. *Information Systems, Volume 32*, pages 344–364, April 2007.

[17] J. Min, C. Chung, and K. Shim. An adaptive path index for xml data using the query workload. *Information Systems, Volume 30*, page 467487, September 2005.

[18] D. Olteanu, T. Kiesling, and F. Bry. An Evaluation of Regular Path Expressions with Qualifiers against XML Streams . *ICDE*, pages 702 – 704, 2003.

[19] P. Patel-Schneider, P. Hayes, and I. Horrocks. Owl web ontology language semantics and abstract syntax, w3c candidate recommendation, http://www.w3.org/tr/owl-absyn. 18 August 2003.

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

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

[22] J. Teevan, E. Adar, R. Jones, and M. Potts. History repeats itself: Repeat Queries in Yahoo's query logs. *Proceedings of the 29th Annual ACM Conference on Research and Development in Information Retrieval (SIGIR '06)*, pages 703–704, 2005.

[23] W3C. Extensible Markup Language (XML) 1.0 (fourth edition). 16 August , edited in place 29 September, *http://www.w3.org/TR/xml* 2006.

[24] S. Yi, B. Huang, and W. Chan. Xml application schema matching using similarity measure and relaxation labeling. *Information Sciences*, page 2746, 2004.

[25] L. Zuopeng, H. Kongfa, Y. Ning, and D. Yisheng. An efficient index structure for xml based on generalized sux tree. *Information Systems, Volume 30*, page 283294, April 2007.