Sub-Document Queries Over XML with XSQuirrel

Arnaud Sahuguet
Bell Labs Research
600 Mountain Avenue
Murray Hill, NJ, USA
sahuguet@lucent.com

Bogdan Alexe
Ecole Nationale Supérieure des Télécoms
46 rue Barrault
75013 Paris, France
bogdan.alexe@enst.fr

ABSTRACT

This paper describes XSQuirrel, a new XML query language that transforms a document into a sub-document, i.e. a tree where the root-to-leaf paths are a subset of the root-to-leaf paths from the original document.

We show that this type of queries is extremely useful for various applications (e.g. web services) and that the currently existing query languages are poorly equipped to express, reason and evaluate such queries. In particular, we emphasize the need to be able to compose such queries. We present the XSQuirrel language with its syntax, semantics and two language specific operators, union and composition.

For the evaluation of the language, we leverage well established query technologies by translating XSQuirrel expressions into XPath programs, XQuery queries or XSLT stylesheets.

We provide some experimental results that compare our various evaluation strategies. We also show the runtime benefits of query composition over sequential evaluation.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Query Languages

General Terms

Algorithms, Performance, Languages

Keywords

XSQuirrel, XML, sub-document

1. INTRODUCTION

Let there be no misunderstanding. In this paper, we will try to sell you yet another query language for XML. Our language has a catchy name, but more importantly we think it solves a real problem, in the current web services architecture, that is not properly addressed by the existing query languages for XML.

XML is about trees, and people are rather familiar with tree structures: genealogical trees, tree hierarchy in file systems or bookmarks, etc. Yet, the current query infrastructure for XML does not support a simple operation that consists of taking a document and return a sub-document. As we will show, this is a quite natural operation, that happens to be very useful in a lot of application domains.

But let's start by defining what we mean by sub-document. The notion of a sub-document is a totally natural concept when you think about it. This is nothing more than the counterpart of the result of a select-project (SP) in the relational model.

The relational model deals with tables. SP operations remove rows and columns to produce sub-tables. XML deals with documents and we want to remove sub-trees to produce sub-documents.

Note that sub-documents are not properly captured by existing query languages. XPath consumes a document to produce a node-set. XQuery creates new node ids. We will elaborate more in the next section.

1.1 The notion of sub-document

An XML *document* is a tree defined by $D = (N, V, \lambda, <_d)$ where: (i) N is the set of nodes in the document with n_0 a designated node which is the document's root; (ii) $V \subseteq N \times N$ is the parent/child relationship between nodes; (iii) λ is a function that associates each node with a label; and (iv) $<_d$ is an ordering relation on the nodes of the document.

A *sub-document* $D' = (N', V', \lambda', <'_d)$ of an XML document $D = (N, V, \lambda, <_d)$ is defined as follows: (i) D and D' have the same root; (ii) $N' \subseteq N$; (iii) $V' \subseteq V$; (iv) $\lambda' = \lambda$ and (v) $<'_d = <_d$.

Another way to look at it is to consider a document as the set of *root-to-leaf* paths. In the example of Fig. 1, the set is:
{`/A/B/C,/A/B/D/DD,/A/B/D/EE,/A/B/F/FF,`
`/A/B/F/GG,/A/B/H,/A/B/D/DD,/A/B/D/EE,/A/B/D/II`}.
A sub-document corresponds exactly to a subset. For our example, the subset is:
{`/A/B/D/DD,/A/B/D/EE,/A/B/H,/A/B/D/DD,`
`/A/B/D/EE,/A/B/D/II`}.

Yet another way is to say that a sub-document is the original document where some sub-trees (down to the leaves) have been removed.

1.2 The use for sub-documents

There are many uses for sub-documents. Sub-documents can be used to define XML views. Sub-documents correspond to join-free queries. This is natural for data integration. For instance, one can define views over XML sources, each view being a sub-document of the actual source. These views can then be joined to produce another document. Sub-documents can also be used to define access control over XML data. A sub-document corresponds to the data that can be seen.

Sub-documents are also very useful for data merging and synchronization because the structure of the original document is preserved. Algorithms like the deep union [8] require such information.

Finally, we think that sub-documents can also be used for distributed query processing of XML data. First, they permit to ship around only the relevant parts of the document. Given the verbosity of XML, this can be a huge save in bandwidth-conscious environments. Second, they permit to ship enough structure (path to the root) to conduct semi-join [19] like algorithms over XML.

1.3 One very concrete application: user profile management in GUP$^{\text{ster}}$

We now describe how the notion of sub-document naturally came into play for the privacy conscious management of user profile information.

In today's networks, user profile information (e.g. address book, presence, location, calendar, etc.) is scattered all over. Various initiatives [1, 14] are standardizing XML-based solutions to offer a web service single point of access for this information.

The GUP$^{\text{ster}}$ project [22, 2, 13, 21] at Bell Labs goes in the same direction, with a dual emphasis on integration and access control. In this context, we are looking to: (1) integrate, into a common schema and on a per user basis, XML data coming from various sources; (2) permit users to define access control rules over the integrated data; and (3) let applications query the data while enforcing user privacy. This yields the need for the following concepts:

- a **query**, defining what *portion of the user profile* is requested.
- a **mapping rule**, defining where a *portion of the user profile* is stored (mapping to a data source).
- an **access control rule**, defining under what condition a *portion of the user profile* can be accessed (mapping to boolean function).

Quite naturally, a *portion of the user profile* corresponds to a sub-document of the user profile. One question we have to answer is how we can express sub-documents.

Moreover, for efficiency, we want to avoid to retrieve data that is not needed in the final answer either because it is not part of the query or because it not visible due to access control rules. Therefore, we envision the processing of an incoming query Q as follows. Given a user profile D, a set of mappings $\{M_i\}$, a set of access control rules $\{ACR_j\}$, we want to compute $D' = f(\{M_i\}, \{ACR_j\}, Q, D)$. Or maybe even better, we would like to compute $D' = g(\{M_i\}, \{ACR_j\}, Q,)(D)$. This way, instead of having to send the query and apply the access control on the result that is sent back, we send to each source a query compatible with the access control. This is much more efficient.

The rest of the paper is organized as follows. In Section 2, we explain why currently available query languages are not appropriate for this new kind of queries. We then present the language itself, with its syntax, semantics and operators (union and composition). In Section 5, we describe various strategies to evaluate XSQuirrel queries over XML documents by translating expressions into other query languages: XPath, XQuery and XSLT. We then present some experimental results that (1) compare the various evaluation strategies and (2) show the benefit of composition operator, for queries and data over some XMark [23] dataset. Some related work is discussed in Section 6, before we offer our conclusions and some future work.

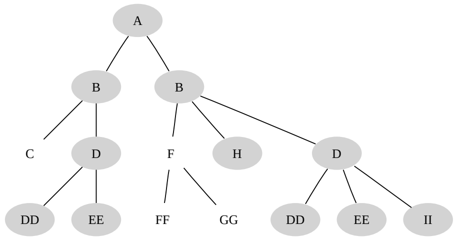

Figure 1: Original document D and sub-document D' defined by $q_1(D)$ (grey marking).

2. A NEW LANGUAGE, REALLY?

In this section we argue that we need a new language because the existing ones do not address the issue of sub-document queries properly. We explain the shortcomings of XPath, XQuery and XSLT and list some requirements for the design of our new language.

2.1 Why not XPath?

The XPath 1.0 [10] has been designed mostly as a navigation language that returns a subset of the nodes of a document. For instance, XSLT uses XPath heavily to match patterns that need transformation.

When applied on a document, XPath returns a nodeset and not a sub-document. From the nodes, it is always possible to reconstruct the document (using the context to find the ancestors of the current node up to the root), but this is not the default behavior and the application using XPath needs to perform this reconstruction. Moreover, when the context is lost (e.g. data shipped from a remote site), this information is lost.

The main problem with XPath is that the input and the output of the query are not the same domain and therefore queries cannot be composed.

2.2 Why not XSLT?

Even though XSLT has the notion of document built-in in its semantics with templates being applied to the current document, it is a language hard to reason about because of its rule-based nature.

Moreover, the output of an XSLT transformation is not necessarily a document.

2.3 Why not XQuery?

XQuery [9] is *the* general purpose query language for XML. The problem with XQuery is three-fold.

First, in our setting, this is like a hammer to kill a fly, since sub-document queries are restricted in nature (remember the analogy with select-project in the relational algebra). Moreover, XQuery does not know about the sub-document semantics.

Second, XQuery processes a document by first deconstructing it into FLWR bindings and then reconstructing it. Not only does this imply a lot of syntax, it also requires the creation of new node ids. When the sub-document consists of only a few nodes removed from the original document, this is a lot of overhead.

Third, XQuery is so rich and powerful that it is very hard to reason about. XQuery is by nature composable. For two queries Q_1 and Q_2 over document D, you can always represent the composed query as `let $x:=`$Q_1(D)$ `return` Q_2`($x)`. But it might be hard to optimize such an expression.

2.4 A new language

Based on the previous considerations, we decided to go for a new domain specific language that would fit our needs and fulfill the following requirements:

- built-in sub-document semantics
- composability of queries
- expressive enough to be useful
- simple enough to be reasoned about and optimized
- concise syntax
- friendly with other XML languages (e.g. reuse of syntax, translation to other languages for evaluation)

3. THE XSQuirrel LANGUAGE

In this section we present the XSQuirrel language. It is based on XPath 1.0 and therefore XSQuirrel is rather a family of languages, depending on which fragment of XPath 1.0 we decide to use.

All these languages – of course – share the same semantics but the details of some of the algorithms may be different, depending on the expressive power of the fragment we use. For a detailed study of these issues and a more theoretical presentation of XSQuirrel, we refer the reader to [5].

In the rest of the section, we will consider **a very limited fragment of XPath**[1]. This is the one we are using in the context of GUP$^{\mathsf{ster}}$ and it has shown so far to be expressive enough for our application domain of privacy conscious integration of XML data.

3.1 Syntax

XSQuirrel expressions are built from a finite set of labels (e.g., tags, names) Σ of an XML schema S. The fragment of the language that we consider in this paper is syntactically defined as follows:

$$p \;:=\; \epsilon \mid l \mid p/p \mid p/(p \cup p) \mid p[q]$$

where ϵ, l denote the empty path (”.” in XPath) and a name in Σ respectively; $\cup$ stands for *union*; ”/” stands for XPath concatenation but here is also used as the XPath *child* axis. q in $p[q]$ is called a *qualifier* and is defined by: $q := p \mid label = v \mid \text{not}(q)$.

From XPath 1.0, we have kept the child and attribute axis and a restricted form of value-based predicates.

EXAMPLE We give below some examples of XSQuirrel expressions.

q_1 : $/A/B/(D \cup H)$
q_2 : $/A/B[H]/(D/DD \cup F)$
q_3 : $/A/(B[C] \cup B[H]/(D/II \cup F/FF))$
q_4 : $/A/B[D/EE]/(D/DD \cup H \cup F)$

Query q_1 for instance returns `D` and `H` nodes, that are children of `B` nodes, themselves children of `A` nodes. Along with these nodes, their descendants, and ancestors up to the root node of the document on which q_1 is evaluated are returned.

[1]The reason why we emphasize this point is that some of the results and conclusions we present here might be different for larger fragments of the language that are still under study.

3.2 Semantics

Intuitively, the result of the evaluation of an XSQuirrel expression q on a document D is document $q(D)$ (sub-document of D) obtained as follows:

1. evaluate q using the usual XPath
2. for each node n obtained from the previous step, get its *descendant* nodes, and its *ancestor* nodes up to the root of D
3. finally, $q(D)$ is constructed by removing all nodes of D that are not in the set of nodes from the previous step (note that the resulting document $q(D)$ is a sub-document of D).

EXAMPLE Consider the XML document D illustrated in Fig. 1 (ignore the grey marking for now) and query q_1 given in Example 3.1. The result of evaluating q_1 over D is sub-document $q_1(D)$ where the nodes have been marked in grey. More specifically, this document is defined by the `D` and `H` nodes returned when evaluating q_1 as an XPath expression on D, their descendants and ancestors up to the root node of D.

More formally: the result of evaluating an XSQuirrel expression q against a document $D(N, V, \lambda, <_d)$ is a sub-document $D'(N', V', \lambda', <'_d)$ of D such that: (i) N' is defined as:

$$N' = [[D]]_q \bigcup_{n \in [[D]]_q} n[[D]]_{\Downarrow_* ::*} \bigcup_{n \in [[D]]_q} n[[D]]_{\Uparrow^* ::*}$$

where $n[[D]]_p$ denotes the set of nodes returned by evaluating XPath expression p on the node n of document D (n is omitted when it is the root), $\Downarrow_*$ and $\Uparrow^*$ are the XPath *descendant* and *ancestor* axis respectively. and (ii) $V' = \{(n_1, n_2) \in V \mid n_1, n_2 \in N'\}$.

3.3 Language operators

At the level of the language we define two language (i.e. syntactic) operators – union ($\cup$) and composition ($\circ$) – such that: $\forall D, (Q_1 \cup Q_2)(D) = Q_1(D) \cup Q_2(D)$ and $\forall D, (Q_1 \circ Q_2)(D) = Q_1(Q_2(D))$. Here the union of two sub-documents corresponds to the union of their nodes.

For lack of space, we only provide the intuition for the algorithms of union and composition. We refer the reader to [5] for a more formal presentation.

3.3.1 *Union*

This operator is pretty straightforward. For both XSQuirrel expressions, we can naturally derive two sets of XPath expressions by distributing over $\cup$. We can then take the union (as sets) of the two, recombine them as an XSQuirrel expression and normalize them.

We provide an example below. The evaluation of the queries over a document is presented in Fig. 2.

```
q2: /A/B[H]/(D/DD ∪ F)
q3: /A/(B[C] ∪ B[H]/(D/II ∪ F/FF))
q2 ∪xq q3: /A/(B[C] ∪ B[H]/(D/(DD ∪ II) ∪ F))
```

3.3.2 *Composition*

This operator is more complicated because it is not symmetric. We distinguish between the inner (Q_i) and the outer expression (Q_o).

The core of the composition algorithm must ensure two things. First, that the final expression is as selective as

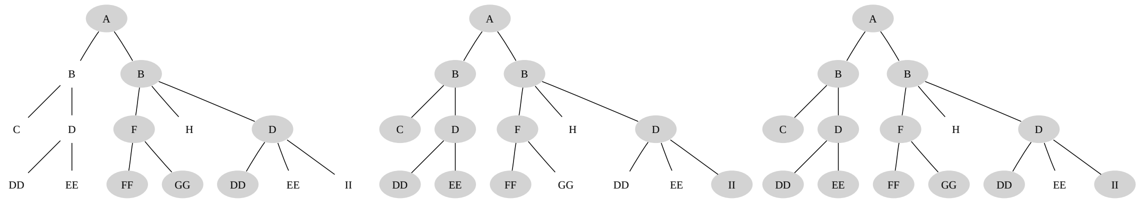

Figure 2: Union: $q_2(D)$ (left), $q_3(D)$ (middle) and $(q_2 \cup_{xq} q_3)(D)$ (right).

the more selective of the two: `/A/B` composed with `/A/B/C` is `/A/B/C`. Second, that the predicates from the outer expression correspond to paths that are defined in the inner expression. This makes perfect sense when you think of the inner query as the one defining an access control view for instance.

EXAMPLE Here is an example.

Q_o	$/A/(B[C] \cup B[H]/(D/II \cup F/FF))$
Q_i	$/A/B[D/EE]/(D/DD \cup H \cup F)$
$Q_o \circ Q_i$	$/A/B[H][D/EE]/F/FF$

We see that node `B[C]` of Q_o does not appear in the composed query (the path $/A/B/C$ for node `B[C]` is not satisfied by the inner query). Node `B[H][D/EE]` is created from nodes `B[H]` and `B[D/EE]` of the outer and inner queries, respectively. Node `D` (and its children) disappears from the resulting query since the outer query (Q_o) requests `II` nodes but the inner query Q_i returns only `DD` nodes. Finally, node `FF` requested by the outer query is added below node `F` (the inner query returns the subtree of `F` but the outer query requests only its `FF` sub-nodes). The evaluation of the queries over a document is presented in Fig. 3.

4. EVALUATING XSQuirrel

In Section 3, our description of the semantics of the XSQuirrel language already implies one way to evaluate expressions using XPath. In this section, we present three evaluations strategies that translate expressions into another query language, mainly XPath, XQuery and XSLT. We provide in Fig. 11 a detailed example of the three translations.

4.1 By translating to XPath programs

The XSQuirrel language is more expressive than the subset of XPath it relies on, because of the union operator and the sub-document semantics. Therefore we cannot translate XSQuirrel expressions into XPath expressions, but rather into XPath programs. The algorithm is presented below.

Algorithm 1: XPath evaluation program

```
  Input  : $D$, $xsq$
1 markedNodes := {}
2 xpathList := expand(xsq)
3 foreach e in xpathList do
4     nodeset := XPath(D, e)
5     foreach n in nodeset do
6         markedNodes += {n}
7         markedNodes += descendant-of(n)
8         markedNodes += ancestor-of(n)
9 D' := trimNodes(D, markedNodes)
  Output : $D'$
```

The intuition is to expand (using the `expand` function) the XSQuirrel expression into a set of XPath expressions, by distributing over the union operator. For each XPath expression, we evaluate the query over the document. We mark the nodes from the result nodeset by putting them in `markedNodes`. For each marked node, we also mark its descendants and its ancestors up to the root. Finally, we remove from the original document all the nodes that have not been marked.

4.2 By translating to XQuery

The good news with XQuery is that because of the expressive power of the language, we can translate any XSQuirrel expression into a single XQuery expression. The bad news is that XQuery requires to deconstruct the document (by binding to FLWR expressions) first and then reconstruct it.

XQuery can easily take care of the union using nodeset concatenation. We need however to enforce the sub-document semantics. For instance, `a/(b ∪ c)` cannot be translated as {`for $x in a/b return $x, for $x in a/c return $x`} because it forces `b` to appear before `c`, which is not necessarily the case in the original document. The right way to do it is to iterate over the children (thus preserving the document order) and check the nature of the child, using an `if` statement and a predicate such as `[self::b]` or `[self::c]`.

This is not going to work either because an expression may contain a union of overlapping paths, such as `a/( b[p1] ∪ b[p2])`. To avoid some subtrees to be added more than once, we need to make sure that the `if` statements are exclusive of each other. For our toy example, this leads to:

```
for $x in a/* return if $x[self::b[p1]] then $x
else if $x[self::b[p2]][not(self::b[p1])]
then $x else {}
```

We describe the translation from XSQuirrel to XQuery using two functions T (for translation) and P (for predicates) defined as follows. For simplicity, we will ignore expressions with attributes.

Each function is described in terms of production rules that consume the structure of the XSQuirrel expression.

For the predicate function Pred, a step with no children simply returns itself. In the presence of children, the predicate consists of the recursive concatenation (using boolean or) of the predicates of the children. For instance `/a[p]/(b ∪ c/d)` will return `a[p][b | c[d]]`.

Pred:	
	`a[p]`
→	`a[p]`
	`a[p]/(`a_1`[`p_1`] ... ∪ `a_n`[`p_n`])`
→	`a[p][ Pred(`a_1`[`p_1`]) ...	Pred(`$a_n$`[`$p_n$`])]`

Figure 4: Translating XSQuirrel to XQuery

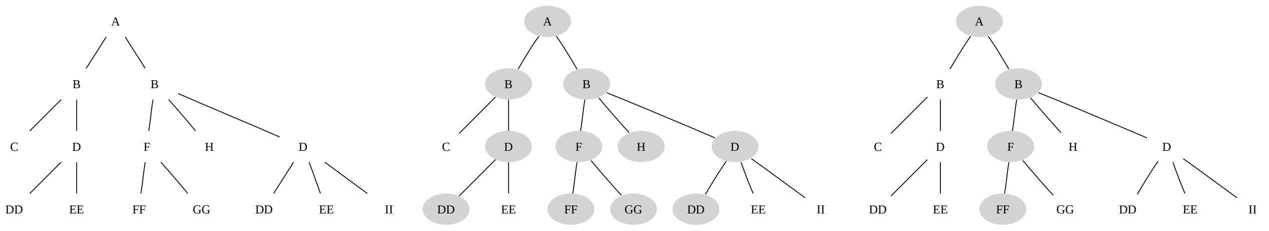

Figure 3: Composition: D (left), $q_i(D)$ (middle) and $q_o \circ q_i(D) = q_o(q_i(D))$ (right)

The translation function T takes three parameters: the location step of the XSQuirrel expression, the current binding (used by the FLWR expression) and a list of predicates. The list of predicates is used to ensure that each `if` statement is exclusive of the others. The XQuery generation proceeds as follows:

- the binding provides the current node to process
- we check that the node satisfies the current location step by applying the predicate `self::Pred(a[p])`
- we make sure that the `if` case is unique by applying the list of the predicates
- if the location step has no children, we return the node or nothing
- if the location step has children, we output the tag name of the current node, we iterate over all its children via a FLWR expression and we close the tag name

```
T:
     /a/path , $$, {}
→    for $x in /* return T(a/path, $x, {})
     a[p], $x, {P_1,...P_m}
→    if $x[self::Pred(a[p])][not(self::P_1)]...
     ...[not(self::P_m)]
     then $x else {}
     a[p]/(a_1[p_1] ... ∪ a_n[p_n]), $x, {P_1 ... P_m}
→    if $x[self::Pred(a[p]/(a_1[p_1] ... ∪ a_n[p_n])]
     and $x[not(self::P_1)]...[not(self::P_m)]
     then <a>
       for $x_{+1} in $x/* return
       T(a_1, $x_{+1}, {} )
     ...
       T(a_n, $x_{+1}, {Pred(a_1) ... Pred(a_{n-1})} )
     </a>
     else {}
```

Figure 5: Translating XSQuirrel to XQuery

The FLWR expression generates a new unique binding (noted by $\$x_{+1}$). For each child, we call T, with the following parameters: the corresponding location step a_i, the new binding $\$x_{+1}$ and a new list of predicates. For each child, the list of predicates corresponds to the predicates from the previous children (Pred(a_1), ..., Pred(a_{n-1})).

4.3 By translating to XSLT

It turns out that XSLT is a much more natural language for sub-document queries. The notion of document is somehow built-in in the semantics of the language itself because of its rule/template system.

The union operator of XSQuirrel can naturally be translated into XSLT templates that will be applied following the structure of the document. Unlike with XQuery, we don't have to worry about deconstructing the document and then reconstructing it by taking good care of the order.

The XSLT ruleset we generate consists of 3 modes of operations: default, regular and leaf. For default and regular, we define default templates that don't do anything. For leaf, the template calls recursively templates for the content of the subtree.

From the navigation step of the XSQuirrel expression, we derive a rule that consumes the root (in mode default) and calls a rule for this step.

The other rules are derived from the XSQuirrel expression as follows, using translation function T' defined in Fig. 6. For sake of clarity, we do not show the exact XSLT rules being generated but abbreviate them using their key components: for the outer template, its mode and match; for the inner template, its mode and its select (the nodes for which the inner template must be applied).

Rule 1: For the first navigation step of the XSQuirrel expression, we generate a template that consumes the first element in mode default and calls for a template over the same element in mode regular.

Rule 2: We translate the last location path of an XSQuirrel expression (`a[p]`) by generating a template that matches the current node and calls for a template over the same node but in mode `leaf` in order to return as is the entire subtree.

Rule 3: The last translation rule generates a template that matches the current node with the predicates corresponding to its child nodes (we reuse the Pred function we defined for XQuery). It calls for a template that matches any of the children. We translate recursively each child element of current location step.

```
T':
1    /a/path
→    template:  mode="default", match="/"
     apply:  mode="regular", select="a"
     T'(a/path)
2    a[p]
→    template:  mode="regular", match="a[p]"
     apply:  mode="leaf", select="."
3    a[p]/(a_1[p_1] ∪ ... ∪ a_n[p_n])
→    template:  mode="regular",
     match="Pred(a[p]/(a_1[p_1] ∪...∪ a_n[p_n]))"
     apply:  mode="regular",
     select="Pred(a_1[p_1]) |...| Pred(a_n[p_n])"
     T'(a_1[p_1])
     ...
     T'(a_n[p_n])
```

Figure 6: Translating XSQuirrel to XSLT

Note: We are aware that we do not provide any formal proof of the correctness of the translations, except for the

XPath translation that corresponds exactly to the semantics of the language. This is left for further study. For all the experiments we ran (see next section), we have checked that the results provided are all equivalent. We are aware of some *pathological cases* of XSQuirrel expressions that get improperly translated into XSLT because of ambiguities. A proper characterization of such cases is also left for further study.

5. EXPERIMENTAL RESULTS

In this section, we want to answer two questions: (1) which is the better way to evaluate XSQuirrel expressions; (2) what is the runtime benefit of query composition, if any.

5.1 Comparing the various strategies

For this experiment, we consider various queries over some XMark [23] generated data and compare their execution time in five different configurations:

- XPath: we use the Apache Xerces implementation because it offers access to node index information. We use the index value as a way to mark the nodes returned by the evaluation of the XPath expressions. The document is parsed once and built in main memory. The nodes not part of the final result are removed.
- XSLT (Xalan): query translated into XSLT and applied using the Java JAXP API.
- XSLT (Saxon8): query translated into XSLT and applied using the Java JAXP API.
- XQuery (Saxon8): query translated into XQuery and applied using the Java JAXP API.
- XQuery (GALAX): query translated into XQuery and applied using the command-line.

We have created five representative queries that demonstrate various features of the language (see Fig. 9).

To avoid comparing apples and oranges, we stream the final result into a SAX content handler that computes the normalized hash of the result document. This permits to check that (1) all the evaluators provide the exact same result and that (2) all content is accessed (some implementations sometimes perform lazy evaluation). The results are presented in Fig. 7.

The XPath evaluation should be looked at as the base strategy since it does things in a straightforward and naive manner, with no room for optimization (XPath expressions applied sequentially). Also note that Xalan and GALAX are clearly not as competitive as Saxon8.

Overall, both translations to XQuery and XSLT are better than the naive XPath program. The XQuery translation seems to be a bit more efficient, due probably to the built-in optimizations and the deterministic nature of the evaluation. XSLT rules can be inherently ambiguous and the engine tries to pick the best match (most restrictive rule). Note also that the way we translate into XSLT introduces some redundant checks (in the `select` and in the `match`) that are probably not optimized (e.g. keep track that the check was successful for a given node).

Some preliminary experiments (not reported here) with an early native evaluator for XSQuirrel show on par performance with XSLT and XQuery. The skeptical will argue that native evaluation does not make sense and that resources should be focused on general purpose query engines. The optimistic will counter argue that being on par with XQuery and XSLT is a very good start and that with more work, evaluation time should be improved significantly and maybe reach the high-level performance shown by native XPath engines [7, 4].

5.2 The benefits of language composition

The second thing we want to measure is the benefit (if any) of our language-based composition for our fragment of the language[2]. Using XSQuirrel, we can replace the sequential evaluation of two queries with the evaluation of just one.

For pairs of queries Q and Q', we compare the evaluation of the composed query $Q \circ Q'$ (computed using our algorithm) with the evaluation of the sequential queries. For XSLT, we can chain the two transformation using `XMLFilter`s from the JAXP API. Note that this does not permit any optimization between the two rulesets. For XQuery, we represent the chained queries using the following XQuery expression: `let D':= Q(D) return Q'(D')`.

The six queries we use for our experiments are detailed in Fig. 10. The results are presented below:

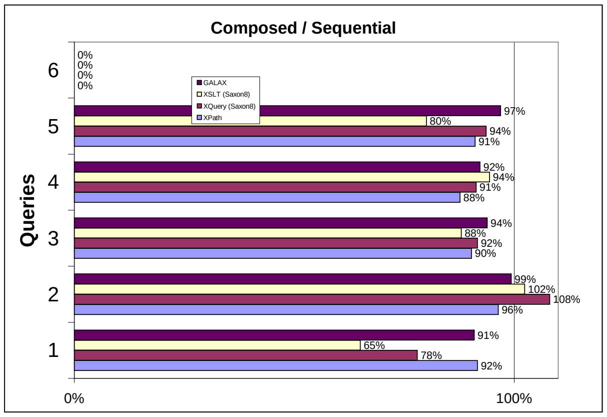

Figure 8: Composed vs sequential evaluation.

The first thing to note is that there is no groundbreaking benefit when using composition. Depending on the queries and the evaluators, improvements are around 10-20%. In a web services environment, with millions of queries a day, this makes a nice difference at the end of the day though. Composition is a clear winner when the composed query is empty (e.g. `Q6`), a very frequent case when one query is used to define access control view. Knowing that the query is empty ahead of time not only saves a lot of processing times, it also saves on communication costs in a distributed environment. Both aspects can be extremely valuable.

6. RELATED WORK

There is a lot of on-going work in both research and industry community around the already existing query languages (XPath, XQuery and XSLT). See for instance [16].

The idea of returning subtrees appears in the context of distribution and replication of XML documents in [3]: the fact that a subtree of a node should be returned has to be explicitly defined in the XPath expression and is not inherent in the semantics of the language as for XSQuirrel.

[2]We want to re-emphasize the fact that these results are for the limited fragment of the language. For larger fragments, we expect the benefits to be larger.

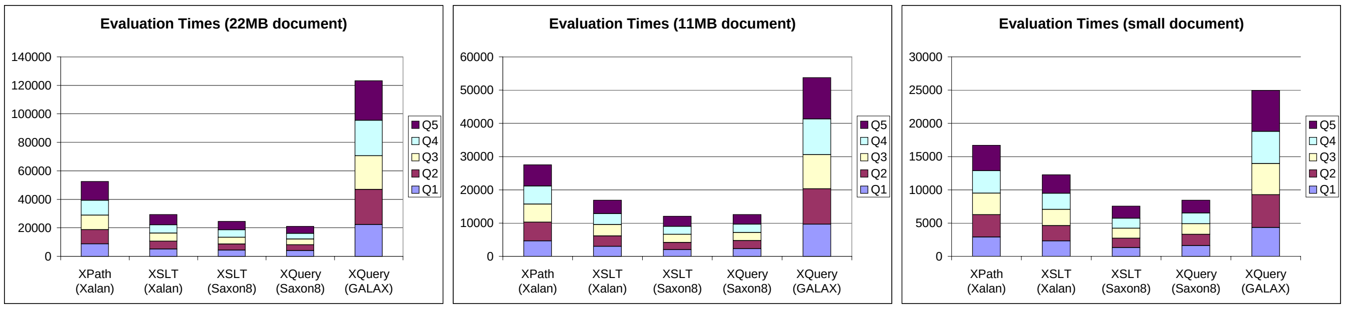

Figure 7: Comparison of the various evaluation strategies.

[18] follow a similar approach to [3] where XPath expressions return documents instead of sets of nodes. But the semantics of the language is not defined formally. The idea of having a project operator for XQuery was proposed in [17], but this is an internal algebraic operator. Some domain specific languages for XML have been proposed for various contexts such as integration with relational sources ([12]), or access control ([11, 6]).

The industry is also pushing for some new languages. In Liberty Alliance, a data service template [15] can define its own query language flavor (the suggested ones are sub-set of XPath). In XCAP[20] (proposed standard for next generation of telecom application), resources can be accessed using a restricted flavor of XPath.

The work on efficient XML processing over streams is also relevant here. Our early implementation of a native XSQuirrel evaluator is inspired by [7, 4], even though the different semantics of XSQuirrel does not make these results directly applicable.

7. CONCLUSION

In this paper, we argue that more and more applications will need to consider sub-document queries, where the result of the query is a sub-document of the original document. Synchronization, access control, distributed query processing are such examples.

The current query languages for XML do not address this specific issue: XPath returns nodes instead of a sub-document; XSLT and XQuery are too expressive and it is hard to guarantee statically what will be returned.

To address this issue, we introduce the XSQuirrel language. Following the XPath syntax, XSQuirrel offers a sub-document semantics where the result of a query is always a sub-document of the original document, which makes it possible to compose queries. One strength of the language is that queries can be composed *at the language level*. For two XSQuirrel queries Q_1 and Q_2, we can syntactically compute $Q_1 \circ Q_2$. Thus, instead of evaluating two consecutive queries, we can simply evaluate the composed query. We have shown the runtime benefits of this approach for various queries ran against the XMark dataset.

Another strength of the language is that it is possible to translate XSQuirrel expression into other XML query languages. Already existing high-performance query engines can therefore be reused and there is no need to build XSQuirrel specific ones. In the paper, we have described the translation algorithms and shown how the translated queries perform on the XMark data set.

There is still a lot of work to be done. Finding the right expressive power for the language (i.e. which fragment of XPath to choose from) is not easy, and new features added to the language may require to modify the rewriting, translating and evaluating algorithms. We also want to build a high-performance native evaluator that can reach the performance of stream-based XPath evaluators. Investigating the benefits of composition for larger fragments of the language is important too.

People have and will argue legitimately about the need for *yet another XML query language*. For our application domain (privacy conscious user profile management), we needed to combine integration and access control of XML profile data and the available query languages were simply not good enough for our needs. We engineered XSQuirrel to fill this gap. By making it close to XPath in syntax and by providing translators to XPath, XQuery and XSLT, XSQuirrel can be seen as some kind of *syntactic sugar* that can be really handy for our application domain. We think that XSQuirrel can also be useful in a broader context and we hope that this paper will convince more people to give it a chance.

Acknowledgments: Irini Fundulaki, Michael Benedikt, Nicola Onose, Guillaume Giraud, Nicolas Pombourcq, the members of the 3GPP GUP working group.

8. REFERENCES

[1] The Third Generation Partnership Project (3GPP). `http://www.3gpp.org`.

[2] S. Abiteboul, B. Alexe, O. Benjelloun, B. Cautis, I. Fundulaki, T. Milo, and A. Sahuguet. An Electronic Patient Record "on Steroids": Distributed, Peer-to-Peer, Secure and Privacy-conscious. In *VLDB*, 2004. (demo track).

[3] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML Documents with Distribution and Replication. In *SIGMOD*, 2003.

[4] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming XPath Processing with Forward and Backward Axes. In *ICDE*, 2004.

[5] M. Benedikt and I. Fundulaki. Specification and Composition of Subtree Queries. Technical Report, Bell Labs. `http://db.bell-labs.com`.

[6] E. Bertino, S. Castano, and E. Ferrari. Securing XML Documents: The Author-X Project . In *SIGMOD* , 2001 (demo track).

[7] F. Bry, F. Coskun, S. Durmaz, T. Furche, D. Olteanu, and M. Spannagel. The XML Stream Query Processor SPEX. In *ICDE*, 2005.

[8] P. Buneman, S. B. Davidson, W. Fan, C. S. Hara, and W. C. Tan. Keys for XML. In *WWW*, 2001.

[9] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and L. Stefanescu. XQuery: A Query Language for XML. `http://www.w3.org/TR/xquery`, February 2001.

[10] J. Clark and S. D. (eds.). XML Path Language (XPath) Version 1.0, 1999. `http://www.w3c.org/TR/xpath`.

[11] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure XML Querying with Security Views. In *SIGMOD*, 2004.

[12] M. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. SilkRoute: A framework for publishing relational data in XML . *TODS*, 27(4):438–493, 2002.

[13] I. Fundulaki and A. Sahuguet. Share your data, keep your secrets. In *SIGMOD (Demo)*, 2004.

[14] Liberty Alliance Project. `http://www.projectliberty.org`.

[15] Liberty Alliance ID-WSF Data Services Template Specification, Version 1.0. `http://www.projectliberty.org/specs/liberty-idwsf-dst-v1.0.pdf`, 2002.

[16] I. Manolescu and Y. Papakonstantinou, editors. *Proceedings of the First International Workshop on XQuery Implementation, Experience and Perspectives <XIME-P/>*, June 2004, Paris, France, 2004.

[17] A. Marian and J. Simeon. Projecting XML Documents. In *VLDB*, 2003.

[18] M. Petropoulos, A. Deutch, and Y. Papakonstantinou. Query Set Specification Language (QSSL). In *Informal Proc. WEBDB*, 2003.

[19] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw Hill, 2003.

[20] J. Rosenberg. The Extensible Markup Language (XML) Configuration Access Protocol (XCAP). IETF draft, Feb 2004. `http://www.jdrosen.net/papers/draft-ietf-simple-xcap-02.txt`.

[21] A. Sahuguet, B. Alexe, P.-Y. Laligand, A. Shikfa, and I. Fundulaki. User Profile Management in Converged Networks (Episode II): Share your data, Keep your secrets. In *CIDR*, Asilomar, CA, USA, January 2005. Online Proceedings.

[22] A. Sahuguet, R. Hull, D. Lieuwen, and M. Xiong. Enter Once, Share Everywhere: User Profile Management in Converged Networks. In *CIDR*, Asilomar, CA, USA, January 2003. Online Proceedings.

[23] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, M. J. Carey, I. Manolescu, and R. Busse. Why and How to Benchmark XML Databases. *ACM SIGMOD Record*, 3(30):27–32, September 2001.

```
Q1) /site/regions/europe/item/mailbox
Q2) /site/regions/europe/item[shipping][payment](mailbox/mail/text ∪ description)
Q3) /site/regions/europe/(item/payment ∪ item[not(shipping)])
Q4) /site/(people/person[homepage] ∪ closed_auctions/closed_auction[annotation]/seller)
Q5) /site/( open_auctions/( open_auction[not(reserve)] ∪ open_auction[privacy])
          ∪ regions/europe/item/description[parlist/listitem])
```

Figure 9: Queries used to compare evalution strategies.

```
Q1)
I: /site/regions
O: /site/regions/europe/item/mailbox
C: /site/regions/europe/item/mailbox

Q2)
I: /site/regions/europe/item[location]/description/parlist/listitem[text]
O: /site/( regions/europe/item[description] ∪ open_auctions/open_auction[not(reserve)] )
C: /site/regions/europe/item[location][description]/description/parlist/listitem[text]

Q3)
I: /site/( regions/europe/item[mailbox/mail/from]/description/parlist[listitem/text]
         ∪ open_auctions/open_auction[privacy] )
O: /site/( regions/europe/item[description/parlist] ∪ open_auctions )
C: /site/( regions/europe/item[description/parlist][mailbox/mail/from]/description/parlist[listitem/text]
         ∪ open_auctions/open_auction[privacy] )

Q4)
I: /site/(regions/europe/(item[not(quantity)] ∪ item/location) ∪ people)
O: /site/(regions/europe/item ∪ people/person[homepage])
C: /site/(regions/europe/(item[not(quantity)] ∪ item/location) ∪ people/person[homepage])

Q5)
I: /site/( regions/europe/item/description[not(parlist)]
         ∪ closed_auctions/closed_auction[annotation]/seller )
O: /site/( regions/europe/item ∪ closed_auctions/closed_auction )
C: /site/( regions/europe/item/description[not(parlist)]
         ∪ closed_auctions/closed_auction[annotation]/seller )

Q6)
I: /site/regions/europe/item/description
O: /site/regions/europe/item/mailbox
C: empty query
```

Figure 10: Queries for sequential vs composition: inner (I), outer(O), composed (C).

XPath:

```
/a[p1]/b, /a[p1]/c[p2][p3]/d, /a[p1]/c[p2][p3]/e, /a[p1]/c[p2][p3]/f[p4]/g
```

XQuery:

```
for $x1 in /* return
  if ($x1[self::a[p1][b | c[p2][p3][d | e | f[p4][g]]]])
  then <a>
{  for $x2 in $x1/* return
    if ($x2[self::b]) then $x2
    else if ($x2[self::c[p2][p3][d | e | f[p4][g]]][not(self::b)])
    then <c>
{     for $x4 in $x2/* return
      if ($x4[self::d]) then $x4
      else if ($x4[self::e][not(self::d)])
      then $x4
      else if ($x4[self::f[p4][g]][not(self::d)][not(self::e)])
      then <f>
{       for $x7 in $x4/* return
        if ($x7[self::g]) then $x7 else ()
      } </f> else ()
    } </c> else ()
  } </a> else ()
```

XSLT:

```
<xsl:stylesheet>
  <xsl:template match="node()|@*">
  </xsl:template> <!-- rule for default behavior (skip) -->
  <xsl:template mode="regular" match="node()|@*">
  </xsl:template> <!-- rule for regular behavior (skip) -->
  <xsl:template mode="leaf" match="node()|@*">
    <xsl:copy>
      <xsl:apply-templates mode="leaf" select="@*"/>
      <xsl:apply-templates mode="leaf"/>
    </xsl:copy>
  </xsl:template> <!-- rule for leaf behavior (keep whatever is underneath) -->
  <xsl:template match="/">
      <xsl:apply-templates mode="regular" select="a[p1]"/>
  </xsl:template>
  <xsl:template mode="regular" match="a[p1][b | c[p2][p3][d | e | f[p4][g]]]">
    <xsl:copy>
      <xsl:apply-templates mode="regular" select="b | c[p2][p3][d | e | f[p4][g]]"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template mode="regular" match="b">
    <xsl:apply-templates mode="leaf" select="."/>
  </xsl:template>
  <xsl:template mode="regular" match="c[p2][p3][d | e | f[p4][g]]">
    <xsl:copy>
      <xsl:apply-templates mode="regular" select="d | e | f[p4][g]"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template mode="regular" match="d">
    <xsl:apply-templates mode="leaf" select="."/>
  </xsl:template>
  <xsl:template mode="regular" match="e">
    <xsl:apply-templates mode="leaf" select="."/>
  </xsl:template>
  <xsl:template mode="regular" match="f[p4][g]">
    <xsl:copy>
      <xsl:apply-templates mode="regular" select="g"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template mode="regular" match="g">
    <xsl:apply-templates mode="leaf" select="."/>
  </xsl:template>
</xsl:stylesheet>
```

Figure 11: Translations for XSQuirrel expression `/a[p1]/( b ∪ c[p2][p3]/(d ∪ e ∪ f[p4]/g) )`