RC23850 (W0601-117) January 27, 2006
Computer Science

IBM Research Report

Phantom XML

Kristoffer Rose, Lionel Villard
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Phantom XML

If you look too hard it isn't there!

Kristoffer **Rose**

Lionel **Villard**

Abstract

The original purpose of the W3C Extensible Markup Language (XML) was as a generic representation and interchange format for structured and semi-structured data. In practice this means that mappings exist for many data formats between the native form and an XML version. There are even standards for setting up such mappings, such as SQLX for mapping relational databases and the Global Grid Forum "Data Format Description Language" (DFDL) for mapping binary formats. More mappings will certainly arise and there might be a time when XML will be the long awaited data unification language...but there are still some rocks on the path. Indeed, XML suffers from a specific problem: the default data representation as character sequences is not well suited for processing, not even for processing by the XML standard languages. XPath 2.0, XSLT 2.0, and XQuery 1.0, part of the next generation of such languages, acknowledge this fact by having their behaviour specified in terms of an abstract version of XML, the data model, with a separate document describing the relationship between instances of this data model and actual "serialized" XML documents. An especially interesting case is when data is only accessed through queries in XPath. In this case the naive model does not work: it is overly expensive in space (and thus time) to parse or convert entire data structures to an in-memory XML data model instance and then run a generic XPath engine on the converted results.

We propose to make XPath access space efficient in general by the following strategy: *First*, virtualize the XML Data Model to allow creation of lightweight cursor-based adapters making various data structures appear as if they were XML without introducing unnecessary overhead. An example of this is to wrap access to the entire file system as a single virtual XML node corresponding to the "current file or directory" from which XPath navigation gives access to the whole filesystem. *Second*, catalog useful profiles of restricted Data Model instances for which such access is as efficient as directly accessing the underlying data structure. Examples, explained in the paper, are streaming, streaming with back-pointers, linear, and random access. *Third*, describe an analysis of XPath queries that determines the profile requirements for that query (for a specific XPath evaluator). This analysis can determine, for example, that simple "downwards" XPath queries can be executed within the constraints of the streaming profile. *Finally*, provide a cache wrapper that allows use of a data source with a restricted profile in a context requiring a more complete profile, for example, materializing all actually visited nodes in an internal tree permits use of the random access profile over streaming data sources.

In this paper, we describe each of these components in detail and show through examples how they interact in a running XML processing system. We call the approach for phantom XML to highlight the fact that the combination of direct adapters and an optional cache implies that no traditional XML is stored in the system. We measure actual efficiency (in space and time) of running some realistic example XPath queries against, both real XML documents and virtual XML data, using our implementation. This demonstrates, for example, that streaming XPaths runs in constant space at competitive speeds, and that XPaths with locally bounded predicates run in space unrelated to the overall data size.

Table of Contents

1. Introduction

1.1. Background

The Extensible Markup Language, [XML], was designed as a generic representation and interchange format. But XML suffers from a problem: the default data representation as character sequences is not really well suited for processing, not even for processing by the XML standard languages such as [XPath] and [XSLT]. This is adressed in the next generation of XML processing languages, [XPath2], [XSLT2], and [XQuery], by defining the languages in terms of a well-defined *data model* [DM], and extracting the relationship to traditional textual XML documents.

At the same time, the value of XML is greatly enhanced by the XML Schema structure description language [XS]. Indeed the ubiquity of XML has lead to the use of XML Schema as one of the primary structure description formats for specification of services of all kinds, notably through the prime data declaration component of the Web Services standards [WSDL] that are now being used widely in service oriented architectures (SOA). To properly deal with existing data this has lead the creation of mappings for many structured formats between the native form and an XML version described with XML Schema. There are even standards for setting up such mappings such as SQL/XML [SQLX] for relational data and the emerging Data Format Description Language [DFDL] for binary and textual formats.

1.2. Idea

As a consequence of the above it is becoming commonplace to convert from all sorts of formats into XML and then do XML processing. An especially common case is when data is converted to XML only to be accessed with a single XPath expression (or a few). This happens, for example, when processing converted structures with XML languages that exploit XPath such as [BPEL] as well as for writing simple "filter" applications using XPath-based APIs such as [SDO].

However, the naive model does not work very well in this case: it is overly expensive in both time and space to first convert structures into textual XML documents, then parse the textual XML into an in-memory internal data model instance, and finally extract the selected nodes as specified by the XPath (most XPath implementations work off an in-memory representation of the XML data model instance). What is needed is a good way to avoid *any* unnecessary materialization. We suggest doing this by

- analysing the XPath(s) and determine a strategy for accessing just the nodes of the data model instance that are needed, and
- write a *pull* wrapper that extracts just those parts of the data that are needed to *pose* as the needed data model nodes.

The good thing about this separation is that the first step is completely generic, depending only on the semantics of the XPath language [FS] and the data model [DM], whereas the second step is specific to the data format.

In practice we will achieve this in four steps:

1. Exploit the notion of *cursor* [SEQUEL2] as the central notion in a new lightweight interface to the XML data model that allows the creation of lightweight *phantom XML* adapters that make arbitrary data structures appear as if they were XML without introducing unnecessary overhead. An example of this is to wrap access to a file system as a cursor into a single virtual XML document corresponding to the "current file or directory" from which XPath navigation gives access to the whole filesystem.

2. Catalog useful *profiles* of features allowed by restricted Data Model instances for which such access is as efficient as directly accessing the underlying data structure. Examples, explained below, are streaming, streaming with back-pointers, linear, and random access.

3. Describe an analysis of XPath queries that determines the *profile requirements* for that query (for a specific XPath evaluator). This analysis can determine, for example, that simple "downwards" XPath queries can be executed within the constraints of the streaming profile.

4. Provide wrappers that allows use of a data source with a restricted profile in a context requiring a more complete profile, for example, a cache that materializes all actually visited nodes in an internal tree permits use of the random access profile over streaming data sources.

1.3. Our approach in an example

We will illustrate this approach through an example. Our task will be to extract the top-level numbered headings from an OpenOffice [OOo] document.

An OpenOffice document (with extension `.sxw`) is really a [Zip] archive. Example 1, “OpenOffice Document Archive Structure” shows the archive members of a simple document, "example.sxw".

Example 1. OpenOffice Document Archive Structure

```
Archive:  example.sxw
  Length     Date   Time    Name
 --------    ----   ----    ----
       30  09-15-05 20:50   mimetype
     7966  09-15-05 20:50   content.xml
    14181  09-15-05 20:50   styles.xml
     1066  09-15-05 20:50   meta.xml
     8363  09-15-05 20:50   settings.xml
      752  09-15-05 20:50   META-INF/manifest.xml
 --------                   -------
    32358                   6 files
```

The text in the document is in the "content.xml" member, in XML following an OpenOffice-specific DTD. Example 2, “OpenOffice Document Content XML Structure” shows the "content.xml" component of our sample document (except it has been formatted and lots of style information has been replaced by "..." so the example fits).

Example 2. OpenOffice Document Content XML Structure

```
<?xml version="1.0"?>
<!DOCTYPE office:document-content
   PUBLIC "-//OpenOffice.org//DTD OfficeDocument 1.0//EN" "office.dtd">
<office:document-content xmlns:office="http://openoffice.org/2000/office" ...>

  ...
  <office:body>
    ...
    <text:ordered-list text:style-name="Numbering 1">
      <text:list-item>
        <text:h text:style-name="P2" text:level="1">...
          Introduction
        </text:h>
      </text:list-item>
    </text:ordered-list>
    <text:p text:style-name="Text body">
      <text:span text:style-name="T1">
XML and XSLT show great promise for bridging the gap between designers
and implementers. Design and build tasks can occur
simultaneously. Stylesheets can even be written by technically savvy
designers and used in the website without
reverse-engineering. However, tools are lacking.  Back-end data is
usually not delivered as XML and front-end tools for creating and
testing XSLT style sheets are still in the prototype stages.
      </text:span>
    </text:p>
    <text:ordered-list text:style-name="Numbering 1"
                      text:continue-numbering="true">
      <text:list-item>
        <text:h text:style-name="Heading 1" text:level="1">
          Conclusion and perspectives
        </text:h>
      </text:list-item>
    </text:ordered-list>
    <text:p text:style-name="Text body">
As a result of this analysis we can conclude that this solution is perfect.
    </text:p>
  </office:body>
</office:document-content>
```

Notice how headings are identified by a "text:h" element with a "text:style-name" attribute value of "Heading 1". We can express this with an XPath expression over the "content.xml" member:

```
//text:h[@text:style-name = 'Numbering 1']/text()
```

But to actually get to this step the normal procedure would be to first unzip the archive and then run the XPath expression on the "content.xml" member.

In this paper we will instead take the "virtual XML" approach and introduce two additional XPath functions in our arsenal:

- `unzip`(*filename*) returns an XML document where the zip file structure is represented in XML. Example 3, “Unzip Virtual XML Extension” shows the result of using the `unzip` function on our sample zip archive (except that we have formatted the XML and replaced the longer byte sequences with "...").

- `parse`(*base64data*) returns the XML content obtained by parsing the binary data as an XML file.

Example 3. Unzip Virtual XML Extension

```
<?xml version="1.0"?>
<zip>
  <entry name="mimetype" size="30" time="2005-09-258T20:50:12Z">
    <bytes>YXBwbGljYXRpb24vdm5kLnN1bi54bWwud3JpdGVy</bytes>
  </entry>
  <entry name="content.xml" size="7966" time="2005-09-258T20:50:12Z">
    <bytes>...</bytes>
  </entry>
  <entry name="styles.xml" size="14181" time="2005-09-258T20:50:12Z">
    <bytes>...</bytes>
  </entry>
  <entry name="meta.xml" size="1066" time="2005-09-258T20:50:12Z">
    <bytes>...</bytes>
  </entry>
  <entry name="settings.xml" size="8363" time="2005-09-258T20:50:12Z">
    <bytes>...</bytes>
  </entry>
 <entry name="META-INF/manifest.xml" size="752" time="2005-09-258T20:50:12Z">

    <bytes>...</bytes>
  </entry>
</zip>
```

Combining the two functions with the expression above means that the expression in Example 2, “OpenOffice Document Content XML Structure” selects the top-level headings in the OpenOffice document: evaluating it returns the single text value "Conclusion and perspectives" (assuming that the namespace prefixes have been properly set up).

Example 4. XPath to select OpenOffice top-level headings

```
parse(decode(
    unzip('example.sxw')/zip/entry[@name = "content.xml"]/bytes/text(),
    'UTF-8'))//text:h[@text:style-name = 'Numbering 1']/text()
```

The challenge of *phantom XML* is to evaluate the above expression without ever materializing the actual XML document. Informally we can imagine doing it like this with lazy evaluation:

1. Evaluation starts by the XPath engine realizing that it needs to produce some output as the result of the whole query is being printed. This invokes the parse function. The parse function immediately returns ready for XML navigation of the (not yet) parsed byte stream parameter.

2. The XPath engine starts navigationg the XML document build by the parse function, searching for a "text:h" element. This makes the parse function need to look at its argument byte sequence. This invokes the unzip function which allows navigation to the "zip" element which can be done without actually reading the zip file (because it is always present).

3. Still out to produce bytes for the parse function, the XPath engine next searches for an "entry" child of the "zip" element. This makes the unzip function actually open the zip file and visit the archive members. For each the XPath engine requests the value of the "name" attribute until one with the value "content.xml" is found.

4. For the "content.xml" entry the XPath engine then requests the bytes element and immediately thereafter the text contents. This returns a handle to the zip entry byte stream which then enables the parse function to start parsing bytes.

5. The XPath engine now starts navigating the result of the parse function, with the parse function producing the relevant XML nodes on the fly as it parses. Every time parsing has produced a "text:h" element then the XPath engine attempts to navigate to the "text:style-name" attribute to check that its value is the text "Numbering 1". When one is found the text() value is extracted and contributed to the resulting node sequence.

Below we shall explain how the above is achieved in practice.

1.4. Organization of the paper

This paper is organized as follows. In Section 2, “Lightweight Virtualization of the XML Data Model” we present the adaptable and cursor-based data model implementation that represents our "phantom" XML virtualization, along with a set of profiles corresponding to common data access patterns. In Section 3 we show how this can be combined with an XPath analysis to allow adapting data source profiles to XPath requirements. In Section 4 we present some measurements and in Section 5 we conclude.

2. Lightweight Virtualization of the XML Data Model

In this section we explain the data model and in particular how the iterator/cursor technique can be used to present a lightweight XML data model interface.

We shall adhere to the following principles to make the focus model as light-weight as possible:

- A single data structure (or object) should suffice for all access into an XML document as well as any sequence of nodes from a collection of XML documents.

- All access to the data model properties is through the focus: It is not possible to dereference the cursor and obtain a reference to the "real" XML data underneath the focus.

We shall call this universal cursor the *focus*.

2.1. Focus operations

The *operations* of a focus are described in the following table. For each the corresponding data model [DM] concept is listed, where applicable.

Operation	Explanation	Data Model concept	Notes
get-node-kind	get node kind of current item	dm:node-kind	
get-node-name	get name of element, attribute, or processing-instruction	dm:node-name	
get-value	get simply typed value of current item, if any	dm:string-value, dm:typed-value	1
get-type	get type of item (falls back to kind)	dm:typed-value	1

Operation	Explanation	Data Model concept	Notes
get-type-name	get type name of current item, if any	dm:typed-value	1
get-namespace-bindings	get list of declared namespace prefixes (including the empty default) with namespace URIs	dm:namespace-bindings	
get-base-uri	get base URI	dm:base-uri	
is-id	test whether the current item is an ID	dm:is-id	
is-idref	test whether the current item is an IDREF	dm:is-idref	
get-size	get context size		2
get-position	get context position		2
to-next	navigate to next item in context sequence *or* fail		2
to-previous	navigate to previous item in context sequence *or* fail		2
to-position	navigate to item with specific position in context sequence *or* fail		2
is-same-document	test whether two foci have current nodes in the same document		
is-same-node	test whether two foci have the same current node	node identity	
is-document-order-comparable	test the document order of the current node of two foci	document order	
to-attributes	navigate to attributes (replace the context sequence with first attribute in document order as current) *or* fail	dm:attributes	2
to-children	navigate to children (replace the context sequence with first child as current) *or* fail	dm:children	2
to-parent	navigate to the parent (with no other nodes in the context sequence) *or* fail	dm:parent	
to-root	navigate to the root (with no other nodes in the context sequence) *or* fail		3
duplicate	duplicate (clone) the focus		4
free	free the focus		4
set-value	set the simply typed value of current item		5
add-attribute	add attribute to element		5

Operation	Explanation	Data Model concept	Notes
remove-attribute	remove attribute from element		5
add-first-child-*kind*	add new first child node of *kind* (text, element w/name, comment, or processing-instruction w/target)		5
add-following-sibling-*kind*	add new following sibling node of *kind* (text, element w/name, comment, or processing-instruction w/target)		5
remove-first-child-subtree	remove entire subtree rooted at first child		5
remove-following-siling-subtree	remove entire subtree rooted at following sibling		5

Notes.

1. The data model accessor dm:typed-value and dm:string-value are simplified in the focus model:

   - get-value only returns *unchanged* simply typed values: it always succeeds when the focus points to text nodes but the result may be typed (if the text is contained in a simply typed element); in addition it always succeeds for attribute nodes (with the typed result), comment, and processing-instruction nodes; finally for non-node items in the top-level sequence it just returns the typed value.

   - get-type just returns a type that would type-match the current item of the focus.

2. Context sequence navigation is not just used for the top-level context sequence, it is also used for the sequence of attributes and children established by those navigation operations.

3. Navigation to the root is not part of the data model.

4. Duplicate is used to clone the focus into two identical but independent foci: navigation on either focus does not affect the other. The two still reference the same underlying document, however, so mutation performed by one will affect the document seen by the other. Each of the two clones should be free'd independently.

5. The update, or *mutation* operations, all operate *near* but not *on* the current node. They all operate on the XML data model *tree* so we do not actually have a way to modify the top-level sequence.

As we shall see in the remainder of this section, many use patterns can be expressed in terms of pure subsets of these operations. Two cases are so common and restrictive that they deserve special mention.

2.2. Streaming input

The most common pattern is to read an XML document in a single, left-to-right pass. This profile is ideal for applications such as content-based routing or searching through large (or even infinte) streaming XML data feeds (for instance a stock quote stream). This corresponds closely to the operations of XML "Pull Parser" APIS [XPP][StAX], indeed it is trivial to write adapters for "real" XML using these very efficient parsers when restricting access to the streaming input profile. (We shall discuss the details of how we classify concrete XPath expressions as streaming, or transform XPaths into streaming form, in Section 3.)

For our operation set, the streaming input use pattern occurs when the sequence of operations conforms to this grammar (with non-terminals in italics):

Document ::= to-children *Sibling* free
Sibling ::= get-node-name? (get-value | *Attributes Children*)
Attributes ::= duplicate to-attributes (*Attribute-traversal*)? free
Attribute-traversal ::= (get-node-name? get-value? to-next)+
Children ::= duplicate to-children (*Sibling-traversal*)? free
Sibling-traversal ::= (*Sibling* to-next)+

with the additional constraints that

- At any time only the result of the last duplication not freed can be operated on, corresponding to a LIFO or *duplicate stack discipline*.
- For each optional fragment "(...)?" the immediately *preceding* "to-" operation determines whether it is used: on failure the optional fragment is skipped, otherwise it is included.
- For each repeated fragment "(...)+" the *trailing* to-next operation determines whether another repetition is unfolded: on failure no further repetition, otherwise one more is attempted.

Indeed we can seperate this into two constraints: *duplicate-stack-discipline*, corresponding to the first constraint, and *streaming-siblings*, corresponding to the grammar with only the second constraint.

Notice that we could not have used a single focus combined with using the to-parent operation because to-parent does not recover the context sequence with the full list of children of the parent but rather effectively "forgets" the other siblings, and we have no "to-following-sibling" navigation operation. In practice this is alright because the combination of the constraints means that the "duplicate" and "free" operations match up cleanly so only one focus is active at any time.

Another characteristic of the streaming input profile is because of its simplicity, it is very easy and very fast to implement new adapters. These adapter can then be used in a streaming fashion by any applications built on top of virtual XML without any specific requirements.

2.3. Streaming output

Conversely, streaming output corresponds to "push" streaming such as the events of [SAX] except that the streaming output profile emits attributes one at the time rather than as a single event. The streaming output profile is useful for serializers and similar "sink" applications that emit XML without rereading the emitted XML.

Streaming output can also be realized by restricting the sequence of allowed operations to occur only as specified by a grammar:

Document ::= add-first-child-element *Content* free
Content ::= (add-attribute)* (*First* to-children (*Following* to-next)* to-parent)?
First ::= add-first-child-text
| add-first-child-comment
| add-first-child-processing-instruction
| add-first-child-element *Content*
Following ::= add-following-sibling-text
| add-following-sibling-comment
| add-following-sibling-processing-instruction
| add-following-sibling-element *Content*

As an example, the result of the entire OpenOffice heading query of Example 4, “XPath to select OpenOffice top-level headings” is meant to be copied to the serializer thus it only needs to implement the streaming profile.

Example 5. Focus Copy Method

One simple application we can now write is a *copy* operation that copies a streaming input to a streaming output; here in Java. Notice how this code directly translates the streaming input operations to streaming ouput operations without using any storage. (For convenience the "first-child" and "following-sibling" variants are joined in the Java implementation using a "where" parameter to distinguish between the operations.)

```
private static void copy(short where, Focus source, Focus target)
{
  switch (source.getKind())
  {
    case Focus.DOCUMENT_KIND :
      copyChildren(where, source, target);
      break;

    case Focus.ELEMENT_KIND : {
      target.addElement(where, source.getName());
      Focus attr = source.duplicate();
      if (attr.toAttributes(null)) {
        do {
          copy(where, attr, target);
        } while (attr.toNext());
      }
      attr.free();
      target.toChildren(null);
      copychildren(where, source, target);
      target.toParent();
      break;
    }

    case Focus.ATTRIBUTE_KIND :
      target.addAttribute(source.getName(), source.getValue());
      break;

    case Focus.TEXT_KIND :
      target.addText(where, source.getValue());
      break;

    case Focus.COMMENT_KIND :
      target.addComment(where, source.getValue());
      break;

    case Focus.PROCESSING_INSTRUCTION_KIND :
       target.addProcessingInstruction(where, source.getName(),
                                               source.getValue());
       break;
  }
}

private static void copyChildren(short where, Focus source, Focus target)
{
  source = source.duplicate();
```

```
  if (source.toChildren()) {
    copy(Focus.WHERE_FIRST_CHILD, source, target);
    while (source.toNext())
      copy(Focus.WHERE_FOLLOWING_SIBLING, source, target);
  }
  source.free();
}
```

2.4. Forward only with skipping profile

The next profile up the "food chain" of profiles is for data where we allow arbitrary skipping of subtrees as we read the document from left to right: this is a useful abstraction for parsed data with size information for the data fragments corresponding to subtrees, or where children at each level are indexed such as, for example, when accessing file systems (where skipping subtrees is cheap).

Both of the result of the unzip and parse functions from the OpenOffice headings XPath of Example 4, “XPath to select OpenOffice top-level headings” can be evaluated with just the features duplicate, to-children, and to-next, to visit child elements, as well as to-attributes, to visit and test the attributes. This fits within the forward-only profile but not the streaming profile because the use of predicates means that the XPath engine skips the subtrees for which the predicate was false.

2.5. Downward only profile

An interesting profile for XPath is the profile that disallows upward navigation, i.e., to-parent and to-root: with this profile *only* states "saved" with duplicate enable referencing ancestor nodes.

The advantage of this profile is that only a "window" on the XML data is available at any time during processing. This window is delimited by the first focus, the one nearer to the beginning of the tree than the other foci, and the last focus which is ahead of the others foci. When the first focus moves forward, then all previous nodes can be discarded. When the last focus moves forward then nodes are cached in order to be made available for the later foci.

When constrained by the *duplicate stack discipline* this profile operates in the same manner as the streaming input profile: no nodes are cached which implies partial streaming processing (the only difference being that siblings can be accessed in any order).

2.6. Full data model profile

Everything is allowed. This typically implies that the entire XML document needs to be stored in memory. The purpose of this paper is to avoid the full data model profile whenever possible.

2.7. Fixing missing profile features

So what should one do when an application is provided with a source that just supports the streaming input yet needs to do arbitrary navigation? We shall wrap the limited data source in a "fixer" focus that provides the missing features.

Here are some basic fixers:

1. *Compare* fixes the node comparison operations by ensuring that nodes are tagged with an id that can be used to determine document order.

2. *Size* implements the context size.

3. *Skip* fix to allow arbitrary skipping.

4. *Cache* fixes all read features.

The next level of set of features implies caching parts or the whole data model instance. In the XML world, the natural structure to cache XML data is to use a DOM-like representation. However in order to avoid the eager materialization of the entire XML data, a cache manager is associated to it in order to control the internal representation (figure?). For instance, it can decide to materialize lazily only certain parts of the XML data, to skip other parts and even to discard existing materialized XML data.

Three profiles are identified with different levels of caching. In the first subsection we describe the first profile which allows only forward navigation. The second section extends the first profile with the subtree skipping capabilities. All features are allowed in the full profile describe the the third section.

2.8. Profile and fixer summary

The table below summarizes some profiles (supported features marked "*", constrained features marked "-") and fixers (that provide features marked "*" when the underlying profile supports features marked "o").

Allowed operation	**streaming**	**forward**	**down**	**read-only**	**streaming-out**	**read-write**		**compare**	**size**	**skip**	**cache**
duplicate-stack-discipline	*	*	*	*		*		o	o	o	o
streaming-siblings	*	*	*	*		*		o	o	o	o
get-size			*	*		*			*		*
to-next	-	*	*	*	-	*				*	*
to-previous			*	*		*					*
to-position			*	*		*					*
is-same-document			*	*		*		*			*
is-document-order-comparable			*	*		*		*			*
to-attributes	-	*	*	*	-	*				*	*
to-children	-	*	*	*	-	*				*	*
to-parent				*	-	*					*
to-root				*		*					*
duplicate	-	*	*	*		*				*	

Allowed operation	**streaming**	**forward**	**down**	**read-only**	**streaming-out**	**read-write**		**compare**	**size**	**skip**	**cache**
free	-	*	*	*		*				*	*
streaming-output				*	*	*					
set-value					-	*					
add-attribute					-	*					
remove-attribute						*					
add-element					-	*					
add-text					-	*					
add-comment					-	*					
add-processing-instruction					-	*					
remove-subtree						*					

The "pseudo-operations" defined by special rules are italicized, in particular the "first-child" and "following-sibling" variants have been combined and most of the "get-" operations have been omitted as they are always supported.

3. Optimizing XPath for Data Model Profiles

In the previous section we identify profiles requiring different levels of resources consumption. The basic streaming input profile allows very few features, and therefore supports only a subset of XPath. On the other hand it does not store any XML data in memory. At the other side of the spectrum, the full profile allows the processing of full XPath but with the price of having potentially all XML data in memory. In this section we explore rewriting techniques that change the category of an XPath, for instance from a forward only XPath to a basic XPath. By changing the category, an XPath which wasn't scalable can now be processed over large XML data.

3.1. XPath processor overview

Each XPath expression is processed by first compiling it into an interpretable form or a directly executable form. It is then interpreted or executed. We describe those two phases in the following sections.

3.1.1. Static phase

The compilation phase consists of several rewriting steps on the original XPath, in the following order:

- The first step is parsing the expression within the static context given by the hosting application (i.e. Java, XSLT, etc...). The result of this first step is an AST representing the parsed expression in memory.

- The second step consists of normalizing the initial expression into Core XQuery [FS]. Since the resulting "core expression" can be large, it is sent over a stream of expression events instead of being fully built in-memory. It is then reduced on-the-fly by various optimization processes, described below.

- The following steps perform on-the-fly reductions and optimizations. It includes the removal of dead code, unused variables and removal of distinct-doc-order. After those steps, the core expression is fully represented in-memory as an AST.

- Global analysis and optimizations are then performed such as function call analysis, specialization, etc... One particular analysis which is in special interest in this context is the feature set analysis. It determines which features are needed to execute the XPath.

- The last step consists of compiling the optimized-reduced core expression into an interpretable form or directly into an executable form (i.e compiled C code, Java bytecode, etc...)

Designing a full highly optimizing compiler is a tremedous task which requires a very sophisticated cost model for all the components involved in the processing of XPath expression, from the system to the application. In order to limit the scope of this paper, a very simple cost model is defined with the following assumptions:

- Accessing the data is relativelly costly since data is accessed through the virtualization layer.

- Input document can be very large, even infinite (i.e. events stream).

During the compilation phase, the expression is transformed in such a way that the number of accesses to the source data and the number of intermedite buffers is minimized. The best case is when the data that is needed for processing is only accessed once. The worst case is when the input data must be materialized fully. (Section 3.2, “Rewrite XPath to constrain used features” described some of those transformations.)

Example 6, “doc('content.xml')//text:p[string(.) = 'XPath'] after compilation” shows an example result of compilation, more specifically normalization and optimization. The features set analysis can determine that for this expression the downward profile is required. Indeed, the variable `fs:sequence` bound to the expression `descendant-self::node()` is referenced twice and therefore its value is potentially read twice. Therefore it needs to be cached which in this case correspond to the entire input document. Later in the paper we will see how this expression can be executed within the streaming profile only.

Example 6. doc('content.xml')//text:p[string(.) = 'XPath'] after compilation

```
fs:distinct-doc-order(
  let $fs:sequence :=
    fn:doc(fs:convert-simple-operand(fn:data('content.xml'), ' '))
  return
    let $fs:last := fn:count($fs:sequence)
    return
      for $fs:dot in $fs:sequence return
        fs:distinct-doc-order(
          let $fs:sequence := descendant-self::node() return
            let $fs:last := fn:count($fs:sequence) return
              for $fs:dot in $fs:sequence return
                let $fs:sequence := fs:distinct-doc-order(child::p) return
                  let $fs:last := fn:count($fs:sequence) return
                    for $fs:dot at $fs:position in $fs:sequence return
                      if (typeswitch(fs:eq(
                            fs:convert-operand(fn:data(fn:string($fs:dot)),
                                               'string'),
                            fs:convert-operand(fn:data('XPath'), 'string')))
                          case $fs:v1 as fs:numeric return
                            op:numeric-equal($fs:v1, $fs:position)
                          default $fs:v1 return
                            fn:boolean($fs:v1))
                      then
                        $fs:dot
                      else ()
        )
)
```

3.1.2. Runtime phase

The runtime phase follows strictly the steps described in [XPath2], Section 2.2.3.2 Dynamic Evaluation Phase. Our XPath processor is pull-based with the following characteristics. First, it is *lazy* which means that the expression computation expression is perfomed incrementally each time the application (the one requesting XPath evaluations) asks for the next result, as discussed before. In particular the input document is parsed incrementally. The second characteristic is that the profile of each source involved in the expression computation is checked and "fixed" if needed (see Section 2.7, “Fixing missing profile features”). The fixing step depends on the features needed by both the host application and the XPath engine. For example, for an XPath requiring only streaming profile, no fixing will be performed. However if the application navigates through the XPath result downwards or upwards, then each source needs to be fixed in order to allow it.

3.2. Rewrite XPath to constrain used features

In this section is described some rewriting techniques that decrease the number of features needed for evaluating an XPath.

3.2.1. Forward-only transformation

The forward-only transformation aims at converting arbitrary XPath expressions into an equivalent expressions using only forward axes (`child`, `descendant`, etc...). (The details of this kind of transformation is well described in the literature, see for example [Geneves], [Xaos], and [Olteanu]).

One characteristic of this transformation is to change the number of features required for running an XPath, eventually transforming it into a pure streaming (only one cursor needed) XPath. For example, Consider the XPath expression `/descendant::employee/ancestor::manager[1]` which enumerates all employee elements and then collects for each the closest manager ancestor element. Example 7, "/descendant::employee/ancestor::manager[1] after normalization and forward-only transformation" shows the result of normalization [FS] and the forward-only transformation steps on the expression.

Example 7. /descendant::employee/ancestor::manager[1] after normalization and forward-only transformation

```
ddo(
  let $managers := /descendant-or-self::manager
  return
    let $seq := /descendant::employee
    return
      for $dot in $seq
      return
        let $seq :=
          for $m in $managers
          return
            if $d/descendant-or-self::node() intersect $dot
            then $d else ()
        return
          let $last := count($seq)
          return
            for $dot in $seq
            return
              let $rpos := count(
                  for $m in $managers
                  return
                    if $d/descendant-or-self::node() intersect $dot
                    then $d else ()
                 )
              return
                let $pos := $last - $rpos + 1
                return
                  if $pos eq 1 then $dot else ()
)
```

In this transformed expression, instead of using the ancestor axis, it searches all potential ancestors (by using the `descendant-or-self` axis) and for each it tests that it has the context node as a descendant (`$d/descendant-or-self::node() intersect $dot`). The bottom part of the expression (when the variable `last` is bound) performs the position filtering.

Notice that this expression is not purely streamable and that several intermediary buffers (represented as variables) are used. In this case, only the `manager` elements which have a `employee` as an ancestor need to be stored which is potentially much less than the entire document.

3.2.2. distinct-doc-order() function removal

During the process of normalization, document order and duplicate removal is explicitly performed by the formal semantic function called `distinct-doc-order`. This function hurts the streamability of XPath expression in a very bad way. Indeed, it required the storage of the expression result passed as a parameter of this function. Hopefully, in some cases it is possible to determine statically whether this function is useful or not [Fernandez]. For example, let us consider the expression `$log/*/descendant::content-len`. The expression obtained after the normalization step is the following:

```
distinct-doc-order(
  for $fs:dot in distinct-doc-order(
                        for $fs:dot in $log return child::*
                      )
  return descendant::content-len
)
```

Each step is evaluated with respect to an implicit context node, which is bound to the variable `$fs:dot`. The `distinct-doc-order` function sorts its input in document order and removes duplicates. However in this example the result of the inner for loop is already in document order and contains no duplicates. That is a property of all axis, in particular the `child` and the `descendant` axis. Therefore, the `distinct-doc-order` function can be removed, resulting in the following expression:

```
for $fs:dot in
    for $fs:dot in $log return child::*
return
  descendant::content-len
```

This expression is now streamable and can be evaluated within the only-one cursor profile.

3.2.3. Schema-based rewriting

Whenever the schema of sources is known while compiling XPath expression, several schema-based optimizations can be performed. It also help reducing the number of features required to run XPath expression. For example, consider the operator `|`. In addition to constructing the union of two sequences, it eliminates duplicate nodes and it returns node in document order, which requires at least storing the entire evaluation result before being executed. Consider the expression `//(africa|europe)//@id` which looks for all the id attribute within africa and europe. If the schema associated with the input document specifies that `africa` and `europe` occur only once and that `africa` always occurs before `europe` then the union operator can be transformed into an equivalent function, say `simple-union()`, that does not perform duplicate elimination and reordering.

3.2.4. Synthesis

By applying the optimizations described in the section above, in addition of some others simple optimizations based on type analysis, contextual analysis and cardinality analysis, the expression `//text:p[string(.) = 'XPath']` is reduced as follows:

```
    let $fs:sequence := fn:doc('content.xml')
    return
        for $fs:dot in $fs:sequence
        return
            for $fs:dot in descendant-self::node()
            return
                for $fs:dot in child::p
                return
                    if (fs:eq(fn:string($fs:dot), 'XPath')
                    then $fs:dot
                    else ()
```

Compare to the normalized expression, this expression is much simpler and requires much fewer features to be executed. In fact, this expression can be executed within the streaming profile as it contains only forward axis, each variable declaration is referenced only once, and the functions involved are themselves streamable.

4. Experimentation and measurements

We have implemented a prototype based on the ideas presented in this paper [Virtual]. This prototype is written in Java and covers most of the XPath 2.0 language. Several virtual XML adapters have also been implemented, including the ones described in the paper, `unzip` and `parse`, plus a generic Data Format Description Language [DFDL] adapter, a generic relational database adapter, as well as adapters for entire file systems.

The prototype has allowed us to performed some very preliminary measurements. Our main goal is to show that for some expressions it uses a constant amount of memory and that in this context it is fast. The type of expressions we have focused on so far are those that can be evaluated using the streaming input profile, like this one (inspired by Example 4, “XPath to select OpenOffice top-level headings”):

```
v:parse(v:decode(
 v:unzip('examples.sxw')/zip/entry[@name eq 'content.xml']/bytes/text(),
 'UTF-8'))/*/body[1]/h[3]/text()
```

We have run this expression against different input sizes using our prototype and also [Xalan] 2.6.1. In order to run this expression using Xalan, the two functions `unzip` and `parse` have been coded directly in Java and the input stream associated to the file `content.xml` is given as the input of Xalan's XPath engine. The results are shown in Figure 1, “Performance and scalability measurements”. The number of entries in the zip file remains constants through the various tests, only the size of `content.xml` varies. On the left, the part of the document that is after the third header varies in size, whereas and the right the part before varies in size.

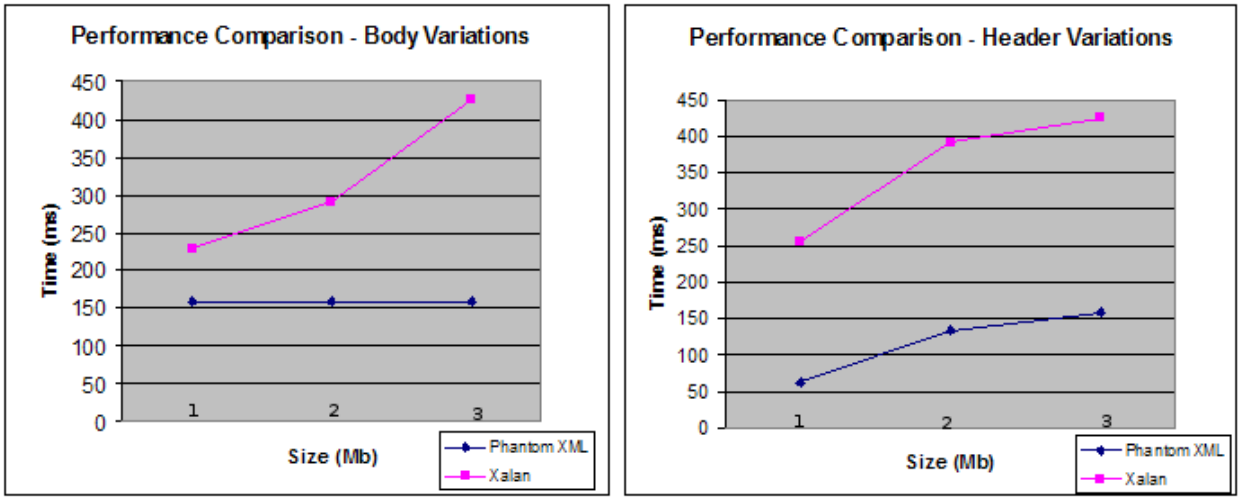

Figure 1. Performance and scalability measurements

In general, our prototype performs better than Xalan for two main reasons: our XPath engine reads input documents lazily and for this specific expression it does not have to read the entire document before returning the third header. The left curve corresponding to Phantom XML is flat and therefore illustrates this behavior. The second reason is the input document is not materialized in memory since the expression can be evaluated within the streaming input profile. Xalan, on the other hand, fully reads input documents and builds an in-memory representation of it.

5. Conclusions and perspectives

We have shown how the use of "XML data model profiles", with a subset of features of the full data model, can capture data access patterns, and how a cursor model using such profiles is can exploit results of an analysis of data access patterns exercised by XPaths queries to obtain a scalable XPath processor allowing the processing of large XML efficiently by caching in memory only the nodes that are required for evaluating a given query. Moreover, we have shown how XPath queries can be highly optimized using cutting-edge optimization techniques (such as distinct-doc-order reduction) further improving performance and we have explained how direct (or "on-demand") adapters can create virtual XML views on non-XML data without requiring any materialization of actual XML structures. This is what we call "phantom" XML. Finally, we have outlined a comparison of the performance of a Phantom XML implementation and a traditional one, giving an example where Phantom XML does not suffer the same scalability issues as the traditional solution.

5.1. Next steps

Experiments ave started with the virtual and phantom XML notions using our released prototype [Virtual]. Especially the combination of virtual XML adapters shows a lot of promise in streamlining the process of eliminating the data conversion costs of the on-demand enterprise. Further adapters in development include adapters for the entire web and multimedia files. All steps that will facilitate the adoption of XML and the XML processing languages without incurring an unreasonable performance penalty.

5.2. Related Work

One particular case that has been studied extensively is *streaming*. Most streaming XPath processors support some subset of Xpath suitable for streaming [STX][XmlReader] but recently some more full-featured variants have been reported [Xaos][XSQ][BEA]. Common to these approaches is that they employ an automata approach where the XPath

is translated into some kind of state machine that is then run over the "event stream" of the input. This is very fast for some XPaths but can lead to exponential blow-up of the state space, for example when finding identically named pairs of nodes throughout a document; partial caching can improve on this and indeed hybrid engines use this [Virtual][BEA]. One engine goes further and define explicit event processing extensions to XQuery [FluXQuery]. It is even possible to use clever data structures for automatic caching [ViteX].

Bibliography

[BEA] Daniela Florescu, Chris Hillery, Donald Kossmann, Paul Lucas, Fabio Riccardi, Till Westmann, J. Carey, and Arvind Sundararajan *The BEA streaming XQuery processor*. The VLDB Journal - The International Journal on Very Large Data Bases, 13, 3, September 2004.

[BPEL] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, SatishThatte, Ivana Trickovic, and Sanjiva Weerawarana *Business Process Execution Language for Web Services version 1.1*. OASIS, May 2003. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.

[Geneves] Pierre Genevès and Kristoffer Rose *Compiling XPath into a State-less Forward-only Subset* First International Workshop on High Performance XML Processing. May 2004. http://wam.inrialpes.fr/www-workshop2004/Program.html.

[DFDL] MikeBeckerle. *Data Format Description Language (DFDL)*. Global Grid Forum Data Format Description Working Group draft, September 2005. https://forge.gridforum.org/projects/dfdl-wg/.

[DM] MaryFernandez, AshokMalhotra, JonathanMarsh, MartonNagy, and NormanWalsh *XQuery 1.0 and XPath 2.0 Data Model*. World Wide Web Consortium Candidate Recommendation, November 2005. http://www.w3.org/TR/xpath-datamodel

[Fernandez] Mary Fernandez, Jan Hidders, Philippe Michiels, Jerome Simeon, and Roel VercammenFernandez et al *Optimizing Sorting and Duplicate Elimination in XQuery Path Expressions* 16th International Conference on Database and Expert Systems Applications Copenhagen, Denmark, August 2005.

[FluXQuery] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier *Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams*. International Conference on Very Large Data Bases (VLDB), Toronto, Canada, 228-239, September 2004.

[FS] DeniseDraper, PeterFankhauser, MaryFernandez, AshokMalhotra, KristofferRose, MichaelRys, JeromeSimeon, and PhilipWadler *XQuery 1.0 and XPath 2.0 Formal Semantics*. World Wide Web Consortium Candidate Recommendation, September 2005. http://www.w3.org/TR/xquery-semantics

[Olteanu] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry *XPath: Looking Forward*. Proceedings of Workshop on XML-Based Data Management (XMLDM) at EDBT 2002, Prague, March 2002.

[OOo] *OpenOffice.org*. http://www.openoffice.org/

[SAX] *Simple API for XML*. http://www.saxproject.org/

[SDO] John Beatty, Stephen Brodsky, Micahel Carey, Raymond Ellersick, Martin Nally, and Radu Preotiuc-Pietro *Service Data Objects*, Version 2.0. IBM and BEA, June 2005. Available (with Java interfaces) from http://www-128.ibm.com/developerworks/library/specification/j-commonj-sdowmt/.

[SEQUEL2] Don Chamberlin et.al. *SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control*. IBM Journal of R&D, 20(6), November 1976, 560.

[SQLX] *SQL/XML* (ISO/IEC 9075-14:2005(E): Information technology - Database languages - SQL - Part 14: XML-Related Specifications). ISO/IEC JTC 1/SC 32/WG 3. http://www.sqlx.org/SQL-XML-documents/5FCD-14-XML-2004-07.pdf

[StAX] *Streaming API for XML*. Java Community Process JSR-173, December, 2003. http://www.jcp.org/aboutJava/communityprocess/final/jsr173/

[STX] Oliver Becker, Paul Brown, and Petr Cimprich *An Introduction to Streaming Transformations for XML*. XML.com, February 2003. http://stx.sourceforge.net/

[Virtual] Kristoffer Rose, Lionel Villard, Achille Fokoue, Rajeshwari Rajendra, Paul Castro, Christopher Holtz, William Li, and Stefan Schmidt *Virtual XML "Garden"*, IBM alphaWorks, November 2005. http://www.alphaworks.ibm.com/tech/virtualxml

[ViteX] Yi Chen, Susan B. Davidson, and Yifeng Zheng *ViteX: a Streaming XPath Processing System*, International Conference on Data Engineering (ICDE), Tokyo, Japan April 2005. http://csdl.computer.org/comp/proceedings/icde/2005/2285/00/22851118.pdf

[WSDL] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana *Web Services Description Language (WSDL) 1.1*. World Wide Web Consortium Recommendation, March 2004. http://www.w3.org/TR/wsdl

[Xalan] *Xalan*. Apache Software Fundation February, 2004. http://xml.apache.org/xalan-j/

[Xaos] Charles Barton, Philippe Charles, Deepak Goyal, Mukund Raghavachari, Vanja Josifovski, and Marcus Fontoura *Streaming XPath Processing with Forward and Backward Axes*. ICDE - International Conference on Data Engineering, Bangalore, India, March, 2003. http://www.research.ibm.com/xaos/applications.html

[XML] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, Francois Yergeau, and John Cowan *Extensible Markup Language (XML) 1.1*. World Wide Web Consortium Recommendation, February 2004. http://www.w3.org/TR/xml11

[XmlReader] Dare Obasanjo and Howard Hao *The Best of Both Worlds: Combining XPath with the XmlReader*. MSDN, May 2004. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexxml/html/xml05192004.asp

[XPath] JamesClark and SteveDeRose *XML Path Language (XPath) Version 1.0*. World Wide Web Consortium Recommendation, November 1999. http://www.w3.org/TR/xpath

[XPath2] Anders Berglund, Scott Boag, Don Chamberlin, Mary Fernandez, Michael Kay, Jonathan Robie, and Jerome Simeon *XML Path Language (XPath) 2.0*. World Wide Web Consortium Candidate Recommendation, November 2005. http://www.w3.org/TR/xpath20

[XPP] Aleksander Slominski, *XML Pull Parser (XPP)*. http://www.extreme.indiana.edu/xgws/xsoap/xpp/

[XQuery] Scott Boag, Don Chamberlin, Mary Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon *XQuery 1.0: An XML Query Language*. World Wide Web Consortium Candidate Recommendation, November 2005. http://www.w3.org/TR/xquery

[XS] David Fallside and Priscilla Walmsley *XML Schema Part 0: Primer Second Edition*. World Wide Web Consortium Recommendation, November 2004. http://www.w3.org/TR/xmlschema-0/

[XSQ] Feng Peng and Sudarshan Chawathe *XPath Queries on Streaming Data*. ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2003. http://www.cs.umd.edu/projects/xsq/

[XSLT] JamesClark. *XSLT Transformations (XSLT) Version 1.0*. World Wide Web Consortium Recommendation, November 1999. http://www.w3.org/TR/xslt

[XSLT2] MichaelKay. *XSLT Transformations (XSLT) Version 2.0*. World Wide Web Consortium Candidate Recommendation, November 2005. http://www.w3.org/TR/xslt20

[Zip] *Info-ZIP*. http://www.info-zip.org/

Biography

Kristoffer **Rose**
Research Staff Member
IBM Corporation [http://www.ibm.com]
T. J. Watson Research Center [http://www.research.ibm.com/]
Yorktown Heights
New York
United States of America
krisrose@us.ibm.com

Kristoffer Rose is a research staff member at the IBM Thomas J. Watson Research Center. He got his Ph.D. in computer science from the University of Copenhagen in 1996 doing research in tree and graph rewriting systems. After four years in academia, last as an associate professor at Ecole Normale Superieur in Lyon, France, he joined IBM in 2000, where he conducts researches into the theory and practice of XML processing.

Lionel **Villard**
Advisory Software Engineer
IBM Corporation [http://www.ibm.com]
T. J. Watson Research Center [http://www.research.ibm.com/]
Yorktown Heights
New York
United States of America
villard@us.ibm.com

Lionel Villard is an advisory research engineer at the IBM T. J. Watson Research Center. He received his Ph.D. at the Institut National Polytechnique de Grenoble (INPG) in March 2002. Dr. Villard's research interests include multimedia documents, contextual adaptation, authoring tools, document transformations, incremental transformations, and high performance.