A Stream-Based Selectivity Estimation Technique for Forward Xpath

muath alrammal and gaétan Hains

Université Paris Est, France

muath.alrammal@u-pec.fr , gaetan.hains@u-pec.fr

ABSTRAT

Extensible Markup Language (XML) rapidly establishes itself as the de facto standard for presenting, storing, and exchanging data on the Internet. However, querying large volume of XML data represents a bottleneck for several computationally intensive applications. A fast and accurate selectivity estimation mechanism is of practical importance because selectivity estimation plays a fundamental role in XML query performance. Recently proposed techniques are all based on some forms of structure synopses that could be time-consuming to build and not effective for summarizing complex structure relationships. To overcome this limitation, we propose an innovative selectivity estimation algorithm, which consists of (1) the path tree synopsis data structure, a succinct description of the original document with low computational overhead and high accuracy for processing tasks like selectivity estimation, (2) the streaming selectivity estimation algorithm which is efficient for path tree traversal. Extensive experiments on both real and synthetic datasets show that our technique achieves better accuracy and less construction time than existing approaches.

Keywors

XML data, XPath queries, query optimization, stream processing, selectivity estimation.

1. INTRODUCTIN

XML [6] 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 [4] and XQuery [5] have been proposed for accessing XML data. They provide a syntax for specifying which elements and attributes are sought to retrieve specific pieces of a document.

A stream of XML data is the depth-first, left-to-right traversal of an XML document [6]. Cost-based optimization of XML stream querying requires calculating the cost of XPath query operators. Usually the cost of an operator for a given XPath query depends heavily on the number of the final results returned by the query in question, and the number of temporary (intermediate) results that are buffered for its sub-queries [23]. Therefore, accurate selectivity estimation is necessary for cost-based optimization, but insufficient as we explain below.

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

As a result, the current selectivity estimation techniques appear necessary but incomplete for managing queries on large documents accessed as streams. We therefore propose a new stream-based selectivity estimation technique. We compute the path tree, a synopsis data structure from the input XML document *D*. The purpose is to obtain a small but full structure synopsis that is traversed by an efficient streaming algorithm to reduce the computational overhead of complex XPath queries on *D*.

The remainder of the paper is structured as follows: the next section is a short survey of existing work on synopses data structures and twigs selectivity estimation. In the third section, we present our motivations and contributions. The fourth section presents our stream-based selectivity estimation technique. In the fifth section we compare our technique with the existing ones, and the paper then concludes with an outline of future work.

2. RELATED WORK

Various research works in estimating the selectivity of XPath queries have been published. The majority [1] [14] [13] [22] [11] have focused on linear XPath queries (e.g. //A//B/C). It is not clear how these approaches can be extended to XPath twig queries (quereis with predicates e.g. //A[.//B]/C) so as to cover a larger fragment of XPath.

Several structure synopses, such as Correlated Suffix Trees [7], Twig-Xsketch [17], TreeSketch [16], and XSeed [24], have been proposed for twig query selectivity estimation. They generally store some form of compressed tree structures and simple statistics such as node counts, child node counts, etc. Due to the loss of information, selectivity estimation heavily relies on the statistical assumptions of independence and uniformity. Consequently, they can suffer from poor accuracy when these assumptions are not valid. The above proposed structures synopses can not be evaluated by ordinary query evaluation algorithms, they require specialized estimation algorithms.

The authors of [7] proposed a correlated sub-path tree (CST), which is a pruned suffix tree (PST) with set hashing signatures that helps determine the correlation between branching paths when estimating the selectivity of twig queries. The CST method is off-line, handles twig queries, and supports substring queries on the leaf values. The CST is usually large in size and has been outperformed by [1] for simple path expressions.

Described in [17] the Twig-Xsketch is a complex synopsis data structure based on XSketch synopsis [13] augmented with edge distribution information. It was shown in [17] that Twig-Xsketch yields estimates with significantly smaller errors than correlated sub-path tree (CST). For the dataset XMark [18] the ratio of er- ror for CST is 26% vs. 3% for Twig-Xsketch.

TreeSketch[16] is found on a partitioned representation of nodes of the input graph-structured XML database. It extends the capabilities of XSketch [13] and [17] Twig-Xsketch. It introduces a novel concept of count-stability (C-stability) which is a refinement of the previous F-stability of [13]. This refinement leads to a better performance in the compression of the input graph-structured XML database.

Paper [15] introduced XCLUSTER, which computes a synopsis for a given XML document by summarizing both the structure and the content of document. The XCLUSTER-based synopsis data structure is a node- and edge-labelled graph, where each node represents a sub-set of elements with the same tag, and an edge connects two nodes if an element of the source node is the parent of elements of the target node. Nodes and edges of this graph are then equipped with special aggregate statistical information.

Paper [24] proposed the XSeed synopsis to summarize the structural information of XML data. The information is stored in two structures, a kernel, which summarizes the uniform information, and an HET (Hyper-Edge Table), which records the irregular information. By treating the structural information in a multi-layer manner, the XSeed synopsis is simpler and more accurate than the TreeSketch synopsis. Moreover, XSeed supports recursion by recording "recursion levels" and "recursive path expression" in the synopses. However, although the construction of XSeed is generally faster than that of TreeSketch, it is still time-consuming for complex datasets.

Paper [12] proposed a sampling method named subtree sampling to built a representative sample of XML which preserves the tree structure and relationships of nodes. The number of data nodes for each tag name starting from the root level is examined. If it is sufficiently large, a desired fraction of data nodes are randomly selected using simple random sampling without replacement and the entire subtrees rooted at these selected data nodes are included as sampling units in the sample. If a tag has few data nodes at the level under study, then all the data nodes for that tag at the level are kept and they move down to check the next level in the tree. The path from the root to the selected subtrees are also included in the sample to preserve the relationships among the sample subtrees. Though a subtree sampling synopsis can be applied to aggregations functions such as SUM, AVG, etc., it is shown in [12] that XSeed [24] outperforms subtree sampling for queries with Parent/Child on simple dataset e.g. XMark [18], while it is the inverse for complex datasets.

3. MOTIVATIONS AND CONTRIBUTIONS

Having explored the state of the art, we summarize our motivations as follows:

- A 2005 study [20] of Yahoo's query logs revealed that 33% of the queries from the same user were repeated and that 87% of the time the user would click on the same result as earlier: repeat queries are used to revisit information [20]. This motivates our intense use of preprocessing: its cost can most often be amortized. Moreover it is possible to update our synopsis data structure by streamed and incremental updates.

- The proposed structures synopsis above (in section 2) can not be evaluated by ordinary query estimation structure, they require specialized estimation algorithms or rules.

- Though the construction time for structures synopsis vary, for example: the construction of XSeed is generally faster than that of TreeSketch as it is shown in [12]. The techniques used for synopsis construction are still time-consuming for complex datasets e.g. TreeBank [19].

- Most selectivity estimation techniques do not process the complete fragment of Forward XPath (the grammar of this frag ment is introduced in section 4.

Our contributions can be summarized as follows:

1. We present a new stream-based selectivity estimation technique. Where, we present the path tree, a synopsis structure for XML documents that is used for accurate selectivity estimates. We formally define it and we introduce a streaming algorithm to construct it. Furthermore, we introduce an efficient selectivity estimation algorithm for traversing the synopsis structure to calculate the estimates. The algorithm is well suited to be embedded in a cost-based optimizer.

2. Extensive experiments were performed. We considered the accuracy of the estimations, the types of queries and datasets that this synopsis can cover, the cost of the synopsis to be created, and the estimated vs measured memory allocated during query processing. Experiments demonstrated that out technique is both accurate and efficient.

4. STREAM-BASED SELECTIVITY ESTIMATION TECHNIQUE

The stream-based selectivity estimation technique consists of (1) the path tree structure synopsis: a concise, accurate, and convenient summary of the structure of the XML document, (2) the selectivity estimation algorithm: an efficient streaming algorithm used to traverse the path tree synopsis to provide the end user with different estimates which allow him to optimize his query if needed.

The current version of our selectivity technique processes queries which belong to the fragment of Forward XPath*: a sub fragment of XPath 1.0 consisting of queries that have: child, descendant axis. NodeTest which is either element, wildchard, 'text()'. Predicate with ('or','not', 'and') and arithmetic operations.*

For a precise understanding of Forward XPath, we illustrate its grammar in figure 1. A location path is a structural pattern com- posed 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 './/'. Where *//node()* is a direct child, *//node()* is a descendant, *[.//node()*] is a child predicate node for refinement, and *@node()* is an attribute;

```
Path := GenericPath
GenericPath := GenericStep | GenericStep GenericPath | GenericStep1
GenericStep := Axis NodeTest | Axis NodeTest '[' Predicate ']'
AttributeStep := '@' NodeTest  | '@' NodeTest '[' Predicate ']'
GenericStep1 := Axis NodeTest1
Axis := '/' | '//'
NodeTest := name | '*'
NodeTest1 := 'text()'
Predicate := PredicatePath | PredicatePath CompOp constant
                           | Predicate 'and' Predicate
                           | Predicate 'or' Predicate
                           | 'not(' Predicate ')'
CompOp := '=' | '!=' | '>' | '>=' | '<' | '<='
PredicatePath := '.' GenericPath | AttributeStep
```

Figure 1: Grammar of Forward XPath

Figure 2 illustrates our stream-based selectivity estimation technique. As shown in the figure, the path tree is built for the target XML document by using our streaming algorithm (explained in section 4.1.2). After that, the moment the end user sends an Xpath function estimator provides the end user with query's estimation by using the path tree and the selectivity estimation (explained in section 4.2).

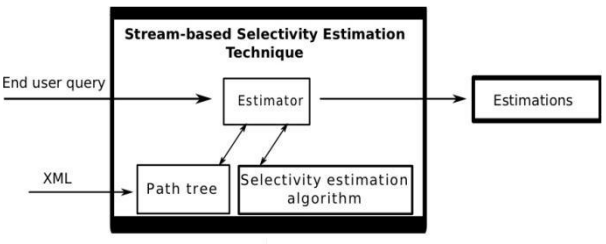

Figure 2: Stream-based selectivity estimation technique

Next, we will explain in details our technique.

4.1 Path tree

4.1.1 Path tree Definition

The path tree is a concise, accurate, and convenient summary of the structure of the XML dataset. It was invented by [1] but with a more restricted application than ours. To achieve conciseness, a path tree describes every distinct simple node-labelled path of a source XML exactly once with its frequency (the number of times it appears). To ensure accuracy, the path tree does not contain node-labelled paths that do not appear in the source XML dataset. The structure is convenient because it can be processed by ordinary query evaluation algorithms (stream-querying/stream-filtering algorithms) in place of the actual dataset.

Given an XML dataset *D*, the path tree is (a tree with node labels taken from *D*) defined as follows in figure 3. The details of path tree construction and updating are in [3]. However, we present below the pseudo code of the streaming algorithm for path tree construction with an example.

$\Sigma_{(D)}$ = is the finite set of nodes label of D.
$paths(D) = \{p = A_1, A_2, ...A_k \in \Sigma^*_{(D)} \mid p$ is a node-labelled path starting from the root of D *i.e.* A_1 is the root$\}$.
Remark: all node-label paths in $path(D)$ have A_1 as a prefix.

– **Definition:** we define $PathTree(D)$ as a graph whose nodes are $paths(D)$: $(path(D), \{(p_1, p_2) \mid \exists A \in \Sigma_{(D)}$ such that $p_2 = p_1 A$ *and* $p_i \in$ PathTree(D)$\})$.

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

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

Figure 3: Path tree definition

4.1.2 Path tree Construction

To create a path tree from an XML dataset *D*, we consider that *D* is equivalent to a DFA and its path tree is equal to a minimized DFA. Minimization can be done by creating the DFA completely then applying the automata minimizing algorithm [10]. Another possibility which is more memory efficient is to generate the minimized DFA directly. In this paper, we propose a *streaming* algorithm which takes as input the SAX parser events of *D* and creates directly its minimized automaton. We explain our algorithm through the example below.

The minimized automata is illustrated in figure 4 (autoTable). We start by explaining the structure of this table. *nName*: is the label of the node, where *nName* $\in \Sigma_{(D)}$. *depth*: is the node's depthin *D*. *nDown* and *nUp*: are counters for naming the states in the automata (e.g. 1, 2, ...etc.). Their initialized values = 0. Note that $\delta(nDown, nName) = nUp$. *nFreq*: is the frequency of nName in *D* which have the same node-labelled path. *nSize*: is the size in byte of *nName* in *D* which have the same node-labelled path. A stack named pathStack is used to store the node-labelled path during the construction process of the path tree. At each SAX event *StartElement(nName)*, pathStack is pushed with *(nName, nDown)*, and at each *EndElement(nName)*, the top of pathStack is popped out.

When < A > the root of *D* is read, *depth* = 1 then, we add *A* with its information to accessAutoTable, autoTable and pathStack (algorithm 1 lines 2 − 7). Note that *nUp* of *A* =*0*. When < B >

with *depth = 2* is read, the function *checkS ameNodePath* is called (algorithm 1 line 9). As long *B* is not yet a member of accessAutoTable (algorithm 2 line 1), then we add *B* with its information to accessAutoTable, autoTable and pathStack (algorithm 2 lines 21 − 27).

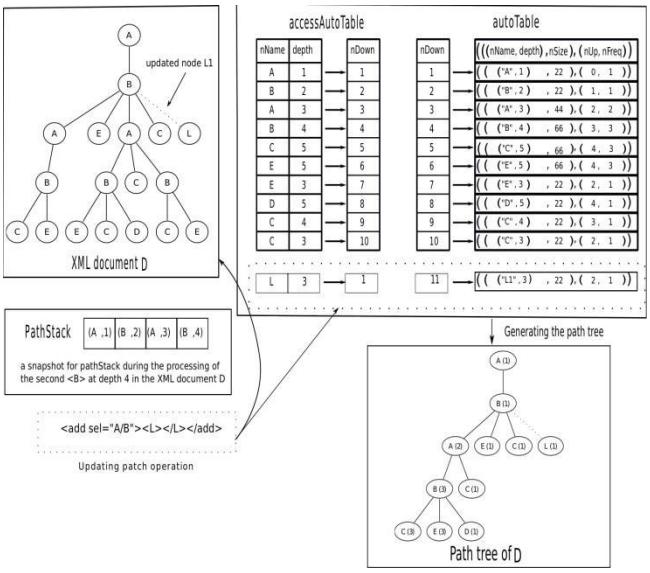

Figure 4: Path tree: Construction and Updating

The value of *nUp* for *B* with *depth = 4* (which is already exist in autoTable) is 3 (see algorithm 2 line 5 and autoTable). Also, in pathStack the value *nDown* for the parent (*depth−1*) of the received *B* is 3 (see algorithm 2 line 6 and pathStack), both values are equals because the parents of both *nName B* have the same node-labelled path, which mean both *nName B* also have the same node-labelled path. Therefore, we increment the frequency and size of *B* (see algorithm 2 lines 8 − 11). If the node-labelled path of *B* was not exist in autoTable (see algorithm 2 line 12), then node *B* with its information is added (see algorithm 2 lines 13 − 19).

Algorithm 1: createAutoTable (*depth*, *nName*, *nSize*)

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

When the second < B > with *depth = 4* is read, *B* is already a member of accessAutoTable (algorithm 2 line 1), therefore, we check whether the node-labelled path of the received *B* exists or not in autoTable (algorithm 2 lines 2 − 19).

The moment < /A > (EndElement of the root) is processed, the complete path tree can be generated and output in SAX events syntax.

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

Our streaming algorithm has time complexity $O(|depth(D)|.|D|)$ and space complexity $O(|depth(D)|.|pathTree(D)|)$.Where $|D|$ is thr XML dataset size

Algorithm 2: checkSameNodePath (*depth*, *nName*, *nSize*)

```
1  if (isMemeber accessAutoTable (depth, nName)) then
2      l← get the list of all nDown in accessAutoTable which have the same key
       (depth, nName)
3      let nodePathExist = false
4      foreach nDown ∈ l do
5          nodenUp= get nUp of nDown from autoTable
6          nodenDownPathStack=get nDown of (depth − 1) from pathStack
7          if (nodenUp= nodenDownPathStack) then
8              nodePathExist= true
9              augmentFrequency (nFreqStack) /* augment the nFreq of
               nName by 1 */
10             augmentSize (nSizeStack, nSize) /* augment the value in
               nSizeStack by nSize */
11             pushPathStack(depth, nName, nDown)        /* update the
               pathStack */
12     if (isNodePathExist = false) then
13         nDown ← (nDown) + 1
14         nDownPathStack= get nDown of (depth − 1) from pathStack
15         nFreqStack= [1] /* initialize the array with nFreq=1  */
16         nSizeStack= [nSize]     /* initialize the array with nSize
           (node Size) */
17         addNodeKey (depth, nName, nDown)      /* add a new node to
           accessAutoTable */
18         addNode (nDown, nName, depth, nSizeStack, nDownPathStack,
           nFreqStack)  /* add a new node to autoTable.  Note that
           nUp=nDownPathStack  */
19         pushPathStack (depth, nName, nDown)           /* update the
           pathStack */
20 else
21     nDown ← (nDown) + 1
22     nDownPathStack= get nDown of (depth − 1) from pathStack
23     nFreqStack= [1]     /* initialize the array with nFreq = 1  */
24     nSizeStack= [nSize]   /* initialize the array with nSize (node
       Size) */
25     addNodeKey (depth, nName, nDown)          /* add a new node to
       accessAutoTable */
26     addNode (nDown, nName, depth, nSizeStack, nDownPathStack,
       nFreqStack)       /* add a new node to autoTable.  Note that
       nUp = nDownPathStack */
27     pushPathStack (depth, nName, nDown) /* update the pathStack */
```

Path tree updating: when the underlying XML dataset is updated, *i.e.* some elements are added or deleted, the path tree can be incrementally updated using XML patch operations [21]. Due to the space limitation, we explain this procedure by a short example. Figure 4 shown an example of a patch operation to update the XML dataset *D*. This operation adds an empty element *L* as a last child under *"A/B"* where element *A* is the root of *D*. The same patch will be sent to the path tree (accessAutoTable and autoTable) for updating. Thus, we check whether the node-labelled path of *L* that is *ABL* exists or not in autoTable. In this example, it is not, therefore we add the new node *L* with its information to accessAutoTable and autoTable (see figure 4). Otherwise (node-labeled path of *L* is exist), the frequency and the size of node *L* will be updated as we shown in algorithm 2 (lines 7-11).

4.2 Selectivity Estimation Algorithm

To enable the selectivity estimation process, we inspired our selectivity estimation algorithm from LQ (the extended lazy stream- querying algorithm of Gou and Chirkova work [9]). Therefore, the advantages of this algorithm are the same as for the lazy stream-querying algorithm. Detailed explanations about LQ and its advantages are in [2].

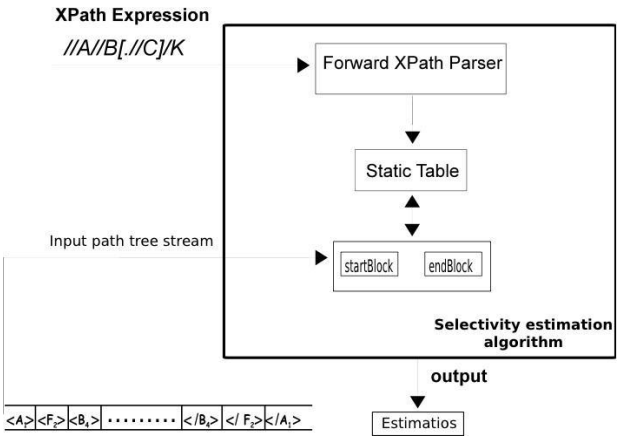

Figure 5: Selectivity estimation algorithm

Figure 5 illustrates our selectivity estimation algorithm. The current version of our estimation algorithm processes queries which belong to the fragment of Forward XPath. The estimation algorithm takes two input parameters. The first one is the XPath query that will be transformed to a query table statically using our Forward XPath Parser. After that, the main function is called. It reads the second parameter (the path tree) line by line repeatedly, each time generating a tag. Based on that tag a corresponding *startBlock* or *endBlock* function is called to process it. Finally, the main function generates as output the estimations needed for the given query.

Estimations are: *NumberOfMatches*: the number of answer ele- ments found during processing of the XPath query *Q* on the XML document *D*. *Cache*: the number of elements cached in the run-time stacks during processing of the XPath query Q on the XML document *D*. They correspond to the axis nodes of *Q*. *Buffer*: the number of potential answer elements buffered during processing of the XPath query *Q* on the XML document *D*. *OutputSize*: the total size in MiB of the number of answer elements found during processing of the XPath query *Q* on the XML document *D*. *WorkingSpace*: the total size in MiB for the number of elements cached in the run-time stacks and the number of potential answer elements buffered during processing of the XPath query *Q* on the XML document *D*. *NumberOfPredEvaluation*: the number of times the query's predicates are evaluated (their values are changed or passed from an element to another).

The algorithms 3, 4, 5 and 6 are the pseudo code of the stack automaton (functions *startBlock* and *endBlock*) of our selectivity estimation algorithm. Detailed explanation of our algorithm and several examples on selectivity estimation process can be found in [2]. However, the pseudo code and the selectivity estimation process are explained through the example below.

4.2.1 Example on the Selectivity Estimation

Figure 6 illustrates different snapshots of the evaluation process of the path tree of *D* on the twig path //A[.//C]/B[.//D]//E which returns *E1*(3), *E2*(1) as result nodes. For each non-leaf node, the algorithm creates a stack. Therefore, in this example, a stack is created for the root node *A* and another one for the node *B*.

Algorithm 3: startBlock (*nName*, *nFreq*, *nSize*, *depth*)

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

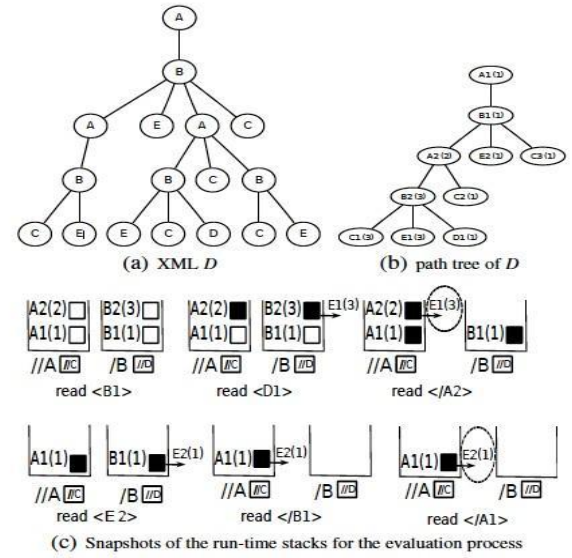

Figure 6: Snapshots of the run-time stacks for the evaluation of the path tree of D on Q (//A[.//C]/B[.//D]//E)

When < B2 > is read, the nodes *A1*(1), *B1*(1), and *A2*(1) were read and pushed (with their information) in their stacks. Concerning the node *B2*, it is also pushed (with its information) in

its stack *B*. Note that for each pushed node, the values of *Cache* and *WorkingSpace* are updated. (algorithm 3 lines 16-17).

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

```
1  if (node ≠ leaf) || (node's stack is empty) then
2      let s =get the top of the node's stack
3      if (node's depth = current depth) then
4          pop out the node
5          if (node's stack is not empty) then
6              check and update the predicates with descendant axis. If
               predicate node has a descendant axis, then increase predCounter
               by 1
7          let bool_Op_List= get the boolean operators associated with predicate
           children of the node
8          match (head bool_Op_List) with
9          | Not → if (the negation is true) then
10             processNodeType nNumber s                /*algorithm 5 */
11         else
12             appendOrDestroy nNumber s                /*algorithm 6*/
13         | And → if (all predicates are matched) then
14             processNodeType nNumber s     /*if the predicate does
               not contain a boolean operator, it will be
               processed as And.  */
15         else
16             appendOrDestroy nNumber s
17         | Or → if (one predicate is matched) then
18             processNodeType nNumber s
19         else
20             appendOrDestroy nNumber s
21         | Non → if (node has no predicate) then
22             processNodeType nNumber s
```

When < D1 > is read, the node *C1* was read, therefore, the value of the predicate C of *A2* was changed to true, and the value of *NumberOfPredEvaluation* was updated. The node *E1* was read, therefore, *E1* was buffered (with its information) to the potential answers list of its parent node *B2*, and the values of *Buffer* and *WorkingSpace* were updated (algorithm 3 lines 13-14). Moreover, by reading *D1*, the value of the predicate *D* of *B2* was changed to true and the value of *NumberOfPredEvaluation* was updated.

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

```
1  if (node type = Axis) then
2      if (node is the query's root) then
3          let potential_answer_list = the list of the potential answers nodes of
           the current node
4          if (potential_answers_list of the current node is not empty) then
5              calculate: NumberOfMatches, OutputSize
6              output the content of potential_answers_list: answers
7      else
8          if (potential_answers_list of the current node is not empty) then
9              append potential_answers_list to the same list of the parent of
               the current node
10 else
11     if (node type =Predicate) then
12         check and update the predicate and increase predCounter by 1
13         if (node axis = Descendant) then
14             clear the predicate's stack
15     else
16         if (node type =Result) then
17             if (node is the query's root) then
18                 calculate: NumberOfMatches, OutputSize
19                 output answers
20             else
21                 append node to the potential answers list of the node's
                   parent
```

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

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

When < /A2 > is read, the node *B2* was popped out from its stack, and the true value of its predicate *C* was passed to its ancestor *B1*, and the value of *NumberOfPredEvaluation* was updated (algorithm 4 line 6). Furthermore, the potential answers list of *B2* was appended to the same list of its parent node *A2* (algorithm 5 lines 8-9). Concerning *A2*, it is popped out of its stack, and as long as it is the root node, the content of its potential answers list is flushed as answers (algorithm 4 lines 13-14 then algorithm 5 lines 2-6).

When *E2* is read, it is buffered (with its information) to the potential answers list of its parent node *B2*, and the values of *Buffer* and *WorkingSpace* are updated (algorithm 3 lines 13-14).

When < /B1 > is read, it is popped out from its stack and its potential answers list is appended to the same list of its parent node *A1*. Finally, when < /A1 > is read, it is popped out from its stack, *A1* is the root node, therefore, the content of its potential answers list is flushed as answers (algorithm 4 lines 13-14 then algorithm 5 lines 2-6).

The result of the XPath query estimation is as follows (estimated values): *NumberOfMatches*: the value is 4, they are: *E1*(3), *E2*(1) = 3 + 1 = 4. *Buffer*: in this example, the value of *Buffer* is the same as *NumberOfMatches*. *Cache*: the value is 7, we present them based on their stacks as follows: stack *A* contains *A1*(1), *A2*(2), while stack *B* contains *B1*(1), *B2*(3). The value then 1 + 2 + 1 + 3 = 7. *WoringSpace*: its size was estimated to 0.0002MiB. *OutputSize*: its size was estimated to 0.00008MiB.

The estimated values equal the real measured ones which shows the accuracy of our selectivity estimation technique.

5. EXPERIMENTAL RESULTS

In this section, we demonstrate the accuracy of our technique by using variety of XML datasets and complex queries. Furthermore, we compare it with other approaches.

5.1 Experimental Setup

We performed experiments on a MacBook with the following technical specifications: Intel Core 2 Duo, 2.4 GHz, 4 GB RAM. The well known XML datasets XMark [18] and TreeBank [19] were selected for the experiments. XMark is a wide and shallow dataset, its size is 116MiB and its maximum depth is 12. TreeBank is a deep and recursive dataset, its size is 86MiB and its maximum depth is 36. The average relative error was used to measure the accuracy of our approach, it is defined as follows

$$\frac{1}{n}\sum_{i=1}^{n}\left|\frac{M_i - P_i}{M_i}\right|$$

where M_i is the measured value of the $i-th$ query in the workload and P_i is its predicted one.

Extensive testing and complex Forward XPath queries were used in our experiments. For example, a complex XPath query applied to XMark

//item[.//payment or .//shipping]//mailbox//mail[./date] and to TreeBank //EMPTY[./S//NP[./*] and .//VP]//*/NNS.

5.2 Accuracy of selectivity estimation technique

Figure 7 illustrates the accuracy of our stream-based selectivity estimation technique

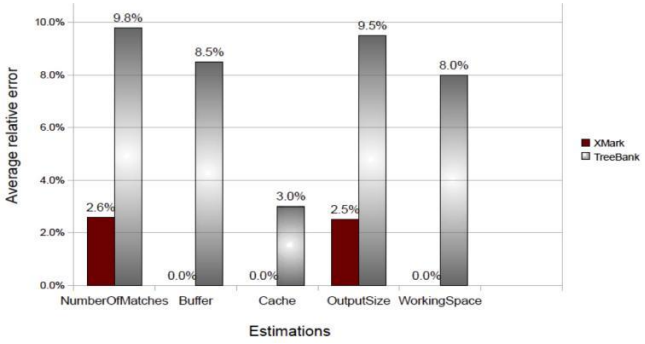

Figure 7: Accuracy of selectivity estimation technique

Our technique estimates the values of *NumberOfMatches*, *Cache*, *Buffer*, *OutputSize*, and *WorkingSpace*. While the different existing approaches estimate only the value of *NumberOfMatches*.
As shown in the figure, the accuracy of our technique on both datasets XMark and TreeBank is remarkable due to the complete structure information of the path tree which captures recursions in the dataset, and due to the efficiency of our selectivity estimation algorithm which supports the complete Forward XPath fragment. For example, in this figure, the max value of average relative error is for *NumberOfMatches* on TreeBank, it is less than 10%, so it is like informing the end user that the number of matches for an XPath query *Q* is 10 while in reality it is between 9 and 11.

5.3 Comparison with the existing techniques.

In this section, we compare our approach with other existing approaches.

5.3.1 Construction time for synopses

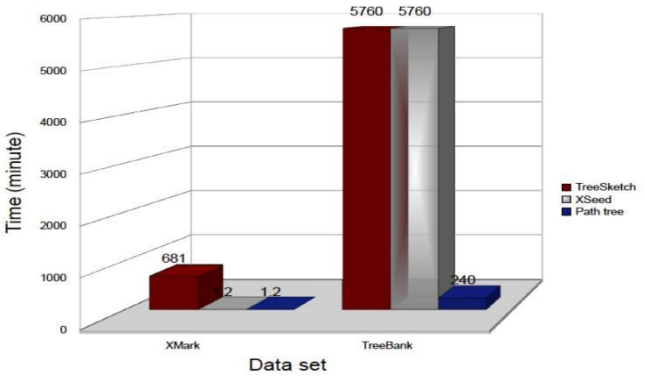

Figure 8: Construction time for synopses

Few approaches present the time needed to construct their synopses, like TreeSketch and XSeed. This is why in figure 8, we compare the construction time of our synopsis path tree with the same time needed for TreeSketch and Xseed.

Figure 8 shows the total construction time of TreeSktech, XSeed and path tree synopses. We do not show the construction time of the Subtree sampling synopsis because it is not a structural one (as we already explained in section 2), while for XCLUSTER and relational algebraic it is unknown.

The construction time of the structural synopses largely depends on the structure of the dataset. Our streaming algorithm for building path tree outperforms considerably the other approaches. The construction time for each of TreeSktech and XSeed for TreeBank 86MiB (depth 36) took more than 4 days (5760 minutes), this result was confirmed in [12]. While for path tree, the construction time for the same dataset took 244 minutes. Concerning XSeed, as the dataset become more complex, performance degrades dramatically [12] and construction time becomes significant. The construction time of path tree for TreeBank 86MiB (depth 36) is 24 times faster than XSeed (see figure 8).

5.3.2 Selectivity of structural queries: accuracy and synopsis size

TreeSketch and XSeed can estimate the accuracy for the number of matches (NumberOfMatches), while our approach estimates the accuracy for: *NumberOfMatches*, *Buffer*, *Cache*, and *OutputSize*, *WorkingSpace* . The accuracy of our approach outperforms the accuracy of TreeSketch and XSeed due to the complete structure of the path tree which captures the recursions in the dataset, and due to the efficiency our modified LQ algorithm which supports the complete Forward XPath fragment. The size of the path tree varies according to structure of the dataset. It is 10% of the size of TreeBank and 0.00006% of the size of XMark.

In all cases, an efficient streaming algorithm is used to traverse the path tree to avoid any computational overhead. Note that to control the space budget (synopsis size), it is possible to use a very partial, hence small path tree, to use no more space than competing approaches, but the accuracy of selectivity estimation will then be much lower.

The construction time for TreeSketch took more than 4 days. Actually we did stop the building process of its synopsis after 4 days. We faced the same situation for XSeed, but the difference between them that XSeed synopsis (as mentioned before) consists of two parts, an XSeed kernel and a hyper-edge table (HET). The kernel was built very fast, but the HET took more than 4 fays, this is why we did stop the construction process of HET. As long as, we could not build the synopsis of tree TreeSketch, and due to the fact that, the accuracy of XSeed outperforms the one of TreeSketch [24] [12], in figure 9 we compare our approach with the kernel of XSeed. We noticed that XSeed does not process queries with nested predicates and predicates with *'or'*, *'not'*, *'and'*, therefore, we refine and simplify the queries used in this experiment.

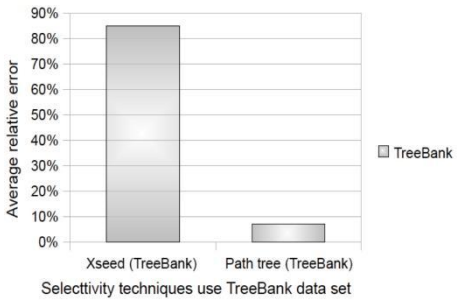

Figure 9: Accuracy of selectivity estimation techniques

As showed in figure 9, the average relative error of XSeed is almost 12 times higher than the same error for our approach.

5.3.3 Further Comparison

The fragment of XPath: the XPath fragment covered by our approach is more general than the one used by XSeed and TreeSketch. The TreeSketch query language does not support queries with *'text()'* [12] or with nested predicates. The XSeed query language does not support queries with *'text()'*, queries with nested predicates or queries with predicate which contain *'and', 'or', 'not'*.

Incremental update of synopsis: minimal synopsis size seems desirable but won't be the best because incremental maintenance would be difficult [8]. This is the case of TreeSketch and XSeed. While path tree preserves the same structure as the structure of its original XML dataset. So any language used to update the XML dataset can be used to update the path tree. Therefore, incremental update is possible, for example, by using the patch operations as we explained in section 4.1.2.

6.CONCLUSION AND PERSPECTIVES

In this paper, we presented our stream-based selectivity estimation technique. It uses the path tree synopsis and an efficient selectivity estimation algorithm to provide the end user with different estimations which allow him to optimize his queries. Extensive experiments were performed to evaluate our technique. We considered the accuracy of estimations, the types of queries and datasets that the selectivity estimation technique can cover, and the cost of the synopsis to be created. Experiments demonstrated that our tech nique is accurate and outperforms the existing approaches.

As an undergoing research, we study how to compute a synopsis for a given XML dataset by summarizing both the structure and the content of the dataset

6. ACKNOWLEDGEMENT

The authors thank M. Zergaoui president of Innovimax SARL for financial support in the form of a CIFRE scholarship for M. Alrammal, for suggesting the initial problem statement and participating in this work's supervision. Financial support from ANRT is also gratefully acknowledged

8. REFERENCES

[1] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In Proc. Of the 27th (VLDB), pages 591 – 600, 2001.

[2] M. Alrammal. Algorithms for XML Stream Processing: Massive Data, External Memory and Scalable Performance. Thesis, Université Paris-Est, 2011. *http://lacl.univ=paris12.fr/Rapports/TR/muth_thesis.pdf*.

[3] M. Alrammal, G. Hains, and M. Zergaoui. Path tree: Document Synopsis for XPath Query Selectivity Estimation. IEEE, In Proc. of the 5th (CISIS), pages 321–328, 2011.

[4] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) 2.0. 14 December 2010. *http://www.w3.org/TR/2010/REC-xpath20-20101214/*.

[5] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language (Second Edition). 14 December 2010. *http://www.w3.org/TR/2010/REC-xquery-20101214/*.

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and F. Yergeau. Extensible Markup Language (XML) 1.0 (fifth edition). 26 November 2008. *http://www.w3.org/TR/REC-xml/*.

[7] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. T. Ng, and D. Srivastava. Counting Twig Matches in a Tree. In Proc. of the 17th (ICDE), pages 595 – 604, 2001.

[8] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In Proc. of the 23rd (VLDB), pages 436–445, August 1997.

[9] G. Gou and R. Chirkova. Efficient Algorithms for Evaluating XPath over Streams. In Proc. of the 2007 ACM SIGMOD, pages 269–280, 2007.

[10] J. Hopcroft and J. Ullman. Introduction to Automata Theory, Language, and Computation. 1979.

[11] C. Luo, Z. Jiang, W.-C. Hou, F. Yan, and C.-F. Wang. Estimating XML Structural Join Size Quickly andEconomically. In Proc. of the 22nd (ICDE), 2006.

[12] C. Luo, Z. Jiang, W.-C. Hou, F. Yu, and Q. Zhu. A Sampling Approach for XML Query Selectivity Estimation. In Proc. of the (EDBT), pages 335–344, 2009.

[13] N. Polyzotis and M. Garofalakis. Statistical Synopses for Graph-structured XML Databases. In Proc. of the 2002 ACM SIGMOD, pages 358–369, 2002.

[14] N. Polyzotis and M. Garofalakis. Structure and Value Synopses for XML Data Graphs. In Proc. of the 28th (VLDB), pages 466–477, 2002.

[15] N. Polyzotis and M. N. Garofalakis. XCluster Synopses for Structured XML Content. In Proc. of (ICDE), 2006.

[16] N. Polyzotis, M. N. Garofalakis, and Y. Ioannidis.Approximate XML Query Answers. In Proc. of the 2004 ACM SIGMOD, pages 263–274, 2004.

[17] N. Polyzotis, M. N. Garofalakis, and Y. Ioannidis. Selectivity Estimation for XML Twigs. In Proc. of the (ICDE), 2004.

[18] A. Schmidt, R. Busse, M. Carey, M. K. D. Florescu, I. Manolescu, and F. Waas. Xmark: An XML Benchmark Project. Technical report, 2001. *http://www.xml-benchmark.org/*.

[19] D. Suciu. Treebank: XML Data Repository. Technical report, University of Pennsylvania Treebank Project, Novmber 1992. *http://www.cs.washington.edu/research/xmldatasets*.

[20] J. Teevan, E. Adar, R. Jones, and M. Potts. History Repeats Itself: Repeat Queries in Yahoo's Query Logs. In Proceedings of the 29th Annual ACM Conference on Research and Development in Information Retrieval (SIGIR), pages 703–704, 2005.

[21] J. Urpalainen. XML Patch Operations Framework Utilizing XPath Selectors. Network Working Group, 2008. *http://datatracker.ietf.org/doc/rfc5261/*.

[22] W. Wang, H. Jiang, H. Lu, and J. X.Yu. Containment Join Size Estimation: Models and Methods. In Proc. of the 2002 ACM SIGMOD, pages 145–156, 2003.

[23] N. Zhang, P. Haas, V. Josifovski, G. Lohman, and C. Zhang. Statistical Learning Techniques for Costing XML Queries. In Proc. of the 31st (VLDB), pages 289–300, 2005.

[24] N. Zhang, M. T. Ozsu, A. Aboulnaga, and I. F. Ilyas. XSeed: Accurate and Fast Cardinality Estimation for XPath Queries. In Proc. of the 20th (ICDE), 2006