Performance measurements towards the optimization of stream processing for XML Data

Muath Alrammal, Gaétan Hains, Mohamed Zergaoui

Performance measurements towards the optimization of stream processing for XML Data

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

January 2009

TR–LACL–2009–9

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–2009–9**

M. ALRAMMAL, G. HAINS, M. ZERGAOUI

Performance measurements towards the optimization of stream processing for XML Data

Abstract

In this report we present the first experimental results to support our proposal [1] as an attempt to optimize stream processing for XML data. A core prototype called `O-Search` was implemented to have a concrete understanding for the complexity of stream-querying algorithms i, with respect to different structures of XML documents (wide, depth, various size). Space and time measurements are applied to state the possible performance gains from the use of meta data to orient searching in stream XML data.

1 Introduction

The extensible markup language (XML) [8], has gone from the latest buzzword to an entrenched e-Business technology in record time. XML is currently being heavily pushed by industry and developers 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 scientific data, and real time news. Query languages like XPath [4] and XQuery [2] 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.

Concerning query evaluation process, usually two operations can be requested. 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 this is called *stream-filtering*. Second, which part or parts of the document D match the query Q? It implies outputting all nodes in an XML data tree D (answer nodes) which satisfy a specified twig pattern Q at its result node *stream-querying*. Different strategies and streaming algorithms were implemented for this purpose, we can categorize some of them based on the time of predicate evaluations in queries. A **Lazy** streaming algorithm evaluates the predicates [1] only when the closing tags of the streaming elements are encountered. While an **Eager** algorithm does that as soon an atom in a predicate is evaluated to true [5].

In our first technical report [1] we surveyed different approaches used to evaluate XPath expressions over XML data streams. Furthermore, we estimated the streamable XPath fragments for the various streaming algorithms. 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 the target XML document. Based on those observations, the notion of Oriented XPath for processing efficiently large XML data streams was proposed as an attempt to optimize for these critical performance parameters.
In this technical report we present the first experimental results to support our proposal in [1].

The rest of the report is organized as follows: in section 2 we introduce the core of our prototype`O-Search`. Section 3 is a detailed presentation of our experiments with all necessary details to allow their reproduction on a different architecture of with similar source code. Finally, in section 4 we analyze the results and evaluate the consequences for our proposed Oriented XPath scheme.

[1] An XPath predicate is contained with-in square brackets [], and comes after the parent element of what will be tested. It filters a node-set with respect to an axis to produce a new node-set.

2 Prototype `O-Search`

We developed the core of a prototype called `O-Search` to have better understanding for the complexity of *stream-querying* algorithms in practice, with respect to different structures of XML documents (wide, depth, various size). The evaluation technique used in our prototype is Lazy. Figure 1 shows the current structure of `O-Search`.

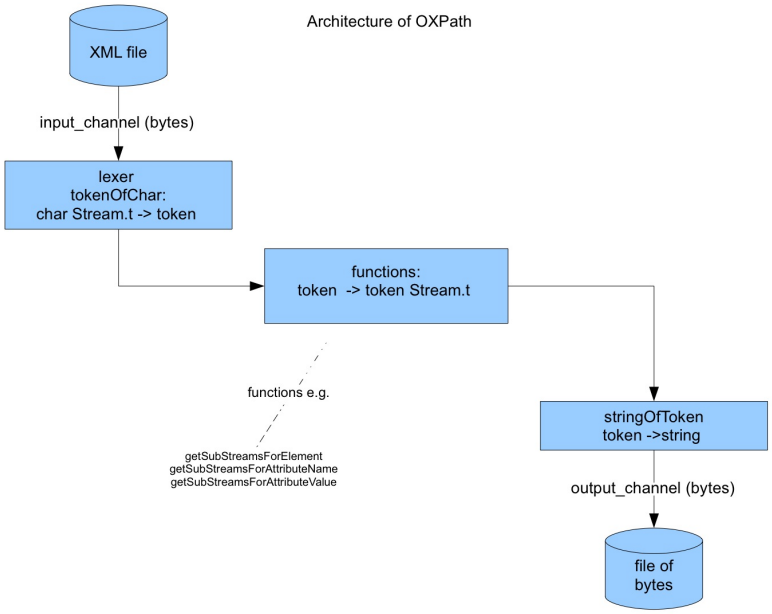

Figure 1: Architecture of `O-Search`.

`O-Search` will become our intermediate prototype for *stream-querying* of XML Data. It is implemented using the functional language Ocaml [2] [6]. We have currently implemented the basic search functions necessary for realizing XPath queries. To explain figure 1 we start in the input that is a simplified XML file which has the abstract syntax as in figure 2.

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

Figure 2: Abstract Syntax

An example of the concrete syntax for figure 2 is figure 3(a). Notice that 3(b) is its XML tree model. There are basically two types of nodes in the XML tree model:

- Element nodes: these correspond to tags in XML documents, and correspond to StartElement token in our concrete syntax, for example `StartElement("A",[])`. An attribute list is associated (optionally) with tags in the XML document , therefore it is associated with StartElement tokens in our concrete syntax, for example `StartElement("B",[("attr","2")])`. Note that, the attribute list is not nested(an attribute can not have any sub-elements), not repeatable (tow same-name attributes can not occur under one element) and unordered (attributes of an element can freely interchange their occurrences location under element). These constraints are standard to XML.

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

[2] Ocaml is a language of the ML family. It is well adapted to tree processing and its optimizing compiler ocamlopt produces fast executables

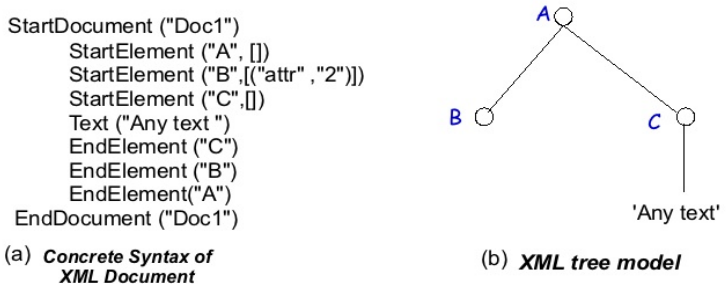

Figure 3: XML Tree Model.

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

```
val tokenOfCharStream:
char Stream.t -> token Stream.t = <fun>
```

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

```
val getSubStreamsForElement:
string ->in_channel -> string -> unit = <fun>
```

were the input arguments are:

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

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

```
val stringOfToken:
token -> string = <fun>
```

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

3 Experimental Results

3.1 Experimental Setup

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

3.1.1 Data sets

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

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

```
val generateWideTree :
string -> int -> in_channel = <fun>
```

   where:
   - *string*: is the output file name which will contain the wide tree data set.

- *int*: is the number of the shallow sub-trees in the wide tree data set. Queries will refer to each sub-tree's "token rank", see figure 4.

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

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

where:

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

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

Figure 4 is an example of wide tree data with the following size:

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

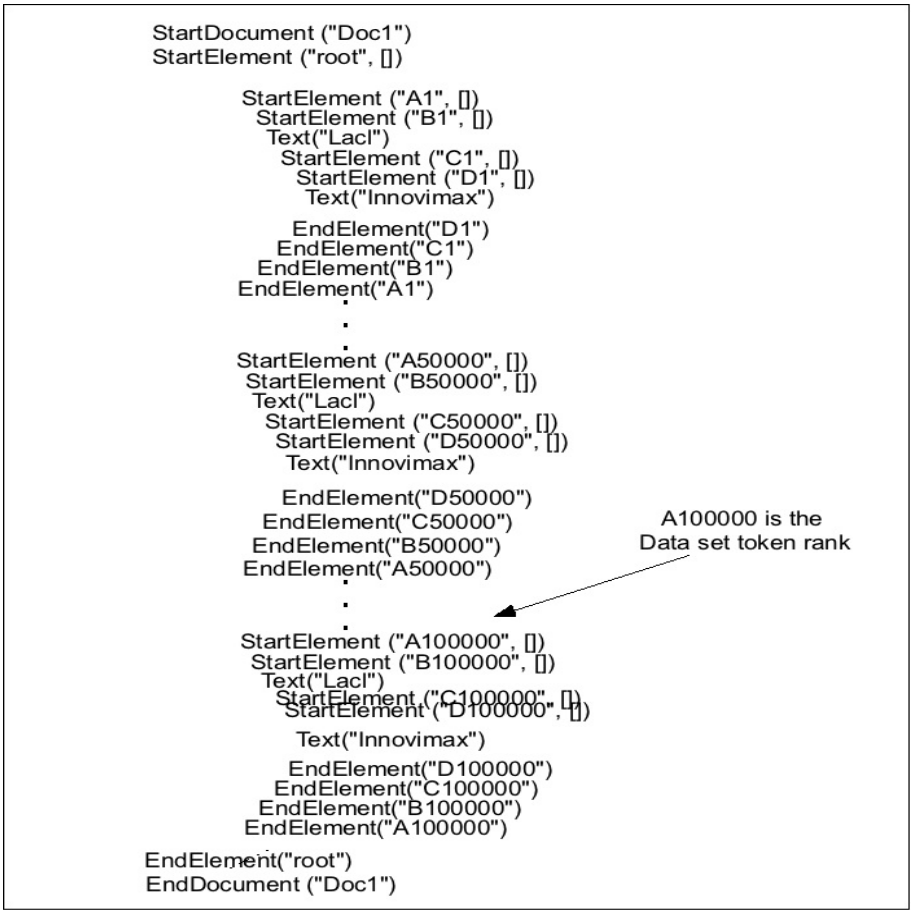

Figure 4: Wide tree data set

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

```
val generateDeepTree :
string -> int -> in_channel = <fun>
```

where:

- *string*: is the output file name which will contain the deep tree data set.

- *int*: is the loop depth of the deep tree data set. Queries will refer to "token rank" in this tree data set, see figure 5.

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

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

where:

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

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

Figure 5 is an example of deep tree data set with the following size:

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

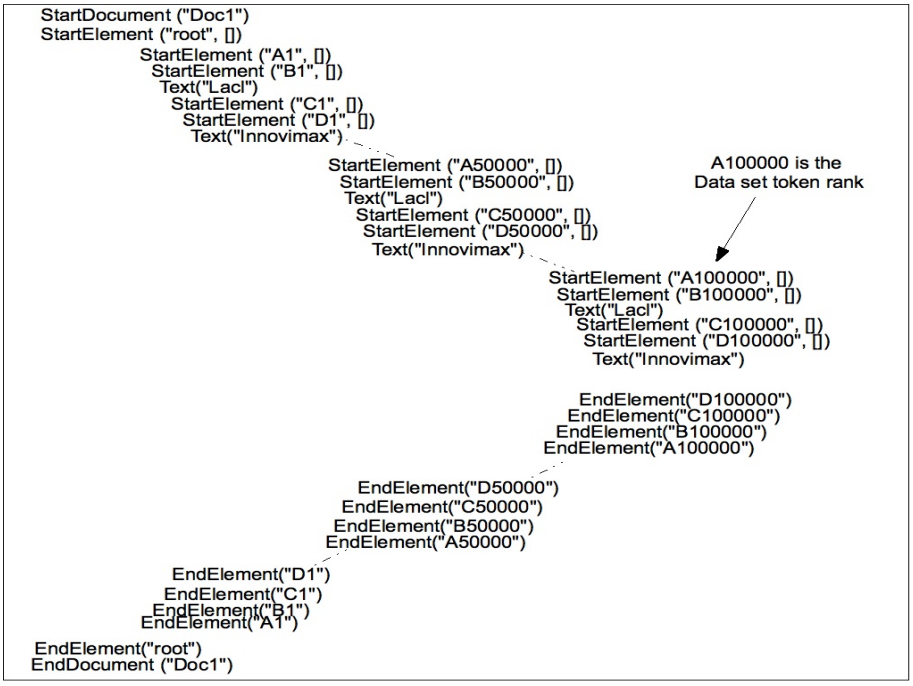

Figure 5: Deep tree data set

3.1.2 Test machine

Experiments were performed using a MacBook with the following technical specifications:

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

Table 1: Specifications of the test machine

3.1.3 Test measurements

- Time: to measure execution time, we use the following function `Sys.time();;`. This function exists in the module Sys[3] of Ocaml. It has a type:`unit -> float`, and it returns the processor time (in seconds) used by the program since the beginning of execution. To return the time of generating (for example) a wide tree data set, we use the following code:

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

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

```
let temp=ref (Sys.time());;
getSubStreamsForElement <token name>(generateDeepTree "inputFile.txt" <Data set token rank>) "outputFile.txt" ;;
print_float (Sys.time()-. !temp);
 print_string " Second\n"; temp:=Sys.time();;
```

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

```
val getMaxDepth:  unit -> int = <fun>
            getMaxDepth();;
```

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

3.1.4 Queries

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

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

where `Token name` can have the following values:

- `A1`: return the sub-tree of the element which exists at the beginning of the tree data set.
- `A<token rank/2>`: return the sub-tree of the element which exists in the middle of the tree data set.
- `A<Data set token rank>`: return the sub-tree of the element which exists at the end of the tree data set.

Furthermore, we use Positive and Negative queries, where:

- Positive: is a query that does not return an empty result.
- Negative: is a query that does return an empty result. We use negative queries with the two types of the tree data sets (Wide and Deep) as a reference for performance measurement.

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

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

[3]Sys: portable system calls.

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

Table 2: Wide tree data set - time and memory tests

3.1.5 Compiler

To test the execution time, we compile the CAML file using ocamlopt: the Objective Caml high-performance native-code compiler. Generated code is almost 10 times faster than generated code by ocamlc.
To test the memory consumption, we use ocamlc which compiles CAML source files to byte-code object files and links these object files to produce standalone bytecode executable files. These executable files are then run by the bytecode interpreter ocamlrun. For memory tests, it is recommended to use ocamlc because it is more accurate than ocamlopt [7].

3.2 Results

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

3.2.1 Wide tree data set (Positive queries)

We performed two tests using this data set, they are:

1. Time test
   To explain this test, we start in explaining table 2 that includes all the information needed:
   - **Query**:

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

   - **Test type**: table 2 contains three tests, they are y1, y2, and y3. In fact, we change the value of Token query rank with each test.
   - **Token name**: the token name we search.
   - **Data set token rank**: the rank of token of sub-tree, for better understanding see figure 4.
   - **Input tree size(tokens)**: the total number of tokens generated in our wide tree data set.
   - **T(total)**: the processor time in seconds since the beginning of the execution to generate the tree data set and answer the query.
   - **T(data set)**: the processor time in seconds since the beginning of the execution to generate the tree data set.
   - **T(total-data set)**: the processor time in seconds since the beginning of the execution to answer the query.
   - **Stack max. depth (token))**: is the maximum depth of the running time stack.

   Figure 6 represents three tests to measure the execution time in the wide tree data set. Test y1 uses a 1M token document, y2 a 5M token document and y3 a 10M token document. We noticed that test y1 is steady linear at 1,81 seconds irrespective of the data set token rank. Also, test y2 is almost steady linear at 9,3 seconds. While test y3 is almost steady linear at 18,7 seconds with a slight increasing particularly at the point (A1000000 - 1000000). Therefore, we conclude that execution time is independent of the data set token rank in the wide tree data set. We observe directly proportional to the input tree size: curves y2, y3 are multiples of y1 in this proportion.

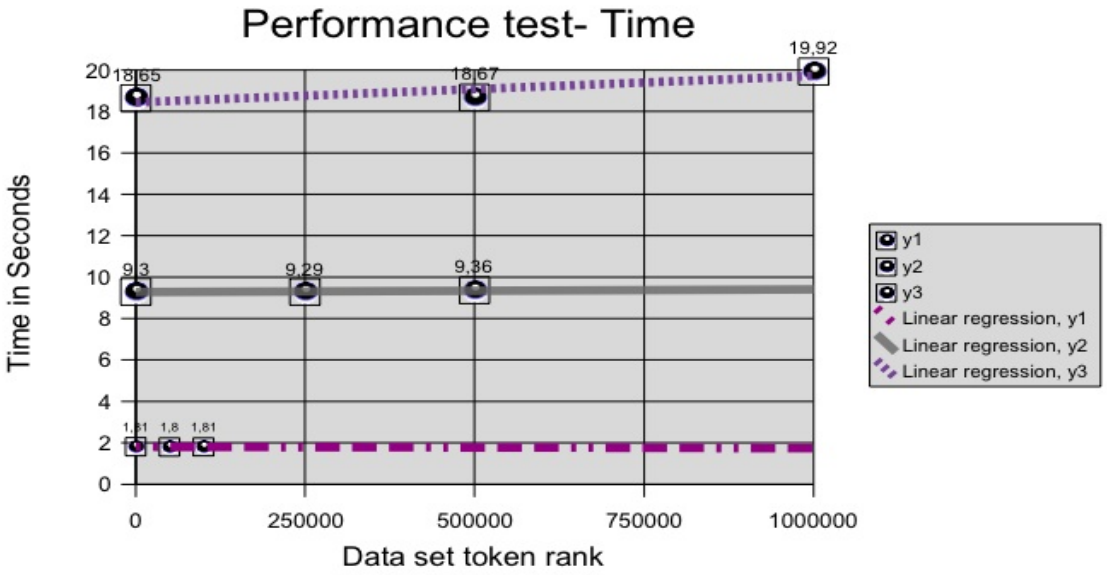

Figure 6: Wide tree data set - Time

2. Memory test

   To explain this test, we start in explaining table 2 that includes all the information needed:

   - **Query**:

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

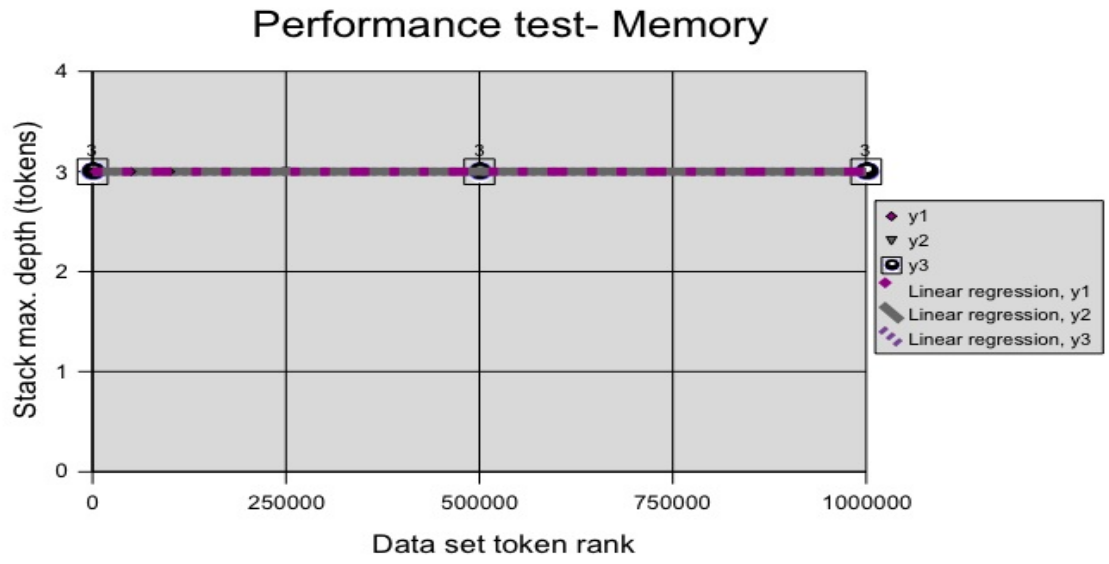

Figure 7: Wide tree data set - Memory.

Figure 7 represents three tests to measure the memory allocated to answer our query in the wide tree data set. Tests y1,y2 and y3 have the same value for the maximum number of tokens in the running stack which is 3 due to the symmetry of all sub-trees in the wide tree data set. Therefore, we conclude that the stack max. depth(tokens) is independent of the data set token rank in the wide tree data set.

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

Table 3: Deep tree data set - time and memory tests

3.2.2 Deep tree data set (Positive queries)

We performed two tests using this data set, they are:

1. Time
   The terms used in table 3 are the same as table 2.

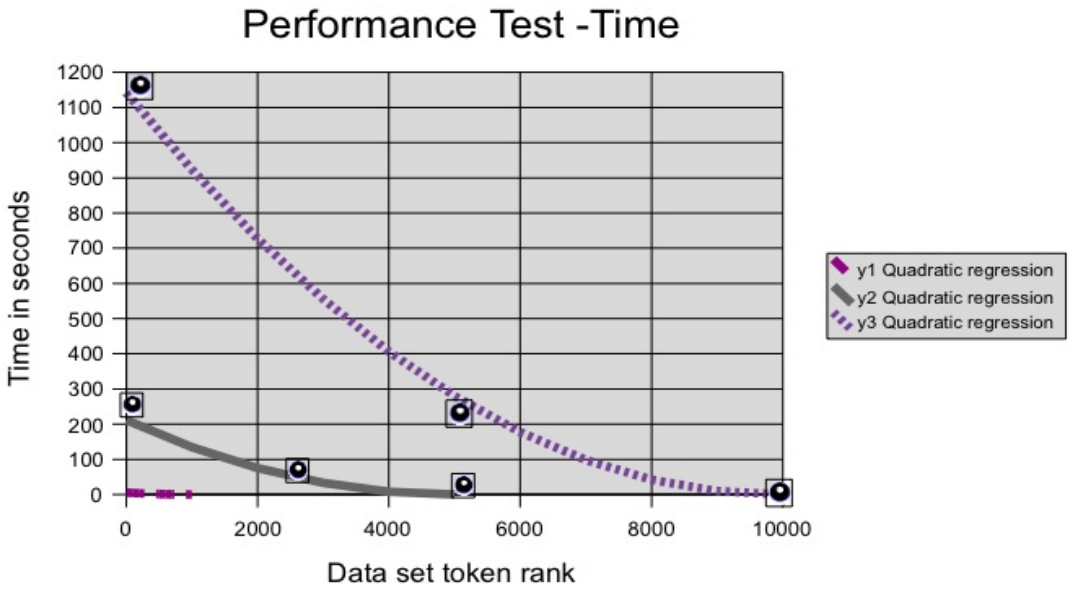

Figure 8: Deep tree data set - Time.

Figure 8 represents three tests to measure the execution time in the deep tree data set. We noticed that execution time increases with the increasing of the data set token rank and the decreasing of token name's value (see section 3.1.4) due to increasing the tokens those correspond the query which increase the execution time. More precisely, in test y1 the relationship between the execution time(y) and data set token rank (x) is $y = (2.2712 - 0.0022 * x)^2$. In test y2 the relation is: $y = (14.5450 - 0.0029 * x)^2$. While for the test y3 it is: $y = (33.8034 - 0.0034 * x)^2$. Therefore, we conclude that execution time is negative-quadratic proportional to the data set token rank in the deep tree data set. The time-rank relationship should normally be negative-linear and its quadratic behavior in our tests is due to a naive list implementation of one primitive. This will be changed in a future version and does not affect our conclusion in section 4.

2. Memory
   The terms used in table 3 are the same as table 2.

   Figure 9 represents three tests to measure the memory allocated to answer our query in the deep tree data set. We noticed that increasing the data set token rank and decreasing the value of query name will increase the value of stack max. depth(tokens). Furthermore, our evaluation technique is lazy [4], therefore we are obliged to buffer the whole subtree. In

[4]Lazy streaming algorithm evaluates the predicates only when the closing tags of the streaming ele-

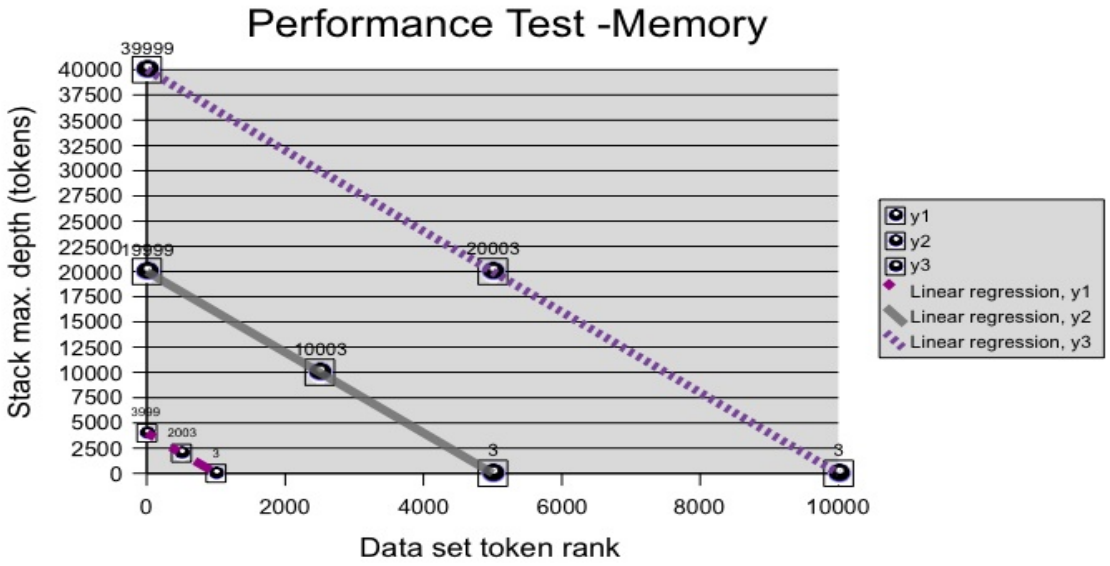

Figure 9: Deep tree data set - Memory.

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

Table 4: Wide tree data set - time and memory tests (negative queries)

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

3.2.3 Wide tree data set (Negative queries)

We performed two tests using this data set, they are:

1. Time
   Terms in table 4 have the same meaning as table 2.

   Figure 10 represents a test to measure the execution time in the wide tree data set using negative queries. We noticed that execution time increases with the increasing of the data set token rank due to the increasing of matching processes. More precisely, in test y1 the relationship between the execution time(y) and data set token rank (x) is as the following $y = 0,00001977 * x - 1.8196$. The importance of this test is to compare the measurements between wide tree data set and deep tree data set using negative queries.

ments are encountered.

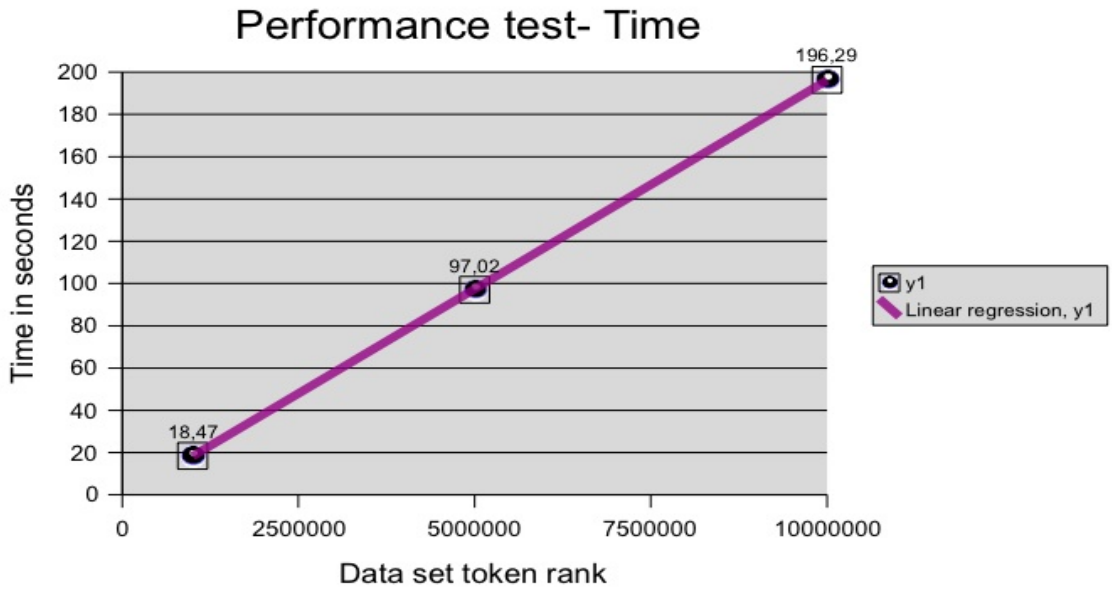

Figure 10: Wide tree data set - Time (negative queries).

2. Memory
   Terms in table 4 have the same meaning as table 2 Figure 11 represents a test to measure

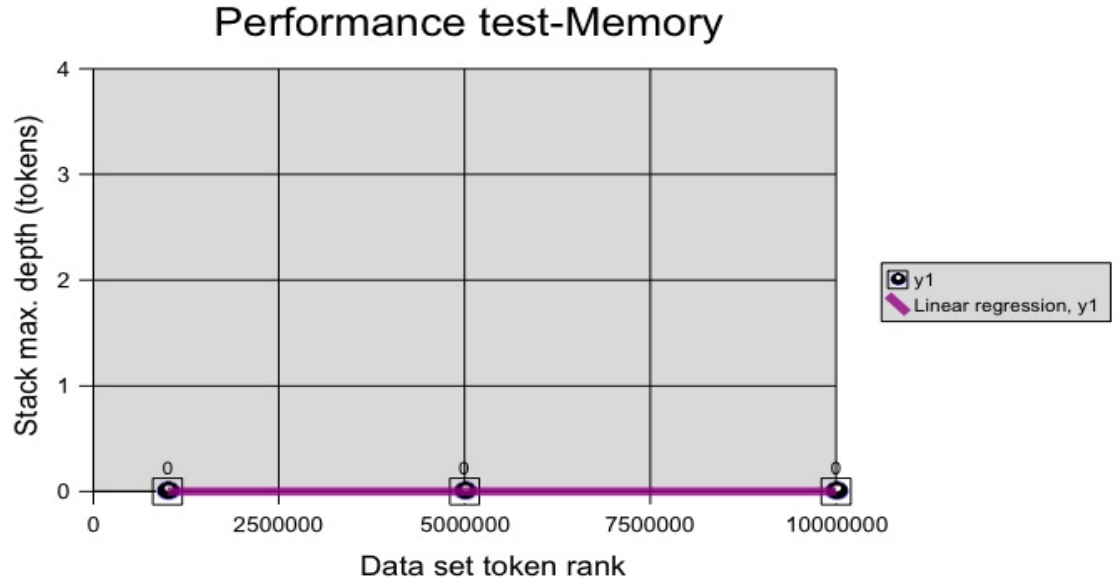

Figure 11: Wide tree data set - Memory (negative queries).

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

3.2.4 Deep tree data set (Negative queries)

We performed two tests using this data set, they are:

1. Time
   Terms in table 5 have the same meaning as table 2.
   Figure 12 represents a test to measure the execution time in the deep tree data set using negative queries. We noticed that execution time increases with the increasing of the data set token rank due to the increasing of matching processes. More precisely,

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

Table 5: Deep tree data set - time and memory tests (negative queries)

in test y1 the relationship between the execution time(y) and data set token rank (x) $y = 0.000019 * x - 1.55737704918$. Comparing the two linear equations to measure the execution time between both deep/Wide tree data sets using negative query indicates that deep tree data set has a better time performance than wide tree data set.

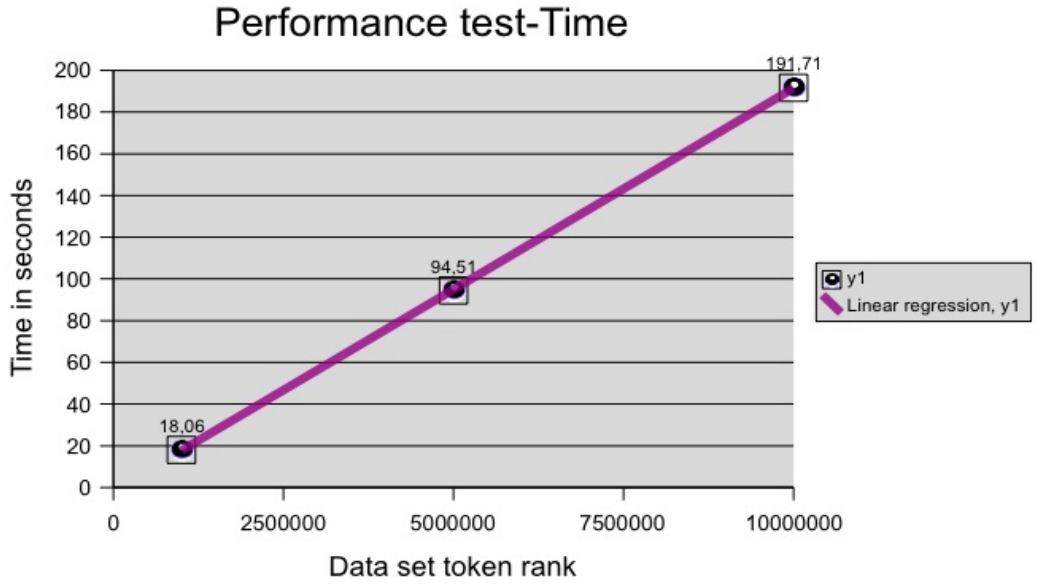

Figure 12: Deep tree data set - Time (negative queries).

2. Memory
   Terms in table 5 have the same meaning as table 2.
   Figure 13 represents a test to measure the maximum depth of the running stack (in tokens) to answer our negative query in the deep tree data set. We notice that the increasing the data set token rank does not affect the stack max. depth because our query is negative so we do not need to cache any element. We conclude that stack max. depth (tokens) is independent of the data set token rank in the deep tree data set.

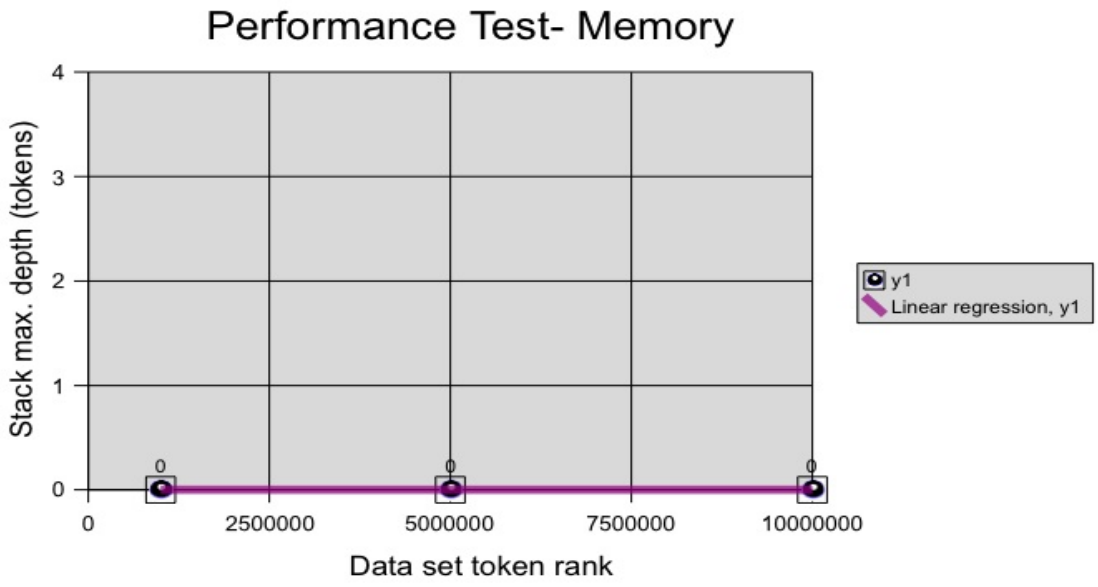

Figure 13: Deep tree data set - Memory (negative queries).

4 The oriented search proposal: evaluation of possible performance gains

4.1 Concept definition: a basic query example

This report supports our proposal for 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 as described below. The idea if 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 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.
In this case, the syntax of metadata could be as follows:

$$<A_4 \text{ order="2.2.1.1" subTreeSize="4"... /A>}$$

In this example, the attribute order provides us with the path from the root to nodeA_4 (Dewey Path), while the attribute subTreeSize gives the size of the subtree of the node A_4 (see figure 14).
To support Oriented XPath queries, the order information of nodes must be captured by the schema to process these queries efficiently. Figure 14 illustrates the information order of document D (ordered axes).

From A_1 to A_2 pure *stream-scanning* is applied to reach the specified point in document D that matches the query root. This process can save caching space as we will see in the example of section 4.2 The moment we reach the element A_4 we stop *stream-scanning* and start *stream-querying* for the whole sub-tree i.e. a complete querying algorithm is applied.

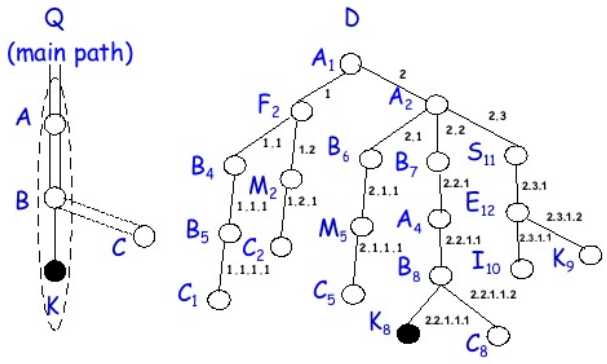

Figure 14: Document Information Order.

4.2 Performance gains

In this section we present two examples that explain the possible performance gains using Oriented XPath. ***Example 1:*** figure 14 represents an XPath expression $//A//B[.//C]/K$ and an XML document with information order (Dewey Path). Note that we use node numbering for example A_2 to explain the performance gain, in other words, the node that we do not need to cache or buffer. In our XPath expression 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. If XPath is provided with information as follows(start at the axis order 2 of node A) then, the syntax of XPath expression including the new meta data will be:

$$//A[@order = "2"]//B[.//C]/K.$$

in this case, based on figure 14 we do not need to cache the following nodes: B_4, B_5, also we do not need to buffer the node C_1 though they would otherwise match the sub-pattern of the query.

Example 2: this example shows the possible gains of time and memory using Oriented XPath on our synthetic data sets (wide/deep). We assume that metadata can either indicate one precise subtree or a subset of subtrees where to search.

1. Wide tree data set: figure 15 is a wide tree data set.
   Optimization for both time and memory has three scenarios:

   - Worst case: we do not have any meta data to orient our search. Therefore there will be no memory or time saving, see figure 15.
   - Average case: meta data orients our query to start searching somewhere after the middle of the tree data set, see figure 15. Stack consumption being already minimal, there is no expected saving from the use of meta data here. The saving in processing time will be proportional here to half of the document size and to the difference between the time to scan one token vs the time to process one token in query mode. This difference is not yet measurable in our tests, which is explained because it should increase with the complexity of query patterns. Our current tests only deal with elementary queries so scan time is currently almost equal to query processing time.
   - Best case: meta data orients our query to a precise sub-tree that answers the query 15. Stack consumption being already minimal, there is no expected saving from the use of meta data here. The saving in processing time will be proportional here to the document size multiplied by (time to scan one token - time to process one token in query mode).

2. Deep tree data set: figure 16 is a deep tree data set.
   Optimization for both time and memory has three scenarios:

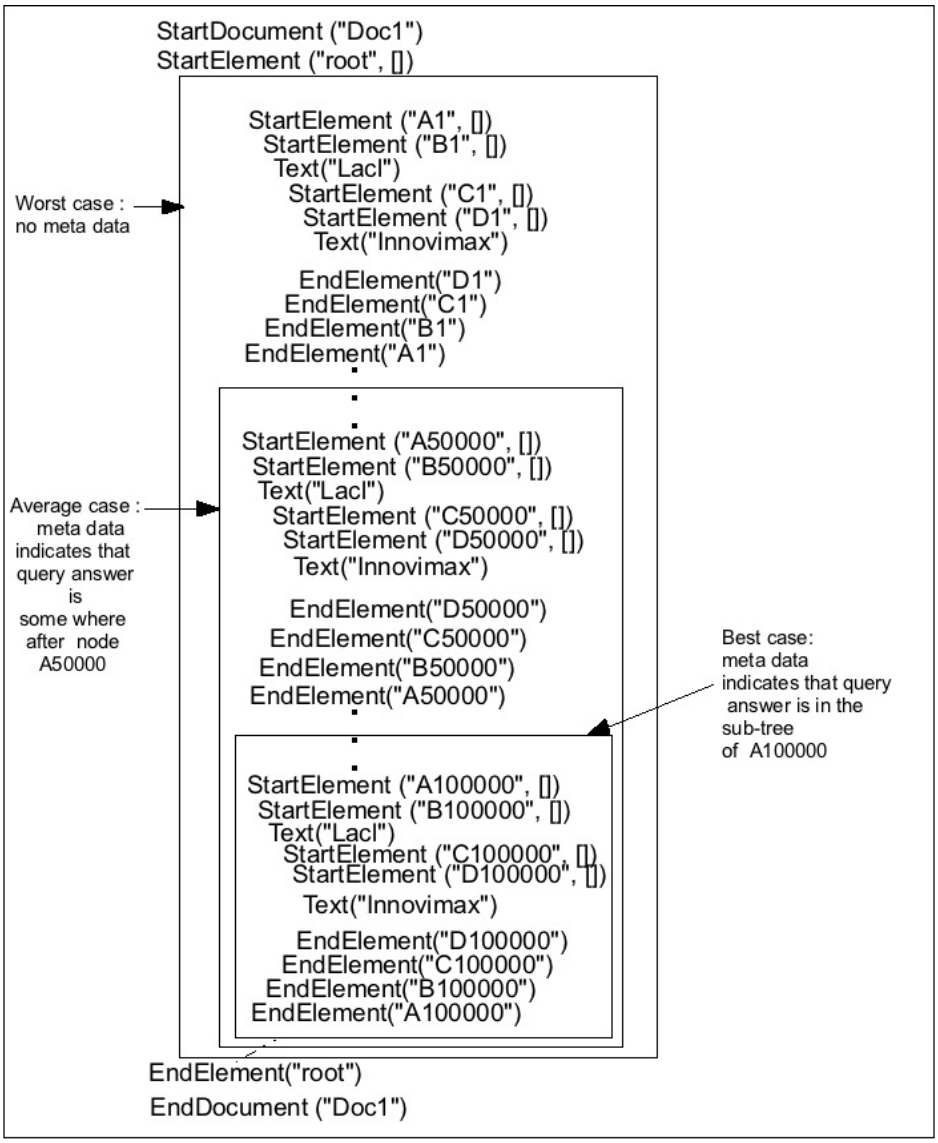

Figure 15: Meta data in the wide tree data set.

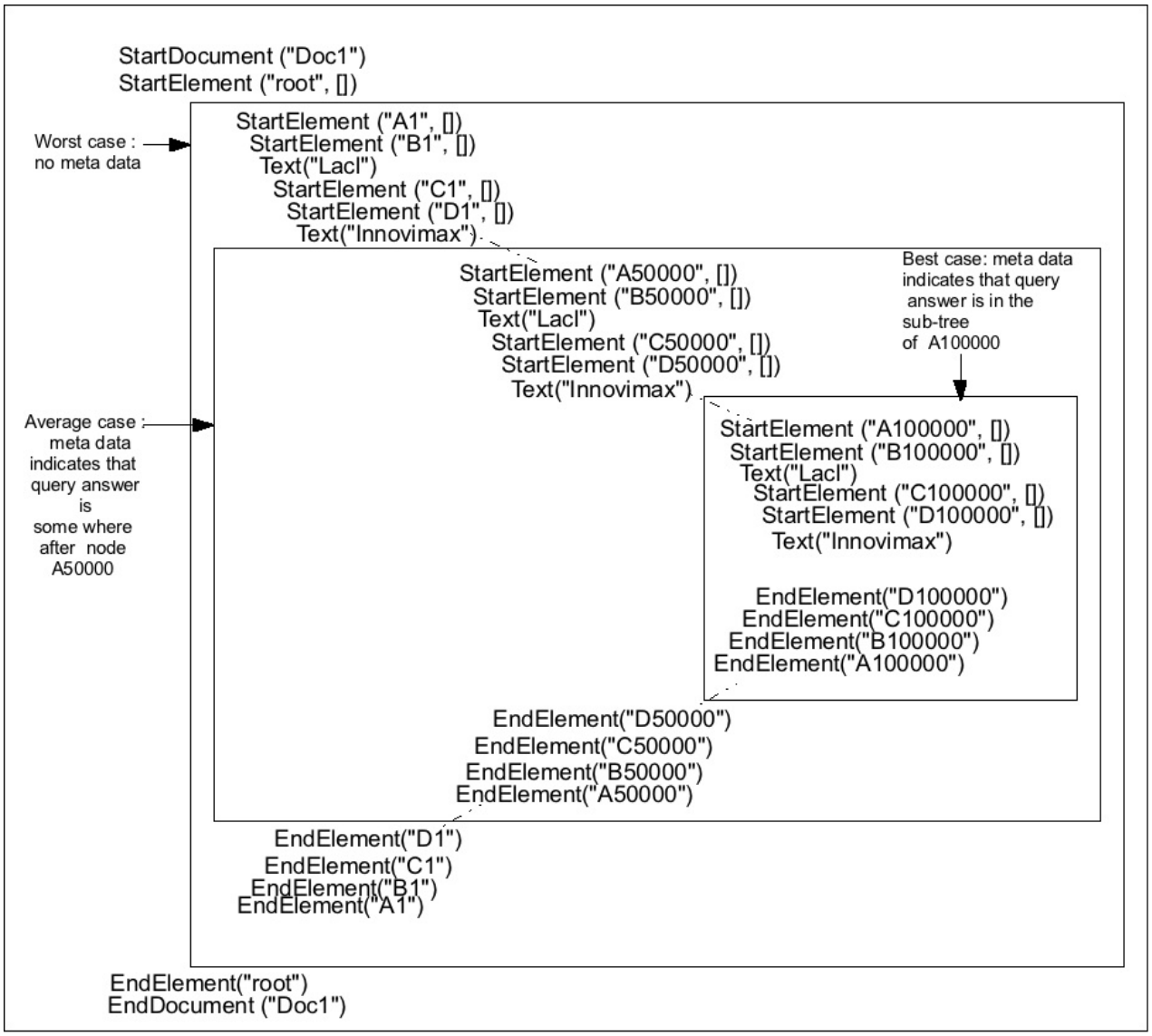

Figure 16: Meta data in the deep tree data set.

- Worst case: we do not have any meta data to orient our search. Therefore there will be no memory or time saving, see figure 16.
- Average case: meta data orients our query to start searching somewhere after the middle-depth of the tree data set, see figure 16. The saving in stack consumption is here of 50% and an example of saving in time can be found by comparing tests in Table 3. For example time could be reduced from 223s to 41s when processing a document of 50004 tokens.
- Best case: meta data orients our query to a precise sub-tree that answers the query 16. Stack consumption is then almost reduced to 0, almost a 100% saving. The possible time saving is here also visible in Table 3. For example time could be dramatically reduced from 223s to 0,08s when processing a document of 50004 tokens.

5 Conclusion

In this report we presented the first experimental results to support our proposal to optimize stream processing for XML data with the use of meta data to orient search. A core prototype called `O-Search` was implemented to have a concrete understanding for the complexity of stream-querying algorithms , with respect to different structures of XML documents (wide, depth, size). measurements have confirmed and quantified the space time complexity of query processing primitives. Meta data can improve performance parameters in average, can not avoid the worst case performance but may in favorable cases yield dramatic improvements.

Our concept remains to be evaluated on complex queries and a realistic framework for providing meta data must be constructed.
Even in the absence of any automated collection of meta data, there are scenarios for which manually-provided meta data is both realistic and necessary. For example, in situations involving portable devices with small memory sizes, explicit meta data provided by the user will allow searches to be performed even though a false negative result is possible. In this case the user would repeat his search. Our experiments and estimations have shown the gains possible in this manner, and future research could generalize its applicability to many other situations.

References

[1] M. Alrammal, G. Hains, and M. Zergaoui. Intelligent Ordered XPath for Processing Data Streams. *Accepted article in AAAI09-SSS09, Stanford -USA,* `http://icep-aaai09.fzi.de/accepted`, 2009.

[2] 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.

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

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

[5] 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.

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

[7] INRIA. Module Gc. http://caml.inria.fr/pub/docs/manual-ocaml/libref/Gc.html.

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

A complete bibliography of our state of the art can be found in [1].