Trading Precision for Throughput in XPath Processing

Engie Bashir
Department of Computer Science
American University of Beirut
Bliss Hall, P.O.Box 11-0236
Beirut, Lebanon
engie.bashir@aub.edu.lb

Jihad Boulos
Department of Computer Science
American University of Beirut
Bliss Hall, P.O.Box 11-0236
Beirut, Lebanon
jihad.boulos@aub.edu.lb

ABSTRACT

We present in this paper a system for rewriting user-specific XPath queries for higher sharing of common sub-expressions in a streaming environment. We rewrite these queries according to an extracted schema from a pre-processed stream and we apply the rewritten queries to a subsequent stream. We show that this rewriting yields a much higher throughput while keeping an error rate under control.

Keywords

XML, XQuery Processing, Precision, Throughput.

1. INTRODUCTION

Xamarkand [1] is a research prototype for XQuery processing; it incorporates a query rewriter for higher sharing among multiple subscribed queries, a query compiler, an optimizer, and a native XML storage manager. It also incorporates a query rewriting component that trades performance for precision where the query rewriting is accomplished according to a provided XML schema or an extracted schema from streaming documents. We present in this paper this query rewriting component where we concentrate on XPath since it is the foundation based on which XQuery is built. We are also more interested here with streaming documents in Publish/Subscribe environments since for stored documents a precise schema can be extracted without any difficulty and query rewriting presents no challenge.

Publish/Subscribe systems suffer from scalability issues. Different optimization approaches have been considered to resolve these issues and they have been proved quite successful. Most approaches have exploited commonalties in the prefix of XPath queries, and some others focused on query indexing. However, most Publish/Subscribe systems face performance and memory problems when handling a large number of queries over a fast stream of XML documents, especially if these queries include unspecified contents or structures (i.e., "$*$" and "//" in the XPath queries) to be matched on recursively defined elements. This imprecision in queries is mostly coming from the subscriber's lack of knowledge of the documents structure to be matched against and hence the non-existence of schemas for the streaming XML documents.

These trends have led us to exploit the possible existence of an implicit schema for streaming documents in order to rewrite some XPath queries. The rewriting mechanism would minimize the number of wildcards and/or ancestor-descendant relationships found in XPath queries. Therefore, we think our approach makes it possible for some unnecessary computations to be avoided.

As a concrete example, imagine an XML schema where among other things, the three elements "$<a>$", "$<b>$", and "$<c>$" are defined in a way such that "$<a>$" is a parent of "$<b>$" which is a parent of "$<c>$". Moreover, imagine the following three XPath queries:

- Q1: $/a//b/c/...$
- Q2: $//a/*/c/...$
- Q3: $/a/b/c/...$

Q1 checks whether every streaming element nested within "$<a> \cdots </a>$" is "$<b>$". However, the schema indicates that "$<b>$" is a direct child of "$<a>$"; hence, there is no need for the descendant checking. As for Q2, all children of "$<a>$" are possible parents of "$<c>$", while the schema indicates that "$<b>$" is the actual candidate. Thus both queries have a simple prefix structure "$a/b/c/...$" as Q3. A rewriting of some of these queries certainly provides an advantageous optimization in query processing. Moreover, a matching system that groups queries based on commonalities in their prefixes would be able to group these queries and many other variants of them into the same processing component. Based on these insights, we develop our approach and present an analysis of its benefits and drawbacks.

Our system is based on extracting an implicit schema from a stream of XML documents and use it to rewrite some of the subscribed queries. If a schema exists for the streaming XML documents, XPath query rewriting is straightforward and simple. If on the other hand no schema exists, a schema can be extracted from a sufficiently large number of streamed XML documents. The idea behind extracting a schema is to capture the structure of the streaming XML documents and track the arrival frequency of each XML element. We derive the schema by building an XML file in which attributes convey statistical data on elements frequency arrivals. These statistical values and the deduced relationships among elements contribute in the rewriting process of XPath expressions. We present the advantages of our approach by comparing the performance of the rewritten queries versus the original one.

Unfortunately, rewriting some XPath expressions according to an extracted XML schema of past-processed XML documents might introduce a certain error rate in missing some matches on subsequent streaming documents. This problem is the result of the non-existence of elements in past processed documents, and hence building non-perfect schemas. Hence, some rewritten queries might not match parts of some newly streaming documents. In our context, we rewrite a query only if the subscriber specifies that an error rate can be tolerated and what is that tolerance rate.

The rewriting mechanism is perceived as matching an XML schema against each XPath expression. After rewriting some queries by replacing some of the wildcards by their respective potential real element names, and some of the ancestor/descendent relationships by direct parent/child relationships or their respective paths from the built schema, we process the matching of the rewritten set of queries against the streaming XML data using an Extended Push-Down Automaton (E-PDA) [10]. We show that this filtering infrastructure is able to handle a much larger number of subscriptions and a higher number of documents per unit time without greatly affecting the precision of the matching process.

In the rest of this paper, we present in Section 2 a formalization of our problem. Section 3 presents our solution and some analysis; in Section 4 we show some performance and precision results of applying our proposed solution. We state some related works in Section 5 and conclude with some ideas for future works in Section 6.

2. PRELIMINARIES

We consider a subset of XPath expressions that contains element names, wildcards ("$*$"), children ("/") and ancestor-descendant relationships ("//"), and branches or predicates ("[...]"). However, non-predicate expressions are the ones considered in our query rewriting mechanism. Although our implementation can handle a larger grammar, the following grammar is the portion that mostly incorporates our target rewriting part of XPath.

$$E \quad \rightarrow \quad E/E \;\mid\; E//E \;\mid\; E[E] \;\mid\; \mathit{label} \;\mid\; * \qquad (1)$$

The problem this paper is concerned with is the processing optimization of XPath queries in XML filtering systems. We consider that in some environments, users are ready to sacrifice some precision for higher throughput and shorter response times. To achieve this goal we introduce the notion of query rewriting according to a derived schema. Hence, our optimization scheme is divided into three phases: schema extraction from a stream of XML documents, query rewriting of a set of XPath expressions, and streaming XML document matching against these rewritten XPath expressions.

2.1 XML Schema Extraction

The primary goal of this part is to derive a statistical XML schema from a collection of streaming XML documents. We build this schema in a way such that it is itself an XML document that preserves relationships, hierarchy, and most importantly the frequency of arrival of XML elements. This arrivals frequency is what makes the error rate in rewritten queries controllable. We can conceptualize this part as a learning phase in our solution approach.

Definition 1. Given a set of XML documents $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$, compute the XML schema $\mathcal{S}$ such that $\forall$ <e> ... </e> in $\mathcal{S}$ with frequency f, $\exists$ <e> ... </e > in f documents in $\mathcal{D}$ such that $children(e, \mathcal{D}) \subseteq children(e, \mathcal{S})$.

2.2 XPath Rewriting

The most visible component in our approach is to rewrite XPath expressions by minimizing the occurrences of wildcards and ancestor-descendant relationships. Thus, we focus on replacing location steps l_i that are of the forms "$/*$" and "$//a$". These replacements take place according to a derived schema and a user-provided error tolerance rate. Each replacement may introduce an error rate $\mathcal{E}(l_i)$.

Definition 2. Given an XPath expression $\mathcal{P}$=$l_1 \ldots l_f$, a tolerated error rate η, and a derived XML Schema $\mathcal{S}$, find the set (including a singleton) of XPath expressions $\{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n\}$ equivalent to $\mathcal{P}$ such that $\sum_{i=1}^{f} \mathcal{E}(l_i) \leq \eta$.

Example: Given an XPath expression $\mathcal{P} = /a//b[//f]/*/d$ and an XML Schema $\mathcal{S}$ where <b> is a direct child of <a> and $<c>$ is the child of $<b>$ having $<d>$ as its direct child, the equivalent XPath expression $\mathcal{P}_1 = /a/b[//f]/c/d$ is generated.

2.2.1 *Wildcard Rewriting*

The wildcard location step ("$/*$") may possess a relatively high processing cost compared to a normal element location step. If we know from the extracted schema what are the elements for this wildcard, we can replace it with those elements.

Example: In Q2 above, the "$/*$" transition is triggered on the arrival of each child of $<a>$ and there might be several of them. However, the query should match on a single element, implying that the computations incurred are useless. Thus, replacing the "$/*$" by "$/b$" would certainly alleviate the processing cost of such a match.

2.2.2 *Ancestor-Descendant Rewriting*

The ancestor-descendant relationship can be rather expensive to evaluate since it requires accessing all descendants of a context node.

Our solution approach is based on reducing the number of occurrences of ancestor-descendant relationships ("//") whenever possible in some XPath expressions. That is to say, the "$/a//b/c$" in Q1 above might change to "$/a/b/c$" according to the extracted XML schema and the tolerated error rate. This gain in performance takes place at the eventual expense of loosing some precision in the results.

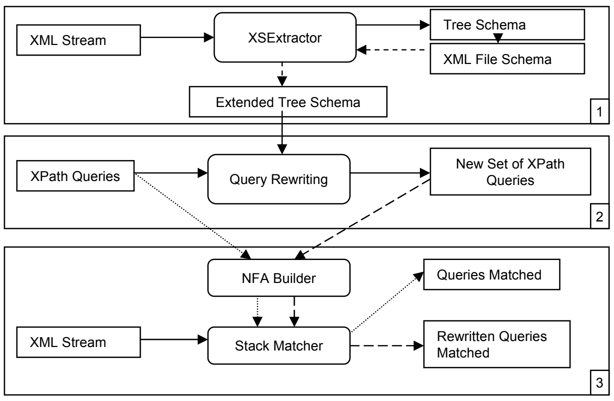

Figure 1: System Architecture.

2.3 Efficient Filtering of XML Documents

Definition 3. Given a set of XPath expressions $\mathcal{P}$=$\{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_m\}$ and a stream of XML documents $\mathcal{D}$=$\{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$. Compute for each document $\mathcal{D}_i \in \mathcal{D}$, the set of XPath expressions that match $\mathcal{D}_i$.

This problem has been addressed by several researchers [2, 5, 8, 9, 11]. It matches a set of XML documents against a set of XPath queries. We do not explain here how this is performed in our system because of space and since this is a known processing mechanism.

3. QUERY REWRITING

We introduce here our approach to rewrite a set of XPath expressions in a streaming environment. We start by providing an overview of our system's architecture; we then give a detailed description of the two major components of our system: the XML Schema Extraction (XSExtract) and the XPath Rewriter. A brief description of the query matcher is also provided.

3.1 System Architecture

Our system is divided into three major components that cooperate to produce a higher throughput while keeping a target error rate under control. The first component is called once for a set of streaming XML documents; it extracts an approximate schema for this set of documents with some other statistics. The second component is called to rewrite the queries in a set of XPath queries; this process is done according to the schema extracted in the first phase. The third component builds a Extended Push Down Automaton (E-PDA) for the rewritten set of queries. These three components constitute the compile-time part of our system; the run-time part—called the Stack Matcher—is executed once for every newly streaming XML document.

A short explanation here of the Query Rewriter may give a better understanding of how the Schema Extractor should be designed. The Query Rewriter rewrites a set of XPath queries, where each query is provided with the user's requirement of the result's accuracy, into another set where some navigation forms have been replaced by simpler ones. It consumes the set of original XPath queries along the Extended Tree Schema—that is derived from the built schema—to produce a new set of XPath queries. The new set of XPath queries are less complex than the original ones since some of the parent-descendant ("//") and the wildcard steps ("/*") are replaced by simpler navigational ones.

The architecture and functionalities of these components are discussed in more details in the following sections. A general overview of these components is given in Figure 2.

3.2 XSExtractor

XSExtractor takes charge of extracting a schema from a set of streaming XML documents. The derived schema reflects the hierarchy and frequency of arrivals of each XML element; it also preserves relationships among these elements. This schema has three models of representation: a Tree Model (TM), an XML File Model (XML-FM), and an Extended Tree Model (ETM). These models are semantically equivalent, but have different structures that facilitate different goals.

3.2.1 Tree Model

The TM is a schema model intended to summarize the structure and content of streaming XML documents. It also tracks the frequency of arrivals of each XML element.

Definition 4. The TM is a tree $T(V, E)$ where each node $v_i \in V$ corresponds to an XML element and $edge(v_i, v_j) \in E$ represents the parent-child relationship between v_i and v_j. Every node $v_j \in V$ has the following attributes:

- $element(v_j)$: the XML element name. In the following we use v_j and $element(v_j)$ interchangeably.
- $count(v_j)$: the frequency of arrival of the element.
- $level(v_j)$: the depth of an element where $level(v_j) = level(v_i) + 1$.

TM extraction is driven by the events of the SAX handler on a set of XML Streams (Algorithm 1 in Appendix A).

3.2.2 XML File Model

The XML-FM is another schema model for XML documents. To build an XML-FM, a TM is in fact itself transformed into an XML document to represent the extracted schema. The XML-FM is derived from the TM according to the following mapping. Let $v_i = Root(T)$, then:

- transform element (v_i) to its corresponding "start" and "end" elements as denoted by <v_i> ... </v_i>,
- transform $count(v_i)$ to an attribute $count = count(v_i)$ to be added to the start element,
- repeat the above steps for the set of direct children $C = \{v_j \in V \mid \exists\, edge(v_i, v_j) \in E\}$ and recursively nest the results accordingly.

3.2.3 *Extended Tree Model*

This model is derived from the XML-FM and is intended to be consumed by the query rewriter. It is an extended representation of the TM where every node in the tree holds additional information about its descendants. Some of this information is available only for descendants whose relative levels to the context node are within a specific range and whose upper bound is a system constant. Specifically, let $m = max$ be a constant that determines which ancestor-descendant relationships are subject for rewriting.

Definition 5. ETM is a tree $T'(V', E')$ where each node $v'_i \in V'$ corresponds to an XML element and $edge(v'_i, v'_j) \in E'$ represents the parent-child relationship between v'_i and v'_j in the XML schema file. Every node $v'_j \in V'$ has the following attributes:

- $element(v'_j)$: the XML element name. Again, we use in the following v'_j and $element(v'_j)$ interchangeably.
- $count(v'_j)$: the frequency of that element arrival.
- $level(v'_j)$: the depth of element v'_j where $level(v'_j) = level(v'_i) + 1$.
- $descendants(v'_j)$: the set of descendants $D_j = \{d_{j_1}, d_{j_2}, \ldots, d_{j_n}\}$ of v'_j such that $(\forall\, d_{j_h} \in D_j, 1 \leq h \leq n \Leftrightarrow \exists\, v'_k \in V' \mid element(v'_k) = d_{j_h})$ and the following two frequencies are collected:
  - $count(d_{j_h}, l)$: the arrivals frequency of descendant d_{j_h} at relative levels $l = level(v'_k) - level(v'_j)$ where $1 \leq l \leq max$.
  - $count(d_{j_h})$: the arrivals frequency of descendant d_{j_h} at all levels where $level(v'_k) > level(v'_j)$.

Given that above attributes, we define the following terms:

- let $P_{v'_k, l} = Pr(v'_k \mid v'_j, l)$ denote the probability of an XML element v'_k at level l given that it is a descendant of element v'_j where $l \leq m$, $(m = max)$.
- let $P_{v'_k} = Pr(v'_k \mid v'_j)$ denote the probability of v'_k at all levels given that it is a descendant of v'_j, i.e., $P_{v'_k} = \Sigma_{l=1}^{m} Pr(v'_k \mid v'_j, l)$.

ETM extraction is driven by the events of the SAX handler on the XML-FM schema file model. The algorithm of ETM extraction in not provided here due to space limitation.

3.3 Query Rewriter

This component, as its name indicates, takes a set of XPath expressions $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n\}$, each accompanied with a tolerated error rate η_i, and rewrites them in simpler forms without violating η_i for each $\mathcal{P}_i$. The rewriting is accomplished through matching each query separately against an ETM schema and according to a set of rules.

The system translates each XPath expression into a Non-Deterministic Push Down Automaton (PDA). This PDA definition is modified to adapt it to XPath queries and the ETM schema. The resulting mechanism is called an ExPush machine.

3.3.1 *Rewriting Rules*

The idea behind rewriting is to optimize the processing of these XPath queries by minimizing the occurrences of ancestor-descendant ("//") and wildcard ("*") steps, knowing that both types of steps are key factors in XPath evaluation complexity. Hence, the following set of rules for rewriting are introduced.

- whenever possible, replace a parent-descendant relationship in an XPath expression $\mathcal{P}_i$ (e.g., "$a//b$") by a set of wildcard location steps (e.g., "$a/*/\ldots/*/b$") for each <b> found to be a descendant of <a> in the derived schema and such that $level(b) - level(a) \leq m$ and $\Sigma_{i=1}^{m} 1 - Pr(b \mid a, i) \leq \eta_i$.
- whenever possible, replace a wildcard step ("/*") with a location steps (e.g., "/a") where <a> is found to be the proper replacement of "*" in the derived schema.

3.3.2 *ExPush Machine*

The ExPush machine is a modified PDA. The purpose of an ExPush machine is to rewrite an XPath expression, $\mathcal{P}_i = l_0 l_1 \cdots l_n$ where each location step $l_j = /e_j \mid //e_j$, while it is being matched against an ETM schema. When it exhausts the input of the schema, the ExPush machine returns a set of equivalent XPath expressions $\mathcal{P}_i = \{ep_1, ep_2, ..., ep_k\}$. The main change from a normal PDA is that it has multiple, extended, and linked stacks to operate on instead of just one. These chained stacks (ExStacks), each with its extensions (i.e., additional parameters), hold information on the rewritten queries. The second change is that the ExPush Machine accepts as input the nodes of the ETM. The ETM is traversed in a Depth First Traversal manner to reflect its XML schema view.

Since the ExPush machine has a PDA with a chain of extended stacks, we simplify its definition to combine the definitions of a PDA machine and an ExStacks machine.

3.3.3 *PDA Machine*

This machine models an XPath query $\mathcal{P}_j$ as a linear (i.e., non-branching) Finite State Machine (NFA). Unlike the NFA represented by other systems [2], our NFA hides some states and transitions related to the branching paths. The reason behind this elimination is that our system does not consider rewriting for branched paths. These XPath branches are stored within their parent's state for later expansion. Hence, each state in our system can have at most two outgoing transitions. An additional stack is joined to this NFA in order to keep track of which steps have been rewritten.

3.3.4 *ExStacks Machine*

The ExStacks Machine consists of a set of chained and extended stacks where each chain holds an XPath expression that could belong to the set of rewritten queries. All these chains are established while matching the ETM against the PDA. A chain construction is based on a *division/split* approach that is due to the occurrence of the consecutive states $\{q_j, q_{j+1},\, q_{j+2}\} \in Q \mid \delta(q_j, \epsilon) = q_{j+1}, \delta(q_{j+1}, e_j) = q_{j+2}$ in the PDA or equivalently to the occurrence of $e_{j-1}//e_j$ in the XPath expression. It is also related to the distribution of the $e_j \in descendants(v'_i)$ where $v'_i = e_{j-1} \mid v'_i \in V'$.

Thus, when these criteria are met a *division/split* occurs whenever a node $v'_k = e_j \mid v'_k \in V'$ in the ETM is reached. Accordingly, a new extended stack is added to the chain.

Definition 6. An ExStacks Machine is an $8-tuple$ $M = (S, K, \Gamma, \Delta_{push}, \Delta_{pop}, \Delta_{traverse}, \Delta_{top}, s_0)$ where

- s_0: initial stack.
- S: a set of extended stacks $\{s_0, s_1, \ldots, s_m\}$; initially $S = \{s_0\}$.
- K: a set of states $Q \cup \{q_{dummy}\}$.
- Γ: a set of XML elements $v'_i \mid v'_i \in V'$.
- Δ_{push}: the transition relation representing a valid and finite subset of $((S \times \Gamma \times K), (S \times K))$.
- Δ_{pop}: another transition relation representing a valid and finite subset of $((S \times \Gamma \times K), (S \times K))$.
- $\Delta_{traverse}$: a function $S \rightarrow \{K, K, \ldots, K\}$.
- Δ_{top}: a function $S \rightarrow K$.

Intuitively, if $((s, a, q), (s', q')) \in \Delta_{push}$, then whenever M encounters on its top-down traversal $v' = a$ from the ETM schema and stack s is in state q, it pushes q' onto s' where $\Delta(q, a) = q'$ or $q'.name = dummy$. Similarly, if $((s, a, q), (s, q')) \in \Delta_{pop}$, then whenever M encounters on its bottom-up traversal $v' = a$ and stack s is in state q, it pops q from s and leaves it with state q'. An $s \rightarrow \{q_i, q_{i+1}, \ldots, q_h\} \in \Delta_{traverse}$ is a function that returns the set of states in stack s such that $q_j.name \notin \{\epsilon, dummy\}$.

An ExStack is an ordinary stack $s \in S$ with the following additional items:

- *splitTag*(s): the XML element e_j that may cause a split from stack s. chain.
- *prevStack*(s): the stack s' that caused the addition of s when a split on s' occurred.
- *level*(s): a value indicating the relative level of the descendant from its ancestor in stack s' when it split.
- *count*(s): a value that indicates the depth of the top of the stack state from the ancestor state q_j.
- *levelsOfSplit*(s): the levels on which the descendant e_j can split. It is derived from the ETM.

Due to space limitations, the Rewriting Algorithm is not shown in this paper but is included in a larger report.

	Nasa	Random
Number of documents for XSExtract	15	75
Number of documents for processing	120	300
Average size of the document(KB)	7.85	0.35
Maximum number of Levels	6	7
Number of elements	58	27

Table 1: Data Sets.

3.3.5 *Error Analysis*

We provide in this sub-section a short analysis of the error rate that might be introduced with our approach of query rewriting. Given an XPath expression $\mathcal{P}_i = l_1 l_2 \ldots l_f$, a tolerated error rate η_i, and a derived ETM schema $(\mathcal{S})$ with nesting degree N, each $l_j = /e_j \mid //e_j$ is associated with $h_j = N - level(e_{j-1})$. Rewriting $\mathcal{P}_i$ might introduce a loss of precision $\mathcal{E}_i$ whose upper bound is η_i. This imprecision is added to α—an expectation value that is estimated at schema extraction time and indicating the maximum error rate of an XML schema with respect to subsequently streaming documents. The accuracy of α depends heavily of the number of streaming documents based on which the extracted schema was built. $\mathcal{E}_i$ is defined as $\mathcal{E}_i = \sum_{j=1}^{f} \mathcal{E}(l_j) + \alpha$

$$\text{where } \mathcal{E}(l_j) = \begin{cases} \sum_{l=m+1}^{h_j} Pr(e_i \mid e_{i-1}, l), & \text{if } l_j = //e_j \\ & \text{and } 1 - \mathcal{E}(l_j) \leq \eta_i \\ 0, & \text{otherwise.} \end{cases}$$

In case $\mathcal{E}(l_j) \neq 0$, the tolerated error rate η_i is updated to $\eta_i = \eta_i - \mathcal{E}(l_j)$ while making sure that $\eta_i \geq 0$.

4. EXPERIMENTAL RESULTS

We describe in this Section some of the experimental results that we got from implementing and experimenting with our proposed system. We run two sets of streaming XML documents on it. XPath queries were generated by a custom-build query generator where we could control different parameters setting.

These experiments have been carried out on two data-sets: *NASA* and *Random*. *NASA* is a well-know public domain XML data set, and *Random* is a synthetic XML data that we generated using another custom-build data generator that helped us control different characteristics of the data. For each data-set, we used three corresponding workloads of 5000 queries. Table 1 shows the initial settings of the experiments that we are reporting here.

Figures 3 and 4 show the gain in performance when rewriting is allowed (WR) versus no rewriting (WNR). η is the tolerated error rate. We can see in Figure 3 that for 1k queries, rewriting with 0 error rate allowed has quite a significant advantage over no rewriting. In Figure 4 (with a logarithmic scale,) things are even more expressed, since without rewriting, the system run out of memory beyond 3k queries.

We got almost the same trends with the *Random* data set; the only difference was that when we stressed the distribution of elements in the generated data sets, the schema was not perfect, and this resulted in missed matches as is shown in Figure 6.

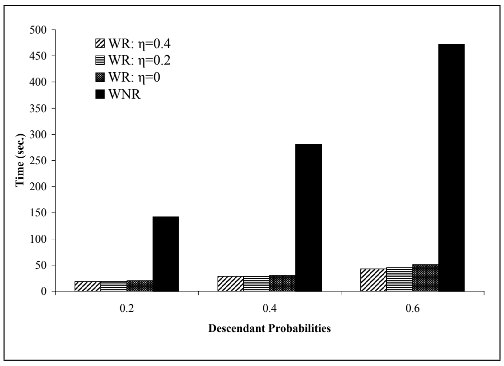

Figure 2: Performance of Rewriting (RW) versus no Rewriting (WNR) on 120 *NASA* documents for 1k Queries.

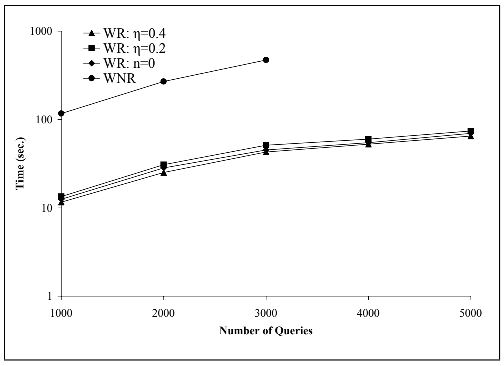

Figure 3: Performance of Rewriting (RW) versus no Rewriting (WNR) on 120 *NASA* documents while varying the number of XPath Queries.

5. RELATED WORKS

Several papers mentioned query rewriting in XPath and XQuery but, to the best of our knowledge, no paper reported an implementation and an analysis of this issue. Most works that we are aware of dealt with multiple query processing [6, 2, 5, 8, 9, 11], and schema extraction [7, 4, 3]. We used some of the ideas reported in these works, but we mostly developed our own ideas in the rewriting parts.

6. CONCLUSION

We presented in this paper our approach for query rewriting in a streaming environment where the rewriting in done based on an extracted schema. We provided an overview of this rewriting, some analysis on the error rate it introduces, and the results of some experiments we made. We intend in the next phase to extend this approach to XQuery directly in order to study the impact of query rewriting in XQuery on the error rate.

7. REFERENCES

[1] http://www.cs.aub.edu.lb/boulos/xamarkand.

[2] M. Altinel and M. Franklin. Efficient filtering of xml documents for selective dissemination of information. *Proceedings 26th International Conference on Very Large Databases(VLDB)*, 2000.

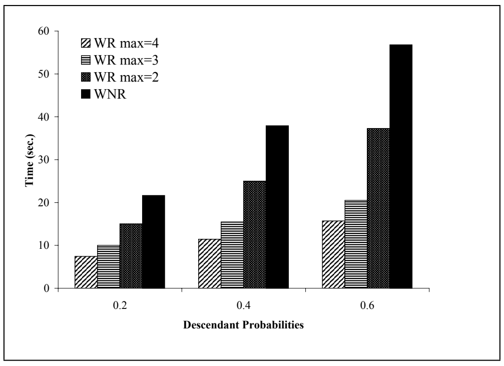

Figure 4: Performance of Rewriting (RW) versus no Rewriting (WNR) on 300 *Random* documents for 1k Queries.

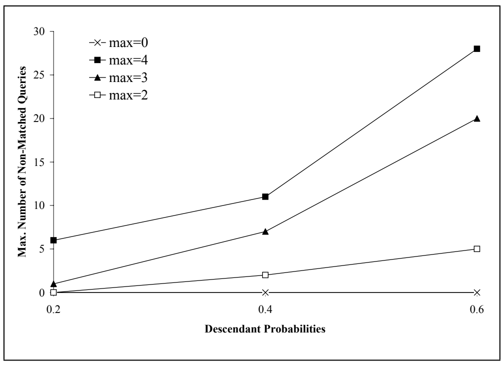

Figure 5: The Average number of missed documents for 5k queries on the *Random* data set.

[3] A. Arasu and H. Garcia-Molina. Extracting structured data from web pages. *Proceedings 2003 ACM SIGMOD International Conference on Management of Data*, 2003.

[4] B. Chidlovski. Schema extraction from xml collections. *Proceedings 2002 ACM JCDL*, 2002.

[5] F. M. Diao, Y. and P. Fisher. Yfilter: Efficient and scalable filtering of xml documents. *Proceedings 18th International Conference on Data Engineering(ICDE)*, 2002.

[6] P. Eugester. The many faces of publish/subscribe environments. *ACM Computing Surveys*, 2003.

[7] M. Garofalakis. Xtract: A system for extracting document type descriptors from xml documents. *Proceedings 2000 ACM SIGMOD International Conference on Management of Data*, 2000.

[8] T. Green. Processing xml streams with deterministic finite automata. *Proceedings 9th International Conference on Database Theory(ICDT)*, 2003.

[9] A. Gupta and D. Suciu. Stream processing of xpath queries with predicates. *Proceedings 2003 ACM*

SIGMOD International Conference on Management of Data, June 2003.

[10] R. Harb. Evaluating xqueries over xml streaming data. Master's thesis, American University of Beirut, 2004.

[11] F. Peng and S. Chawathe. Xpath queries streaming data. *Proceedings 2003 ACM SIGMOD International Conference on Management of Data*, 2003.

APPENDIX

Appendix A

Algorithm 1 Construct-TM(T, S)

```
    ▷ Constructs Tree Model Schema $T(V, E)$ using Stack $S$
 1: procedure createNode($label, count, depth$)
 2:     $i \leftarrow |V| + 1$
 3:     $V \leftarrow V \cup u_i$
 4:     $element(u_i) \leftarrow label$
 5:     $count(u_i) \leftarrow 1$
 6:     $level(u_i) \leftarrow depth$
 7:     $traversed(u_i) \leftarrow$ true
 8:     return $(u_i)$
 9: end procedure

10: procedure updateChild($u_i, child, depth$)
11:     if $\exists\, edge(u_i, u_j) \in E$ and $element(u_j) = child$
          then
12:         if $traversed(u_j) =$ false then
13:             $count(u_j) \leftarrow count(u_j) + 1$
14:         end if
15:     else
16:         $u_j \leftarrow$ createNode($child, depth$)
17:         $E \leftarrow E \cup edge(u_i, u_j)$
18:     end if
19:     return $(u_j)$
20: end procedure

21: procedure startElement($a, attrs, d$)
22:     if isNull($T$) then
23:         $u_i \leftarrow$ createNode($a, d$)
24:     else if $S.size = 0$ then
25:         $u_i \leftarrow$ root($T$)
26:         $count(u_i) \leftarrow count(u_i) + 1$
27:         $traversed(u_i) \leftarrow$ true
28:     else
29:         $u_i \leftarrow$ updateChild($S.top(), a, d$)
30:     end if
31:     $S.push(u_i)$
32: end procedure

33: procedure endElement($a, d$)
34:     $S.pop()$
35:     if $S.size = 0$ then
36:         for all $u_i \in V$ do
37:             $traversed(u_i) \leftarrow$ true
38:         end for
39:     end if
40: end procedure
```