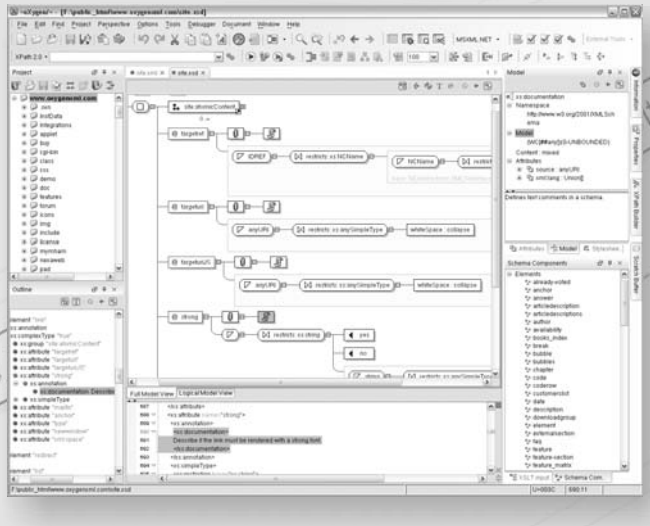
a conference on XML

Lesser Town Campus
Prague, Czech Republic
June 17-18, 2006

www.oxygenxml.com

<oXygen/>®

- XML Editor
- XML Schema Editor
- XSLT Debugger
- XQuery Debugger
- Support for
  - eXist
  - Berkeley DB



<oXygen/> is a simple to use and complete XML editor. The extensive list of features helps you create and transform XML documents, develop schemas and XSL stylesheets. Starting with the version 7 it is possible to run Xquery expressions against native XML databases.

Contact:
Phone: +40 251 461480; Fax: +40 251 461482; support@oxygenxml.com

syncro soft

Contents

General Information

Date

Saturday, June 17th, 2006
Sunday, June 18th, 2006

Location

Lesser Town Campus, Lecture Halls S5, S6
Malostranské náměstí 25, 110 00 Prague 1, Czech Republic

Speakers

David Carlisle, *NAG Limited*
Wolfgang Meier, *eXist XML database*
Eric van der Vlist, *Dyomedea*
Michael Kay, *Saxonica Limited*
Kamil Toman, Irena Mlýnková, *Charles University, Prague*
George Cristian Bina, *Syncro Soft – oXygen XML editor*

Organizing Comitee

Jaroslav Nešetřil, *Charles University, Prague*
Tomáš Kaiser, *University of West Bohemia, Pilsen*
Petr Cimprich, *Ginger Alliance, s.r.o.*
James Fuller, *Webcomposite, s.r.o.*

Proceedings Typesetting

Vít Janota, *Ginger Alliance, s.r.o.*

Preface

This publication contains papers to be presented at XML Prague 2006, a regular conference on XML for developers, web designers, information managers, and students. The conference focuses this year on XML Native Databases and Querying XML. A full day of experts speaking has been extended with an additional day dedicated for participants to hold related BoF sessions and workshops.

The conference is hosted at the Lesser Town Campus of the Faculty of Mathematics and Physics, Charles University, Prague. XML Prague 2006 is jointly organized by the Institute for Theoretical Computer Science and Ginger Alliance, s.r.o.

The conference joins the academic world with the world of IT professionals. The organizers hope this year's conference shall be enjoyable for both audience and speakers, providing a dynamic interchange and environment to discuss XML technologies.

Ginger Alliance

Ginger Alliance is a technology development company that takes a collaborative approach to providing solutions and cost effective outsourced development. Our aim is to make IT simpler and faster, using standards-based and if possible open source technologies.

Ginger Alliance is an acknowledged innovator in XML development and its use in the creation of Service Orientated Architectures to enable web services and deliver mobile applications.

Services

Ginger Alliance can help you to address your needs by providing top-quality professional and managed services.

Professional Sevices
Our motivated, well qualified consultants and developers provide: Technical definition, Software development and System integration

Managed Services
GA Mobile Marketing & Entertainment service (GA-MME) is a tool to create & deliver mobile content offers for marketing campaigns.

Products

GA - GinDjin Suite
A complete solution for mobile content delivery and management, with billing and reporting capabilities.

GA - Messaging Oriented Middleware
GA-MOM is a highly scalable and robust solution for messaging, communication, control and integration of new and existing services.

http://www.gingerall.com

Program

Saturday, June 17, 2006 - Lecture Hall S5

9:00	The Road to an XSLT/XQuery IDE, *George Cristian Bina*
9:35	Index-driven XQuery Processing in the eXist XML Database, *Wolfgang Meier*
10:35	*Coffee Break*
10:55	Optimization in XSLT and XQuery, *Michael Kay*
11:55	*Lunch Break (on your own)*
13:25	XML/RDF Query by Example, *Eric van der Vlist*
14:30	xq2xml: Transformations on XQueries, *David Carlisle*
15:30	*Coffee Break*
15:50	XML Data - The Current State of Affairs, *Kamil Toman, Irena Mlýnková*
16:25	Web 2.0: Myth and Reality, *Eric van der Vlist*
17:00	Lightening Sessions
17:30	*End of Day*

Sunday, June 18, 2006 - Lecture Hall S5

9:00	First eXist Workshop, *Wolfgang Meier*
10:40	*Coffee Break*
10:50	First eXist Workshop, *Wolfgang Meier*
13:00	*End of Day*

Sunday, June 18, 2006 - Lecture Hall S6

9:00	Perl XML BoF, *Petr Cimprich, Petr Pajas*
10:10	Xdefinition BoF, *Václav Trojan*
10:40	*Coffee Break*
10:50	DocBook BoF, *Jirka Kosek*
13:00	*End of Day*

Produced by

Institute for Theoretical Computer Science
(http://iti.mff.cuni.cz/)

Ginger Alliance (http://www.gingerall.com/)

with support of

University of West Bohemia in Pilsen (http://www.zcu.cz/)

Sponsored by

oXygen XML editor (http://www.oxygenxml.com/)

Main Media Partner

ROOT.cz (http://root.cz/)

Media Partners

jax*magazine* (http://jaxmag.com/)

ZVON.org (http://www.zvon.org/)

Bulgaria WebDev Magazine
(http://openideascompany.com/wdm.php)

<xml>*fr* (http://xmlfr.org/)

CHIP (http://chip.cz/)

The Road to an XSLT/XQuery IDE

George Cristian Bina (Syncro Soft – oXygen XML Editor)

Abstract

Most XSLT/XQuery tools provide editing and transformation support. Some provide also debugging and profiling support. However there is a lot more to do in order to have an XSLT/XQuery IDE similar with today's Java IDEs for instance. oXygen started to walk on this road providing a number of advanced development features. An overview of the features that should be available in a modern XSLT/XQuery IDE and more details about the part of these features already implemented in oXygen will be presented.

Checking for errors and error reporting

When editing source code every developer needs to know whether or not his code contains errors, to see where these errors are and to get their descriptions so he can easily correct them. Checking the source for errors can be done on demand or it can be performed automatically (continuous validation). The continuous validation is important because it signals the error immediately when it appears thus the developer is already focused on that and can more easily correct it. The errors can be reported in a list or in a table but they can also be marked directly in the source text. Marking errors in the source text with visual error markers is useful especially when coupled with the automatic validation as the detected errors are located naturally on screen. It is important also to report as many errors as possible and do not stop after the first one. Sometimes the developer can choose to correct an error at a later time but if the error reporting stops on the first encountered error then he will not get any more feedback. To summarize, an IDE should provide support for *continuous validation*, *visual error markers* and it should report *multiple errors*. The validation should be *performed in background* and should not interrupt the user from editing.

Example 1: Error markers in oXygen on continuous validation

The error checking can verify only the XSLT/XQuery syntax or it can perform a more powerful checking, signaling for instance undeclared components, XPath errors, etc. It is desirable for an IDE to have a *more powerful checking* than a simple syntax check.

One of the characteristics of XSLT and XQuery is that they allow modules that can be invalid by themselves but that can became valid if they are used in some context (included or imported for instance). For example the following stylesheet uses a variable `handleElements` that is defined in a stylesheet that includes this one. Thus the stylesheet is invalid by itself, but when included from a stylesheet that defines the `handleElements` variable it is valid.

Example 2: Invalid stylesheet – `sample2module.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="*">
    <xsl:if test="$handleElements='true'">
      <xsl:apply-templates/>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

Example 3: Valid stylesheet that includes an invalid by itself module – `sample2main.xsl`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:variable name="handleElements" select="'true'"/>
  <xsl:include href="sample2module.xsl"/>
</xsl:stylesheet>
```

There are two possible approaches to solve this. One is to *determine the hierarchy of included/imported files automatically* and thus determine main files that are not included/imported and module files that are included/imported.

The problem is this case is that the actual user usage of the files may be different than what the automatic detection can give and the user has no control over it. The other approach is to *allow the user to define a main file* for a module. Then instead of performing the validation on the module file, the main file should be validated. Thus *the module gets validated in the context it was included or imported from.* The problem in this case is that it requires specific user actions. Most of this overhead can be minimized if there are provided some power actions to mark the files as main or module files. For instance the user should be able to trigger an automatic marking of files as main or module files based on the import/include structure. An XSLT/XQuery IDE should provide the means for handling main and module files. In oXygen we plan to implement support for allowing the user to specify the main files for modules.

Navigation and refactoring

An XSLT/XQuery IDE should provide easy navigation through the edited source. This is useful both when editing and when looking over the source code and trying to understand it. Examples of navigation support include going from a *reference to its definition*, *finding all references* or *all occurrences* of some component, *opening an included/imported file.*

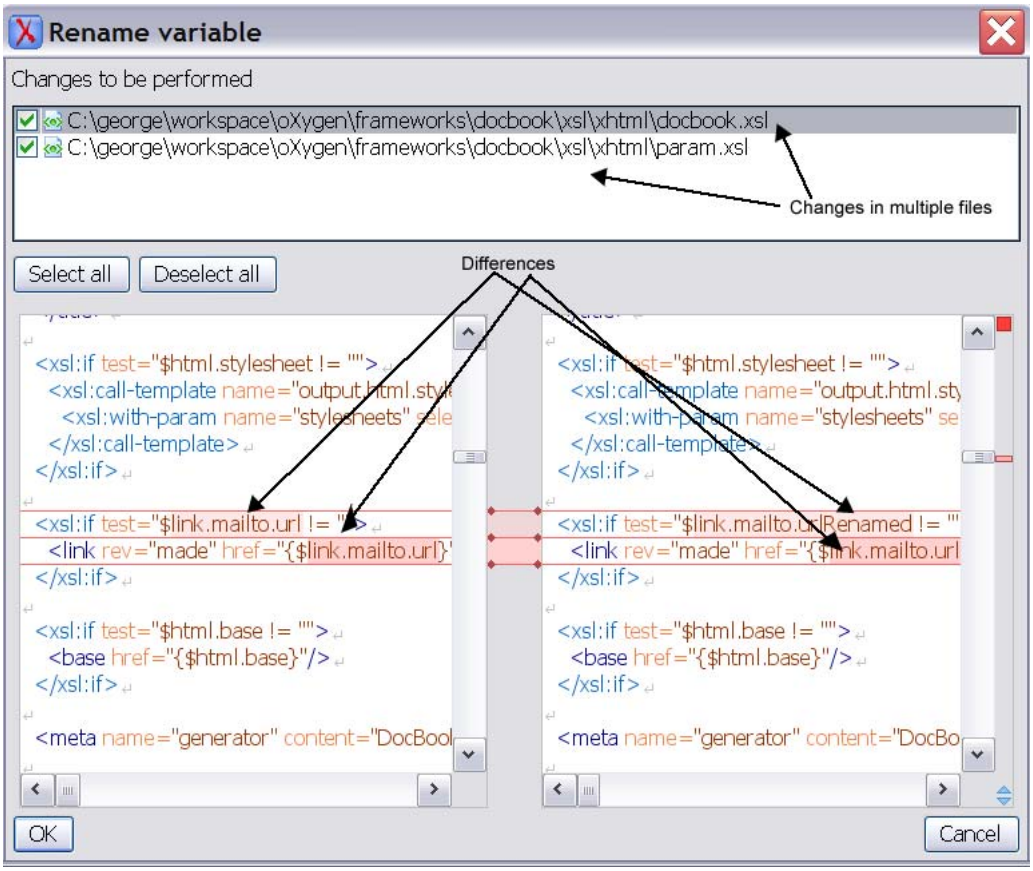
It is useful also to have an *outline* of the edited source, for example for stylesheets one can get a quick view of the stylesheet looking at its defined templates. The outliner should support *synchronization* with the position in source and should allow navigation to the location of presented components.

There are some cases the navigation support should take into account. When editing, *the document may not be always wellformed* so the navigation support should consider handling also this situation. The navigation needs a *scope* (*current file*, *all the project*, etc.) where the search for components definitions or references should be performed. Between current file and all the project the IDE should allow defining also other scopes more generic than the current file and more restricted than the whole project. Possible scopes can be *all the files determined after computing the closure on include/import* starting from a given file or *user defined working sets.*

An IDE should help when performing semantic changes that are grouped in *refactoring actions.* The most useful and used refactoring action is the *rename* action. In general a refactoring action can affect *more than one location in a file* and even *more files*, thus it is useful to present a *diff* between the previous state of the affected files and the state of those files after the refactoring allowing the user to see and approve the changes to be performed.

As for navigation the user should be able to specify the *scope* on which the refactoring action applies.

Example 4: Renaming a variable with oXygen in DocBook XHTML stylesheets

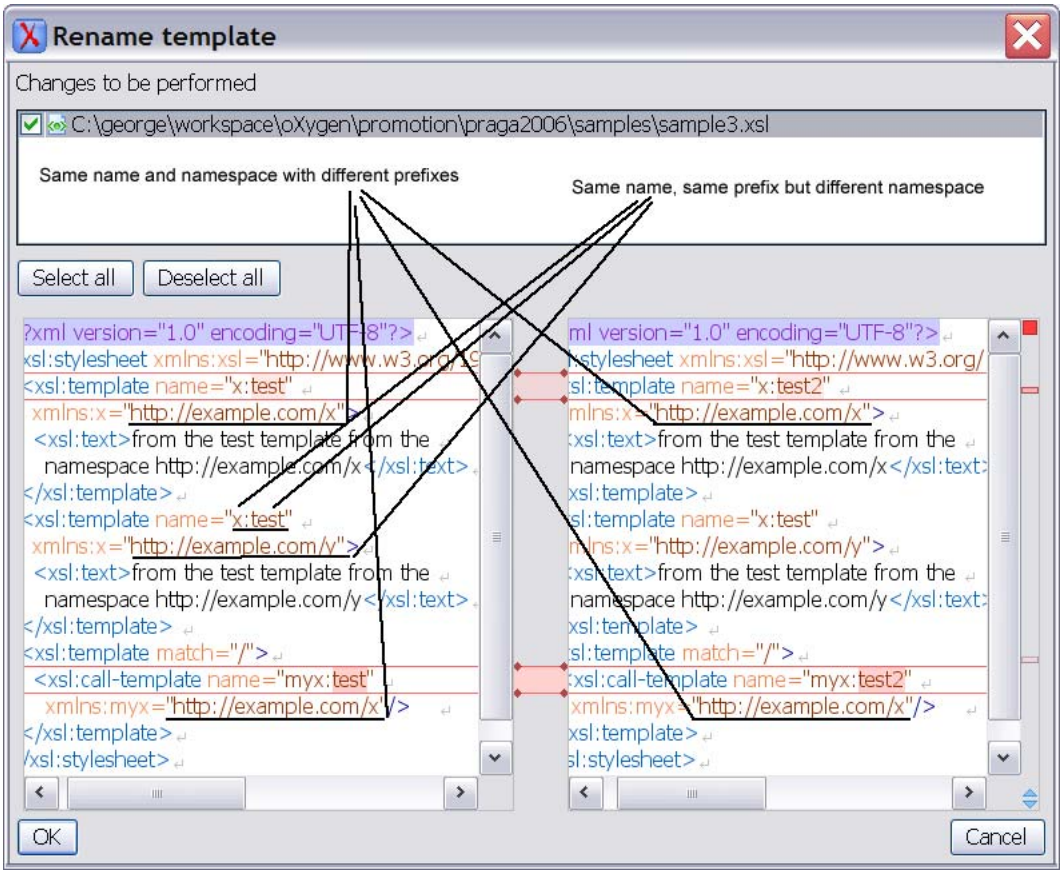


The renaming action applies to all named components defined by the language. The IDE should determine the component name and type automatically. Some components can be qualified (can belong to a namespace) and the IDE should properly handle these cases.

A number of refactoring actions address the organization of the source code to facilitate its understanding and reuse. These actions generally extract parts of the code and place them in a component that can be referred from different locations. Also the reverse should be provided, that is in-lining a reference to a component. oXygen currently implements 3 such actions for XSLT, namely *create template from selection*, *create stylesheet from selection* and *extract attributes as* `xsl:attributes`. Every refactoring action requires careful implementation as they are semantic action and the result code should be *functionally equivalent* with the code before the refactoring. For instance if we take the case of extracting a fragment of code as a template, then that fragment of code should run in the new template in the *same context* as it

used to run, context means in this case *same namespace mappings* and *same variables/parameters*. If a local variable is used in the code fragment that was defined outside that fragment the refactoring action should automatically define a parameter and call the template with that parameter passing it the proper variable value.

Example 5: Renaming a template from a namespace



Example 6: Extract selection as template example

We have a simple stylesheet that matches on the document root and iterates on the children of the root element printing their name followed by their position.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
  <xsl:template match="/">
    <result>
      <xsl:variable name="elements" select="/*/*"/>
      <xsl:for-each select="$elements">
        <xsl:variable name="pos" select="position()"/>
        <xsl:value-of select="name()"/>
        <xsl:text>-</xsl:text>
```

```
        <xsl:value-of select="$pos"/>
      </xsl:for-each>
    </result>
  </xsl:template>
</xsl:stylesheet>
```

After selecting the `for-each` element and extracting that as a template we get a new template with a parameter named `elements` that is used to pass along the `elements` local variable defined before the refactored code fragment.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
  <xsl:template match="/">
    <result>
      <xsl:variable name="elements" select="/*/*"/>
      <xsl:call-template name="printElements">
        <xsl:with-param name="elements" select="$elements"/>
      </xsl:call-template>
    </result>
  </xsl:template>
  <xsl:template name="printElements">
    <xsl:param name="elements"/>
    <xsl:for-each select="$elements">
      <xsl:variable name="pos" select="position()"/>
      <xsl:value-of select="name()"/>
      <xsl:text>-</xsl:text>
      <xsl:value-of select="$pos"/>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

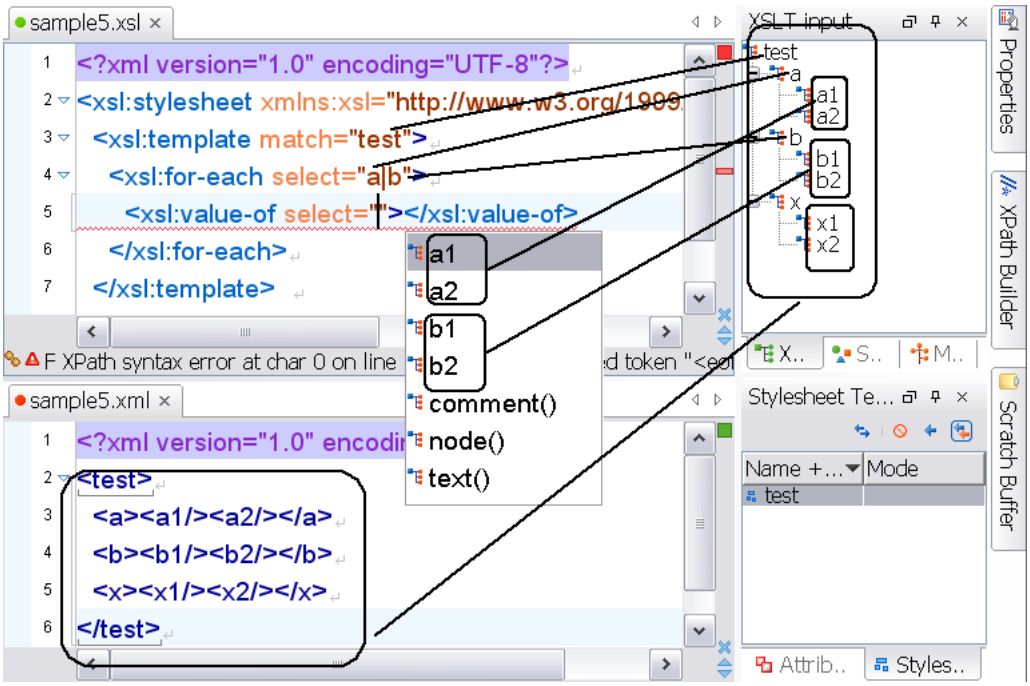
Advanced content completion

An IDE should help easily entering content and one possibility of doing that is by using the content completion. When presenting proposals it is useful to have also a *description* of the presented proposal, similarly with how a Java IDE presents the Javadoc documentation for a method. Basic content completion will handle *static proposals*, while more advanced content completion should support *dynamic proposals* as well. The static proposals are the same for any document and do not depend on the current content, for instance here we can include all the instructions, keywords, built-in functions. Dynamic proposals depend on the document already entered information, here we can include user defined functions or templates, variables and parameters, elements and attributes belonging to the output grammar.

Both XSLT and XQuery use XPath so the IDE should be able to offer also *XPath content completion.* Here we can also distinguish between static

and dynamic proposals. As static proposals we have functions and axes and as dynamic proposals we have variables, parameters, elements and attributes name tests. When computing the available name tests all the instructions that can change the context should be taken into account.

Example 7: Name tests in context



Only the `a1`, `a2`, `b1` and `b2` name tests are proposed by oXygen, as the context is determined first by the `template` match on the `test` element then it is changed by the `for-each` instruction on `a` and `b` elements so inside `value-of` we can only have the children of `a` and `b` and not `x`, `x1` or `x2`.

To bridge the support an IDE can offer in a specific situation with more advanced user needs, an IDE should offer also support for *user defined code templates*. Thus the user can enter specific code fragments and he should be able to easily insert and customize them in the code. For XSLT and XQuery editors the IDE should offer an already defined set of templates covering the most used fragments of code.

Running configurations

The IDE should provide support for *defining multiple running configurations*. It should be possible to *reuse* a running configuration (transformation scenario) defined for a stylesheet or XQuery file for other files. This will be more useful if in defining the running configurations the user can use *variables* like current file, current directory, etc. These allow defining *generic running configurations* like for instance applying an XSLT transformation on

a document with the same name as the stylesheet from the same directory and saving the result in the same directory with the same name but with the `.html` extension, then the same scenario can be used to transform `x.xml` with `x.xsl` into `x.html` and for `test.xml` with `test.xsl` to `test.html` and so on.

Support for multiple XSLT processors and multiple XQuery servers or processors allows the developer to use during development the *same configuration as his production system*. Also this covers the usage of specific *extensions*. oXygen supports the following XSLT processors: Saxon 6, Saxon8B and Saxon8SA from Saxonica, Xalan, XSLTProc, MSXML 3.0, MSXML 4.0 and .NET 1.0 and .NET 2.0 and allows to plug in other processors either through JAXP or as external processors. For XQuery oXygen supports Saxon8, eXist, Berkeley XML, X-Hive, TigerLogic and MarkLogic and also it allows to plug in external XQuery processors.

For obtaining PDF or PS output from XML it would be desirable from an XLST IDE to provide also support for transforming XSL-FO with a formatting objects processor. oXygen includes Apache FOP and can be configured to use also commercial processors like RenderX XEP and AntennaHouse XSL Formatter.

Debugging and profiling

The minimal debugging support like stepping through a transformation, visualize the variables and their values, see the execution trace etc. should be available in an XSLT/XQuery IDE.

There are also a number of advanced features an XSLT/XQuery debugger may implement.

One such feature is the support for *mapping from the output result to the instructions that generated that output* and eventually also to the source node that represented the current context when that output was generated. This allows easy navigation when something wrong is detected in the result output to the source location that output was produced from.

Another advanced debugging feature is support for *conditional breakpoints* that will stop the debugger not when some location is reached but when some condition is met.

Support for *changing variables* in the debugger and continue the execution with the new value can also help although this may induce unexpected and not reproducible results.

One difficulty with implementing debugging is that both for XSLT and XQuery *there is not defined a debugging interface*, thus in each case the implementation is *processor/server specific*. It is useful for an IDE to have

debugging support for as many processors as possible. The users should take into account that sometimes the source may be different than the actual processor execution, a typical example here is for instance the lazy evaluation of variables.

The main views provided by a profiler are the *hot spots view* and the *invocation tree*. The profiling is quite related with the debugging support as basically if someone can execute step by step a transformation then also profiling information can be obtained so the difficulties encountered in debugging are found also in providing profiling support. The main problem with a profiler is to compute the processor time actually spent for an instruction execution. Profilers just take two time readings at the start and at the end of an instruction and compute the difference. However this time is not the actual processor time as it includes the time the processor runs other processes on that machine or other threads of the same application. The ideal profiler should compute the real processor time. For that, in case of Java based processors, it should start the transformation with a Java machine and register itself to that for getting Java profiling information. Then that information should be correlated with the XSLT/XQuery profiling information to find the actual processor time spent for executing an instruction. However, the main performance problems can be discovered also with a not such accurate profiler and very accurate execution times can be used only to observe small performance differences. Thus the effort to obtain such accurate profiling results is not justified.

Visual editing

Visual editing is useful for speeding up development. Here there are a couple of concepts: *drag and drop editing* and *visual mappers*.

Drag and drop editing allows quickly creating fragments of code starting from dropping a node from the input in the source editor. Here the XPath expression should be resolved correctly in context.

Visual mappers allow graphically mapping between input and desired output and they generate the code to implement that mapping.

Documentation and unit testing

An IDE should be able to present the documentation introduced in source to the user when that is appropriate. For instance when configuring a transformation scenario it is useful to present documentation for each parameter, if available. Also the IDE should be able to create documentation from the sources similar with the Javadoc documentation.

Unit testing is a very important component in development and an IDE should help with *defining and running unit tests*.

A lot of little smart actions

The IDE editors should provide help specific to XSLT and XQuery editing whenever possible . Little things although simple can dramatically improve the user experience. A few examples are: *transform an empty element in a non empty one*, *jump to the next editing position*, *smart indenting*, *element selection*, *content selection*, *indent on paste*, *toggle comment*, etc.

Conclusion

Most of the features expected from an XSLT/XQuery IDE are already provided by oXygen, the XSLT support being better than the XQuery support. Further plans are to improve the XQuery support to match the XSLT support, to add main documents support so that modules can be validated in the context they are used in, to add more refactoring actions and to explore the possibilities for offering a visual mapper.

Index-driven XQuery Processing in the eXist XML Database

Wolfgang Meier (eXist XML database)

Introduction

eXist provides its own XQuery implementation, which is backed by an efficient indexing scheme at the database core to support quick identification of structural node relationships. eXist's approach to XQuery processing can thus not be understood without knowing the indexing scheme and vice versa. Over the past years, two different indexing schemes for XML documents were implemented in eXist: while the first was based on an extended level-order numbering scheme, we recently switched to hierarchical node IDs to overcome the limitations imposed by the former approach.

The presentation provides a quick overview of these developments. I will try to summarize the logic behind both indexing schemes we implemented in eXist, and point out their strengths and deficencies. Based on this foundation, we should have a look at some central aspects of XQuery processing and optimization.

Two indexing schemes for XML documents

The indexing scheme forms the very core of eXist and represents the basis of all efficient query processing in the database. Understanding the ideas behind it helps to understand why some query formulations might be faster than their alternatives and is certainly a requirement for those who want to work on improving eXist's XQuery engine.

The main purpose of the indexing scheme is to allow a quick identification of structural relationships between nodes. To determine the relationship between any pair of given nodes, the nodes themselves don't need to be loaded into main memory. This is a central feature since loading nodes from persistent storage is in general an expensive operation. For eXist, the information contained in the structural index is sufficient to compute a wide-range of path expression.

To make this possible, every node in the document tree is labeled with a unique identifier. The ID is chosen in such a way, that we can compute the relationship between two nodes in the same document from the ID alone. Ideally, no further information is needed to determine if a given node can be the ancestor, descendant or sibling of a second node. We don't need to have access to the actual node data stored in the persistent DOM.

If we know all the IDs of elements A and B in a set of documents, we can compute an XPath expression `A/B` or `A//B` by a simple join operation between node set A and B. The concrete implementation of this join operation will depend on the chosen labelling scheme. However, the basic idea will more or less remain the same for most schemes. We will have a closer look at join operations and their characteristics below.

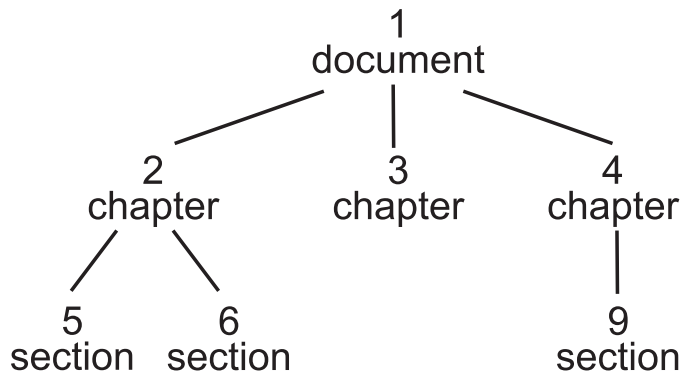
Level-order numbering

As explained above, the indexer assigns a unique node ID to every node in the document tree. Various labeling or numbering schemes for XML nodes have been proposed over the past years. Whatever numbering scheme we choose, it should allow us to determine the relationship between any two given nodes from the ID alone.

The originial numbering scheme used in eXist was based on level-order numbering. In this scheme, a unique integer ID is assigned to every node while traversing the document tree in level-order. To be able to compute the relationship between nodes, level-order numbering models the document tree as a complete k-ary tree, i.e. it assumes that every node in the tree has exactly k child nodes. Obviously, the number of children will differ quite a lot between nodes in the real document. However, if a node has less than k children, we just leave the remaining child IDs empty before continuing with the next sibling node.

As a consequence, a lot of IDs will not be assigned to actual nodes. Since the original level-order numbering scheme requires the resulting tree to be k-ary complete, it runs out of available IDs quite fast, thus limiting the maximum size of a document to be indexed.

Document with level-order IDs



To work around this limitation, eXist implemented a *relaxed* version of the original proposal. The completeness constrained was dropped in part: instead of assuming a fixed number of child nodes for the whole document, eXist recomputes this number for every level of the tree. This simple trick raised the document size limit considerably. Indexing a regularily structured document of a few hundred megabyte was no longer an issue.

However, the numbering scheme still doesn't work well if the document tree is rather imbalanced. The remaining size limit may not be a huge problem for most users, but it can really be annoying for certain types of documents. In particular, documents using the TEI standard may hit the wall quite easily.

Apart from its size restrictions, level-order numbering has other disadvantages as well. In particular, inserting, updating or removing nodes in a stored document is very expensive. Each of these operations requires at least a partial renumbering and reindexing of the nodes in the document. As a consequence, node updates via XUpdate or XQuery Update extensions were never very fast in eXist.

So why did we stay with this numbering scheme for so long? One of the main reasons was that almost all of the proposed alternative schemes had other major drawbacks. For example, level-order numbering works well across all XPath axes while other schemes could only be used for child and descendant, but not the parent, ancestor or sibling axes.

Switching to a new scheme

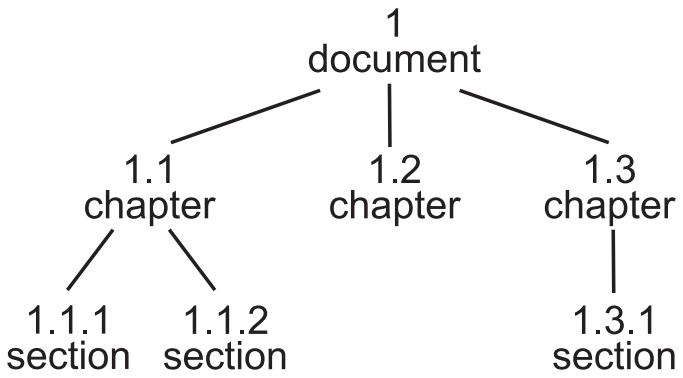
In early 2006, we finally started a major redesign of the whole indexing core of eXist. As an alternative to the old level-order numbering, we chose to implement *dynamic level numbering* (DLN) as proposed by Böhme and Rahm[1]. This numbering scheme is based on variable-length IDs and thus

[1]Böhme, T.; Rahm, E.: Supporting Efficient Streaming and Insertion of XML Data in RDBMS. Proc. 3rd Int. Workshop Data Integration over the Web (DIWeb), 2004

avoids to put a conceptual limit on the size of the document to be indexed. It also has a number of positive side-effects, including fast node updates without reindexing.

Dynamic level numbers (DLN) are hierarchical IDs, inspired by Dewey's decimal classification. Dewey IDs are a sequence of numeric values, separated by some separator. The root node has ID 1. All nodes below it consist of the ID of their parent node used as prefix and a level value. Some examples for simple node IDs are: 1, 1.1, 1.2, 1.2.1, 1.2.2, 1.3. In this case, 1 represents the root node, 1.1 is the first node on the second level, 1.2 the second, and so on.

Document with DLN IDs



Using this scheme, to determine the relation between any two given IDs is a trivial operation and works for the ancestor-descendant as well as sibling axes. The main challenge, however, is to find an efficient encoding which

1. restricts the storage space needed for a single ID, and
2. guarantees a correct binary comparison of the IDs with respect to document order.

Depending on the nesting depth of elements within the document, identifiers can become very long (15 levels or more should not be uncommon).

The original proposal describes a number of different approaches for encoding DLN IDs. We decided to implement a variable bit encoding that is very efficient for streamed data, where the database has no knowledge of the document structure to expect. The stream encoding uses fixed width units (currently set to 4 bits) for the level IDs and adds further units as the number grows. A level ID starts with one unit, using the lower 3 bits for the number while the highest bit serves as a flag. If the numeric range of the 3 bits (1..7) is exceeded, a second unit is added, and the next highest bit set to 1. The leading 1-bits thus indicate the number of units used for a single ID.

The following table shows the ID range that can be encoded by a given bit pattern:

ID ranges

No. of units	Bit pattern	Id range
1	0XXX	1..7
2	10XX XXXX	8..71
3	110X XXXX XXXX	72..583
4	1110 XXXX XXXX XXXX	584..4679

The range of available IDs increases exponentially with the number of units used. Based on this algorithm, an ID like 1.33.56.2.98.1.27 can be encoded with 48 bits. This is quite efficient compared to the 64 bit integer IDs we used before.

Besides removing the document size limit, one of the distinguishing features of the new indexing scheme is that it is insert friendly! To avoid a renumbering of the node tree after every insertion, removal or update of a node, we use the idea of sublevel IDs also proposed by Böhme and Rahm. Between two nodes 1.1 and 1.2, a new node can be inserted as 1.1/1, where the / is the sublevel separator. The / does not start a new level. 1.1 and 1.1/1 are thus on the same level of the tree. In binary encoding, the level separator '.' is represented by a 0-bit while '/' is written as a 1-bit. For example, the ID 1.1/7 is encoded as follows:

```
0001 0 0001 1 1000
```

Using sub-level IDs, we can theoretically insert an arbitrary number of new nodes at any position in the tree without renumbering the nodes.

XQuery processing

Based on the features of the numbering scheme, eXist uses structural joins to resolve path expressions. If we know the IDs of all nodes corresponding to e.g. *article* elements in a set of documents and we have a second node set representing *section* nodes, we can filter out all sections being children or descendants of an article node by applying a simple join operation on the two node sets.

In order to find element and attribute occurrences we use the central structural index stored in the database file `elements.dbx`. This is basically just a map, connecting the QNames of elements and attributes to a list of (*documentID*, *nodeID*) pairs. Actually, most of the other index files are structured in a similar way. To evaluate an XPath expression like

`/article/section`, we look up *article* and *section* in the structural index, generate the node sets corresponding to the occurrences of these elements in the input set, then compute an ancestor-descendant join between both sets.

Consequently, eXist does rarely use tree traversals to evaluate path expressions. There are situations where a tree traversal is without alternative (for example, if no suitable index is available), but eXist usually tries to avoid it wherever it can. For example, the query engine even evaluates an expression like `//A/*[B = 'C']` without access to the persistent DOM, though `A/*` would normally require a scan through the child nodes of A. However, eXist instead defers the evaluation of the wildcard part of the expression and later filters out those nodes which cannot be children of A.

This approach to index-based query processing leads to some characteristic features:

First, operations that require direct access on a stored node will nearly always result in a significant slow-down of the query − unless they are supported by additional indexes. This applies, for example, to all operations requiring an atomization step. To evaluate `B = 'C'` without index assistance, the query engine needs to do a *brute force* scan over all B elements. With a range index defined on B, the expression can again be processed by just using node IDs.

Second, the query engine is optimized to process a path expression on two given node sets in one, single operation. The join can handle the entire input sequence at once. For instance, the XPath expression `//A/*[B = 'C']` is evaluated in a single operation for all context items. Also, the context sequence may contain nodes from an arbitrary number of different documents in different database collections. eXist will still use only one operation to compute the expression. It doesn't make a difference if the input set comes from a single large document, includes all the documents in a specific collection or even the entire database. The logic of the operation remains the same − though the size of the processed node set does certainly have an impact on performance.

This behaviour has further consequences: for example, eXist prefers XPath predicate expressions over an equivalent FLWOR construct using a `where` clause. The `for` expression forces the query engine into a step-by-step iteration over the input sequence. When optimizing a FLWOR construct, we thus try internally to process a `where` clause like an equivalent XPath predicate if possible. However, computing the correct dependencies for all variables that might be involved in the expression can be quite difficult, so a predicate does usually guarantee a better performance.

Unfortunately, many users prefer SQL-like `where` clauses in places where an XPath predicate would have the same effect and might even improve the

readability of the query. For example, I sometimes see queries like this:

```
for $i in //entry where $i/@type = 'subject'
or $i/@type = 'definition' or $i/@type = 'title' return $i
```

which could be reduced to a simple XPath:

```
//entry[@type = ('subject', 'definition', 'title')]
```

Finally, for an explicit node selection by QName, the axis has only a minimal effect on performance: `A//B` should be as fast as `A/B`, `A/ancestor::B` or even `A/following-sibling::B`. The node set join itself accounts only for a small part of the processing time, so differences in the algorithm for parent-child or ancestor-descendant joins don't have a huge effect on overall query time.

eXist also has a tendency to compute node relationships bottom-up instead of top-down. Queries on the parent or ancestor axis are fast and it is often preferable to explore the context of a given node by going up the ancestor axis instead of traversing the tree beginning at the top. For instance, the following query uses a top-down approach to display the context of each match in a query on *articles*:

```
for $section in collection("/db/articles")//section
for $match in $section//p[contains(., 'XML')]
return
  <match>
    <section>$section/title/text()</section>
    {$match}
  </match>
```

This seems to be a rather natural way to formulate the query, but it forces eXist to evaluate the inner selection `$section//p[contains(., 'XML')]` once for every *section* in the collection. The query engine will try to optimize this a bit by caching parts of the expression. However, a better performance can be achieved by slightly reformulating the query to navigate to the *section* title along the ancestor axis:

```
ffor $match in collection("/db/articles")//section//p[contains(., 'XML')]
return
  <match>
    <section>$section/ancestor::title/text()</section>
    {$match}
  </match>
```

Outlook

As of May 2006, the redesign of the database core is basically complete and the code is to be merged into the main line of development. The ability to update nodes without reindex has a huge performance impact for all applications that require dynamic document updates. The annoying document size limits are finally history.

We are still not running out of ideas though. Our recent redesign efforts will be extended into other areas. Concerning the database core, there are two major work packages I would like to point out:

First, the *indexing system* needs to be modularized: index creation and maintenance should be decoupled from the database core, making it possible for users to control what is written to a specific index or even to plug in new index types (spatial indexes, n-gram ...). Also, the existing full text indexing facilities have to become more extensible to better adopt to application requirements. The current architecture is too limited with respect to text analysis, tokenization, and selective tokenization. Plans are to replace these classes by the modular analyzer provided by Apache's Lucene.

The second area I would like to highlight is *query optimization*: currently, query optimization does mainly deal with finding the right index to use, or changing the execution path to better support structural joins. Unfortunately, the logic used to select an execution path is hard-coded into the query engine and the decision is mostly made at execution, not compile time. It is thus often difficult to see if the correct optimization is applied or not.

What we need here is a better pre-processing query optimiser which sits between the frontend and the query backend. Instead of selecting hard-coded optimizations, the new optimizer should rather rewrite the query before it is passed to the backend, thus making the process more controlable.

Also, a good part of the remaining performance issues we observe could be solved by intelligent query rewriting. We indeed believe there's a huge potential here, which is not yet sufficiently used by eXist. Though node sets are basically just sequences of node IDs, processing queries on huge node sets can still pose a major problem with respect to memory consumption and performance in general. Query rewriting can be used to reduce the general size of the node sets to be processed, for instance, by exploiting the fact that eXist supports fast navigation along all XPath axes, including parent and ancestor. Those parts of the query with a higher selectivity can be pushed to the front in order to limit the number of nodes passed to subsequent expressions. This can result in a huge performance boost for queries on large document sets.

Optimization in XSLT and XQuery

Michael Kay (Saxonica Limited)

Abstract

XSLT and XQuery are both high-level declarative languages, and as such, performance depends strongly on optimization. This talk gives a survey of the optimization techniques that are used, which are based on established techniques from both the functional programming and relational database traditions. The focus is on the techniques used in the speaker's own processor, Saxon (which implements both XSLT and XQuery) but the same range of techniques can be expected to be found in other products. In most cases, the same techniques apply to both languages, but there are some areas of difference which may be significant when choosing which language is most appropriate for your particular application.

The talk is designed primarily for users of XSLT and/or XQuery, but with the recognition that to get the best out of your chosen XSLT or XQuery processor, it's useful to know a little bit about what is happening under the covers.

Introduction

Performance matters, and it's a shared responsibility between suppliers of products such as compilers and database engines, and the programmers who write applications that use those languages and databases. One of the challenges with high-level declarative languages like XSLT and XQuery (and it applies equally well to SQL or Prolog) is that it's not always obvious to the programmer what the system is doing to execute a particular construct. This makes it hard to know how to write constructs that will perform well and avoid those that perform badly.

Adding to the difficulty is that different implementations of XSLT or XQuery are likely to optimize the same constructs in different ways, which means that the most efficient way of writing an expression for one processor might not be the most efficient on another.

The aim of this talk is to try to give you some idea of the optimizations that you can expect to find in most products, and those that might be present

in some but not others. It would be nice if one could say "assume that there will be no optimization", but unfortunately that just isn't realistic − if there were no optimization at all, then many simple expressions would perform impossibly slowly.

I'm going to concentrate on in-memory processors like Saxon. For XML database products (in fact, for any database product), 90% of optimization is about finding the right indexes to help execute the query. If you've got more than a gigabyte of data, then queries that aren't supported by indexes are going to take a rather long time. Wolfgang Meier has described in previous paper at this conference the indexing techniques used in the eXist product, which I suspect aren't that different at a conceptual level from those used in other XML databases. With an in-memory processor, indexing is also important, but in a rather different way, because any index that you use has to be constructed on the fly: when using an in-memory processor like Saxon the cost of building indexes is therefore higher, and because documents are smaller the benefit is less dramatic.

The term *optimization* is sometimes used exclusively to refer to the process of taking a source query or stylesheet, and generating a query execution plan which is then executed at run-time. I'm going to use the term rather more broadly to cover all the techniques used for performance improvement, which as well as compile-time query rewriting include design of data structures and techniques that kick in dynamically at run-time.

Saxon, incidentally, seems to have performance which is very competitive both among XSLT processors and among XQuery processors. It's not possible to achieve this solely by better optimization (in the sense of query rewriting) because a great many queries and stylesheets are extremely simple and very little optimization is possible. The first priority for any processor is not to do anything fancy or clever, it is simply to be efficient at doing the simple things.

I'm not going to cover XSLT and XQuery separately, because nearly all the techniques are common to the two languages. The main difference between them, from this point of view, is that XSLT has a dynamic despatch mechanism in the form of the `<xsl:apply-templates>` instruction and the associated template rules. This mechanism itself needs to be efficient, of course; but in addition, the existence and widespread use of the mechanism means that it's possible to do far less static analysis of an XSLT stylesheet than is possible with XQuery.

The tree model

The internal design of the data structures used to represent XML trees is probably one of the most critical factors affecting performance. This needs to satisfy a number of conflicting requirements:

- **Economical on memory.** If you're going to allow 100Mb XML documents to be processed in memory, then you can't afford to allocate 200 bytes per node, which is what a naive implementation of the tree structure will easily occupy. Saxon achieves around 20-30 bytes per node, depending on which features are used.
- **Fast construction.** Building the tree representing the input document can take as long as the subsequent query or transformation.
- **Fast access paths.** Navigating the XPath axes − especially the ones that are most commonly used − must be efficient.
- **Support for document order.** The semantics of XPath often require that nodes are returned in document order. It's often possible to achieve this requirement without doing a sort, but when a sort is needed, it must be efficient. That means that given two nodes, it must be very easy to work out their relative position in document order.

Saxon meets these requirements using a data structure called the TinyTree. This holds information in a collection of arrays. Ignoring attributes and namespaces for the moment, information about node N in the document (where nodes are numbered in document order) is held in the Nth element of these arrays. The arrays are as follows:

Array	**Information held**
depth	The depth of the node in the tree
kind	The kind of node (element, text, etc)
namecode	An integer code representing the name of the node
next	Pointer to the next following sibling
prior	Pointer to the preceding sibling
typecode	An integer code representing the type annotation, in the case of schema-aware processing
alpha, beta	Depends on node kind: for elements, pointers to the list of attributes and namespaces; for text nodes, pointers to the start and end of the string value

Note that there is no object representing a node. Instead, information about node N can be obtained by looking at `depth[N]`, `kind[N]`, `namecode[N]`,

and so on. While Saxon is navigating the tree, it uses transient objects to represent nodes, but these will be garbage-collected as soon as they are no longer needed. In fact, Saxon even goes to some care to avoid creating these transient node objects, since even today object creation in Java is an expensive operation.

Some of these arrays are created lazily. The `prior` array is created the first time someone uses the preceding-sibling or preceding axis on a tree. Often these axes are are never used, so one might as well save the space and time used to establish the pointers. Similarly, the `typecode` array is used only if the data has been schema-validated.

Perhaps the most important aspect of this design is the use of an integer code to represent the node name. The mapping from namespace-qualified names (QNames) to integer codes is established in a name pool, which must be available both when the query/stylesheet is compiled, and when the source document is constructed (either might happen first, and of course there is a many-to-many relationship between source documents and queries). The integer code uses 20 bits to represent the namespace URI and local name, and a further 10 bits for the prefix; there is thus sufficient information to reconstitute the prefix, but names can be compared efficiently by applying a simple mask. When executing a path expression such as `//ss:data`, the code for the name `ss:data` is compiled into the path expression, and the scan of the tree structure merely needs to perform a masked integer comparison against the namecodes held in the tree to see which nodes match.

The TinyTree per se does not hold any indexes, so path expressions are always evaluated directly by following the relevant axes. However, there are a few cases where indexes are constructed externally to the tree itself:

- An `<xsl:key>` element in XSLT causes an index to be built for a document the first time it is referenced in a `key()` function call

- A path expression of the form `//xyz` similarly causes an index to be built (or more correctly, an index entry that maintains a list of all the elements named *xyz*). This is again built lazily, the first time the construct is used: the system assumes that if the construct is used once, the chances are it will be used again.

- Saxon-SA, the commercial version of the product, indexes more aggressively. If a path expression of the form `/a/b/c[exp = $value]` is encountered, for example, then it will automatically be indexed using the `xsl:key` mechanism. This decision is made at compile time, but as with `<xsl:key>`, the index is actually constructed on first use.

Experienced XSLT users often advise against use of path expressions such as `//X`, preferring an explicit path such as `/A/B/C/X`. In fact, as the above discussion shows, `//X` isn't necessarily all that bad. In XML database products, in fact, `//X` often performs significantly better than an explicit path, because many XML databases maintain persistent indexes allowing rapid retrieval of all the elements with a given name, while an explicit path simply forces the system to do extra checking. This is a classic example of a case where the best way of writing an XPath expression depends on the characteristics of the product you are using, and it's therefore very difficult to give general advice.

Streaming

We've strayed into a discussion of how path expressions are evaluated, and this leads naturally into a discussion of streamed or pipelined execution. This is a technique widely used in all set-based languages (including, for example, relational databases and functional programming languages) to reduce memory usage.

The formal semantics for an expression such as `A/B/C/D[@id=3]` imply that the system should first form the result of the expression `A` (the set of all A children of the context node); then it should compute `A/B` (the set of all B children of those A children), then `A/B/C`, then `A/B/C/D`, and then finally the subset of this last result that satisfies the predicate `@id=3`. Clearly this naive strategy requires storing a large number of node-sets in memory, and this is costly: quite apart from the memory cost itself, allocating memory takes time, and releasing it (via garbage collection) also takes time. It's easy to see that the memory is unnecessary. Most XSLT and XQuery processors will execute an expression like this by following a navigation path that visits each of the candidate D elements and tests to see whether it should be included in the final result: this way, only enough memory is needed to hold the D elements that satisfy the predicate (and further streaming into higher-level expression may eliminate this node-set as well).

The streaming strategy works irrespective of the access path chosen for visiting the candidate D elements. In many products, Saxon included, the route to the Ds is the obvious one: a depth-first walk of the tree, considering only those nodes at each level with the required name. Other products might use a structure index to reach the D nodes, but the same rule applies: it's not necessary to build a list of the candidate D nodes in memory before applying the filter.

It's not only filter expressions that can be streamed. Similar techniques are available for many other expressions, and the processor is likely to go to some

lengths to use streaming wherever possible. For example, Saxon will use a streaming approach for the set union, intersection, and difference operators: after ensuring that the two input node-sets are sorted in document order, they are combined using a simple merge algorithm which itself delivers a stream of nodes in document order. (Sorting, of course, is one of the few operations that cannot be streamed: instead the strategy here is to avoid sorting wherever possible.)

One benefit of a streaming strategy is that it saves memory. The other big benefit is that it allows early exit. There are two very important cases in XPath where early exit comes into play: firstly with a positional predicate, such as `A/B[1]` or perhaps `A/B[position() < 5]`, and secondly with path expressions used existentially, in a construct such as `<xsl:if test="//@xml:space">`. In the first case, it's possible to quit the execution as soon as the first (or the first four) elements are found; in the second case, where the expression is testing for the existence of a node, it's possible to quit as soon as it's known that there is at least one node. Other cases where early exit is possible include expressions like `XYZ[A = 2]` where as soon as one *A* child is found to have the value 2, there is no need to look for any others.

Internally, there are two ways of implementing streaming, referred to as push and pull (this shouldn't be confused with the use of these terms to describe XSLT coding styles). With pull processing, the evaluation of an outer enclosing expression typically makes repeated calls on a `getNext()` method on each of its sub-expressions to get the next item in the sequence of results delivered by that sub-expression. With push processing, the control is in the other direction: the sub-expression makes a sequence of `write()` methods to add output to a data structure set up by the containing expression. Both methods avoid allocating memory to intermediate results, but with push processing it is less easy to do an early exit.

XSLT 1.0 had a fairly simple structure in which data was read from source documents using path expressions, and data was written to a result document using XSLT instructions. It was very natural to evaluate an XSLT 1.0 stylesheet using a main loop that read input data in pull mode by calling `getNext()` methods provided by the XPath processor, and then wrote output data in push mode by calling `write()` methods provided by the various XSLT instructions. XQuery and XSLT 2.0 allow much more flexible interleaving of input and output. One way of handling this is to do everything in "pull" mode: this is the way some XQuery engines work, if only because that was the tradition in relational database systems. Saxon in many cases can execute instructions in either pull or push mode, and the choice depends on the context in which the result of the expression is used.

Expression rewrites

Let's move on now to the work done by the XSLT or XQuery compiler to devise a query execution plan. The general approach is the same as for any language compiler:

- The source code is parsed, resulting in the creation of an expression tree to represent its logical structure. The resulting tree is known as an abstract syntax tree.

- Cross-references within the tree are resolved, for example variable references and function calls

- The tree is decorated with attributes identifying static properties of the sub-expressions: for example, whether or not a sub-expression is dependent on the context item

- The tree is scanned one or more times to identify sub-expressions that can be replaced with alternative, equivalent expressions that will (hopefully) be faster to execute.

One can divide the rewrites done in the last phase into two categories: rewrites that could have been done by the programmer (because the replacement expressions correspond to constructs available in the language), and those that could only be done by the optimizer, because they use specialized internal constructs. An example in the first category is replacing the expression `count(A)>3` by `exists(A[4])`. This rewrite is useful because it might avoid having to read a million A elements: it relies on the fact that `exists(A[4])` will do an early exit as soon as the fourth A element is found. An example in the second category is rewriting `A[position()=last()]` as `getLast(A)`. Here `getLast()` is an internal function that is available to the optimizer but not to the end user.

Saxon does many ad-hoc rewrites of expressions, as the above examples illustrate. The most important rewrites are probably those listed in the table below:

Rewrite	Explanation
Sort removal	The semantics of path expressions require a sort into document order, with elimination of duplicates. Saxon initially writes the abstract tree for a path expression as `sort(A\B)` where the \ operator, unlike /, does not itself force sorting. In the vast majority of path expressions, however, the sort is unnecessary. Saxon detects these cases and rewrites the expression as `A\B`.
Constant subexpressions	Where subexpressions can be evaluated at compile time, this is done. For example, this applies to constructor functions such as `xs:duration("P1Y")`.
Independent subexpressions in loops	Where a subexpression within a loop (for example `xsl:for-each`, or a FLWOR expression) does not depend on the controlling variables, the expression is evaluated outside the loop. (Or rather, it is evaluated on the first entry into the loop: if the loop is executed zero times, then the independent subexpression is not evaluated. This is important to avoid spurious errors.)
Decomposition of WHERE expressions	A WHERE clause in a FLWOR expression is broken up so that each term isevaluated as a predicate on the innermost "for" clause on which it depends. If the predicate doesn't depend on any of the "for" clauses, it becomes an "if" expression wrapping the entire FLWOR.

In many cases, of course, the rules determining whether a rewrite is possible are quite subtle, because of the need to preserve the semantics of the expression in error cases as well as non-error cases. Fortunately the XPath specification is quite liberal in the scope it gives a processor to skip error-detection where appropriate, but there are still corner cases that can inhibit optimization. An example is an expression such as `count(()|for $x in 1 to 5 return <a/>)` where the correct answer is 5: each iteration of the loop must produce a distinct `<a>` element, which means that an expression capable of constructing new nodes cannot be pulled out of a loop.

Many XSLT 1.0 processors perform XPath evaluation in a completely separate module. With this approach, there is essentially one expression tree for the XSLT instructions, and one tree for each XPath expression (though of course it is possible to view this as a single tree). Optimization in this case is likely to operate at the level of a single XPath expression. With XSLT 2.0, the XPath and XSLT parts of the language are now more composable. Saxon therefore now integrates the XSLT and XPath processing. Identical code is generated for the two constructs

```
<xsl:if test="a/b/c">
  <xsl:sequence select="x/y/z"/>
</xsl:if>
```

and

```
<xsl:sequence select="if (a/b/c) then x/y/z else ()"/>
```

However, Saxon still does XSLT optimization in two phases: first at the XPath expression level, then at the level of an XSLT template or function.

Type checking

One of the phases of processing the expression tree is a type checking phase. This checks, for each subexpression, whether the static types of the operands are known to be subtypes of the required type. There are several possible outcomes:

- The static type is a subtype of the required type (that is, all instances of the static type are instances of the required type). In this case no further action is needed.
- The static type overlaps the required type (some instances of the static type are instances of the required type). In this case Saxon generates "code" (more strictly, adds further expressions to the tree) to force a run-time type check.
- The static type and the required type are disjoint (no instances of the static type are instances of the required type). In this case, Saxon generates a type error at compile time, as permitted by the language specification.

While performing this analysis, Saxon also checks whether the supplied value can be converted to the required type using the permitted conversions such as atomization, casting of untypedAtomic values, and numeric promotion, and generates the extra expressions to perform these conversions where needed. The aim is to minimize the amount of redundant checking performed

at run-time, though there are some constructs where the interpretation is still very dynamic.

During the type checking process, Saxon also removes any existing type checking or conversion code that it finds to be redundant. This might be an unnecessary cast that was written by the user, or it might be code added by the optimizer in an earlier phase of processing, which can now be recognized as redundant because more type information has become available. For example, improved information about the type of a variable reference can become available after the variable declaration itself has been type-checked.

There's one special case where the static type and the required type overlap, but where the only instance common to both is the empty sequence. For example, this happens when the required type is `xs:integer*` (a sequence of integers) and the static type of the supplied value is `xs:string*`. It's not permissible to raise a compile-time error in this situation, because with a carefully-chosen input document, evaluation might succeed. However, it's almost certain that there's a user error here, so Saxon outputs a compile-time warning.

I always advise users to declare the types of variables and function arguments as explicitly as possible. The main benefit of this is that it makes the code more robust: errors are detected earlier, often at compile time. However, there is also a performance implication. Very often, declaring a type causes the type checking and perhaps conversion to be done once, perhaps on entry to a function, when it would otherwise be done repeatedly each time a parameter is used. Sometimes it means that Saxon can avoid run-time type checks entirely. It's also possible that declaring types causes an unnecessary check, which can slow down execution − this happens for example if you declare the result type of an XSLT template, when no subsequent operation actually depends on this type. In general, however, declaring types is beneficial to performance.

Join optimization

Some XQuery researchers and implementers appear to be so steeped in the traditions of relational databases that you might get the impression that optimization of joins is the number one priority for an optimizer. In fact, I think pure joins in XQuery are relatively unusual. XML is a hierarchic data model rather than a flat one, and most relationships are expressed through the tree structure of the XML document rather than by comparing primary keys with foreign keys. Even when value-based relationships are exploited in a query, it is often to produce hierarchic output, something like this:

```
for $c in //customers return
<customer name="{$c/name}"> {
  for $p in $c/customer-orders/product-code
  return <product code="{$p}"> {
      //product[@code=$p]/description
    } </product>
} </customer>
```

The fact that an output element is produced for each customer means that the system has relatively little choice in its execution strategy, compared with the SQL equivalent. Optimizing the join boils down to optimizing the filter expression `//product[@code=$p]`. Saxon-SA does this by generating a key definition equivalent to the XSLT construct

```
<xsl:key name="k1234" match="product" use="@code"/>
```

and replacing the predicate expression with the call `key('k1234', $p)`. (It's a bit more complicated than this because of error conditions, type conversion, and so on, but this is the basic idea.)

Saxon-SA will also use a similar technique if the filter expression is not searching an entire document, but some transient sequence. In this case an index will be generated for the transient sequence. This is very similar to the hash-join approach often used in relational databases.

XSLT users are accustomed to the idea that if they want lookups to go fast, they should define the indexes (and use them) explicitly. The database community has become accustomed to the doctrine that this is the job of the optimizer. To be honest, I'm not sure which approach is right. When I see programmers spending their time trying to rearrange a query in order to "trick" the optimizer into choosing a different execution strategy, and when I read books devoted to educating programmers in this black art, I tend to feel that the language has taken too much control away from the programmer. Anyway, Saxon-SA's XSLT processor gives you best of both worlds: you can define the keys "by hand" you wish, and if you don't, the optimizer will try to do the job on your behalf.

Tail recursion

There's one final optimization technique I would like to cover, namely tail call optimization. This technique is common in functional programming languages, which often use recursion for situations where other languages would use iteration. For example, it wouldn't be uncommon in a functional programming language to see a function to compute the sum of a sequence of numbers defined as

```
sum($x) -> if (empty($x) then 0 else $x[1] + sum(tail($x))
```

Because of the availability of "for" loops, recursion over a simple sequence isn't quite so common in XSLT and XQuery, and it will probably be less common in XSLT 2.0 than in XSLT 1.0. However, there are still many problems where recursive functions are necessary: an example is where the input contains a sequence of transaction records containing credits and debits, and the output is a bank statement in which the current balance after each transaction is shown alongside the transaction.

A naive execution strategy for a recursive function (or XSLT template: there's no difference conceptually) like the one above will create a new stack frame for each recursive call. You can tell when your XSLT processor doesn't optimize tail calls if this function works for sequences of say 300 items, but doesn't work for 1000, because it blows the stack limit.

The standard optimization for this in the functional programming world is to identify *tail calls* − that is, situations where the calling function exits immediately on return from the called function, without using the result, and rearrange the processing so that the function call is done after unwinding the stack, rather than before. The example function above doesn't include a tail call, because the logic uses the result of the recursive call: on return from the called function, it performs an addition. A good book on functional programming will give many ideas on how functions such as this can be rewritten (automatically or by hand) to take advantage of tail call optimization.

Sometimes tail call optimization is confined to recursive calls, but it can in fact be done for all calls. Sometimes it is done entirely by the compiler, by rewriting the recursive function as a loop. Saxon, however, does it using a mix of compile-time and run-time logic. Tail calls are identified and marked as such at compile time, but the reordering of the logic to unwind the stack before doing the call is a run-time operation.

Note that tail call optimization doesn't give any speed-up, it's done purely to conserve scarce stack space. Nevertheless, it can be important for programmers to be aware of cases where it is or isn't happening, and this will vary greatly from one processor to another.

Further reading

I've made a number of references to Saxon and Saxon-SA in this paper. Saxon is an open-source XSLT 2.0 and XQuery 1.0 processor available at `http://saxon.sf.net/`. Saxon-SA is a commercial version of the same product available from `http://www.saxonica.com/`. The main differences are that Saxon-SA is schema-aware (hence the name) and − more relevantly to this paper − that it has a more sophisticated optimizer.

There's an outline of the internal architecture of Saxon at
http://www-128.ibm.com/developerworks/library/x-xslt2/.
Although this was written a while ago, the basic structure hasn't changed.

This paper can be regarded as an update of the paper I published at XML Europe 2004, entitled XSLT and XPath Optimization: see
http://www.idealliance.org/papers/dx_xmle04/papers/02-03-02/02-03-02.html.
Some of the optimizations done by Saxon are described in greater detail in that paper than here: on the other hand, Saxon has moved on in the meantime so the information can't be considered up-to-date.

Most of the academic work on XSLT and XQuery optimization is focused on very specific techniques. Some of these techniques give dramatic improvements, but are applicable only to a tiny number of real queries and stylesheets. Despite the greater maturity and industrial exploitation of XSLT, more academic work has been done on XQuery and XPath, perhaps because these language are smaller and more amenable to theoretical analysis, or perhaps because XQuery fits naturally into the domain of interest of the database research community, while XSLT doesn't have a similar natural home in the academic space. Beware when reading academic papers that researchers have the luxury of choosing a subset of the language to work on. This means, for example, that you will find papers on streaming evaluation of XPath expressions that ignore positional predicates entirely.

XML/RDF Query by Example

Eric van der Vlist (Dyomedea)

Context

The French Institut national de la statistique et des études économiques (INSEE) needs a XML vocabulary to expose the consolidated content of several LDAP databases.

LDAP can be seen as the combination of a graph of classes, objects and a hierarchy similar to traditional file system directories. This combination can be compared to what we would obtain by attaching RDF triples to each directory of a file system.

LDAP as a tree

A first approach to serializing LDAP as XML is to focus on its hierarchical dimension and to map this hierarchy into an XML tree of elements and attributes. This could lead to something such as:

```
<?xml version="1.0" encoding="utf-8"?>
<annuaire>
  <fr>
    <insee>
      <Personnes>
        <inseePerson dn="uid=R3D2,ou=Personnes,o=insee,c=fr">
          <telephoneNumber>0123456789</telephoneNumber>
          <cn>Laurent Dupondt</cn>
          <inseeFonction dn="uid=GS10,ou=Fonctions,o=insee,c=fr"/>
          <inseeTimbre>SED</inseeTimbre>
          <uid>R3D2</uid>
          <inseeNomGIP>Dupondt</inseeNomGIP>
          <inseeDomaineNT>DR40A</inseeDomaineNT>
          <sn>Dupondt</sn>
          <inseeGroupeDefaut>MVS:SE40</inseeGroupeDefaut>
          <employeeType>Interne</employeeType>
          <inseePrenomGIP>LAURENT</inseePrenomGIP>
          <inseeServeurExchange>S40X01</inseeServeurExchange>
          <roomNumber>113</roomNumber>
          <inseeGrade>Attach
          <personalTitle>M</personalTitle>
          <mail>laurendt.dupondt@pas-de-pourriel.fr</mail>
```

```
          <givenName>Laurent</givenName>
          <inseeUnite dn="ou=DR54-SED,ou=Unit%C3%A9s,o=insee,c=fr"/>
          <objectClass>top</objectClass>
          <objectClass>person</objectClass>
          <objectClass>organizationalPerson</objectClass>
          <objectClass>inetOrgPerson</objectClass>
          <objectClass>InseePerson</objectClass>
          <employeeNumber>12345</employeeNumber>
          <ou>DR54-SED</ou>
        </inseePerson>
      </Personnes>
    </insee>
  </fr>
</annuaire>
```

In this example, we have mapped the root of the LDAP structure into an *annuaire* document element and each branch and object of the LDAP directory into an XML element.

This approach gives the primary role to the hierarchical facet of LDAP and does not natively expose its graph facet that needs to be implemented using some kind of links such as ID/IDREF, XLink or application specific links: in this example, we had chosen to use the LDAP "distinguished names" (dn) as identifiers.

LDAP as a graph

This focus on the hierarchical dimension happens to be very far from the most typical use of LDAP repositories at the INSEE: the structure adopted by the INSEE is on the contrary rather flat with many objects under each node. For instance, all the persons are stored under the same node (`ou=Personnes`, `o=insee`, `c=fr`). To take this pattern into account, the natural thing was thus to give the primary role to the graph dimension.

That being said, there is a standard vocabulary for describing graphs in XML and this vocabulary is called RDF.

While it sounded very natural to use RDF to serialize LDAP in XML, I had to take into account the fact that the initial request was to define a XML vocabulary and that I had to maintain this vocabulary acceptable for XML heads.

The context was thus quite similar to the one in which we were when we've published the RSS 1.0 specification: defining a RDF vocabulary that is acceptable by both XML and RDF heads.

I think that being acceptable by RDF heads is the easiest part since this basically means that the vocabulary has to be conform to the RDF/XML Syntax Specification: there is a clear specification to comply with.

Being acceptable by XML heads means that the XML vocabulary has to "look" natural to people who do not know RDF and that the so called "RDF tax" has to be minimized.

Concretely, that involves choosing between the many options available to express the same set of triples in RDF/XML the one should be the most straightforward and least surprising to someone who knows XML but do not know RDF.

We ended up with a vocabulary that looks like:

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://xml.insee.intra/schema/annuaire/"
  xmlns:a="http://xml.insee.intra/schema/annuaire/"
  xmlns:l="http://xml.insee.intra/schema/ldap/">
<inseePerson
  rdf:about="http://xml.insee.fr/ldap/uid=R2D2,ou=Personnes,
  o=insee,c=fr">
  <l:dn>uid=R2D2,ou=Personnes,o=insee,c=fr</l:dn>
  <l:parent rdf:resource="http://xml.insee.fr/ldap/ou=Personnes,
  o=insee,c=fr"/>
  <l:ancestorOrSelf
    rdf:resource="http://xml.insee.fr/ldap/uid=R2D2,ou=Personnes,
    o=insee,c=fr"/>
  <l:ancestorOrSelf
    rdf:resource="http://xml.insee.fr/ldap/ou=Personnes,o=insee,c=fr"/>
  <l:ancestorOrSelf rdf:resource="http://xml.insee.fr/ldap/o=insee,c=fr"/>
  <l:ancestorOrSelf rdf:resource="http://xml.insee.fr/ldap/c=fr"/>
  <l:ancestorOrSelf rdf:resource="http://xml.insee.fr/ldap/"/>
  <telephoneNumber>0383918546</telephoneNumber>
  <cn>Laurent Dupondt</cn>
  <inseeFonction
    rdf:resource="http://xml.insee.fr/ldap/uid=GS10,ou=Fonctions,
    o=insee,c=fr"/>
  <inseeTimbre>SED</inseeTimbre>
  <uid>R2D2</uid>
  <inseeNomGIP>DUPONDT</inseeNomGIP>
  <inseeDomaineNT>DR40A</inseeDomaineNT>
  <sn>Dupondt</sn>
  <inseeGroupeDefaut>MVS:SE40</inseeGroupeDefaut>
  <employeeType>Interne</employeeType>
  <inseePrenomGIP>LAURENT</inseePrenomGIP>
  <inseeServeurExchange>S40X01</inseeServeurExchange>
  <roomNumber>113</roomNumber>
  <inseeGrade>Attach
  <personalTitle>M</personalTitle>
  <mail>laurendt.dupondt@pas-de-pourriel.fr</mail>
  <givenName>Laurent</givenName>
  <inseeUnite
    rdf:resource="http://xml.insee.fr/ldap/ou=DR54-SED,ou=Unit%C3%A9s,
    o=insee,c=fr"/>
```

```
  <objectClass>top</objectClass>
  <objectClass>person</objectClass>
  <objectClass>organizationalPerson</objectClass>
  <objectClass>inetOrgPerson</objectClass>
  <objectClass>InseePerson</objectClass>
  <employeeNumber>12345</employeeNumber>
  <ou>DR54-SED</ou>
</inseePerson>
```

The distinguished names have been translated into URIs usable as RDF identifiers and the hierarchy has been expressed using `<l:dn>`, `<l:parent>` and `<l:ancestorOrSelf>` elements.

The RDF tax has been limited to using these URIs as identifiers in `rdf:about` and `rdf:resource` attributes and using `<rdf:RDF>` as the document element.

The choice of these elements has been dictated by the fact that they match the three ways to define the scope when doing LDAP queries and we'll come back to that point later on.

LDAP as a graph and as a tree

When I have started writing this introduction, it suddenly occurred to me that LDAP wasn't the only technology that superposes a graph and a tree and that this was also the case of RDF/XML documents that are both a graph if you read them with a RDF parser and a tree if you read them with a XML parser.

There is thus a third way which would be to use the XML hierarchy to describe the hierarchical dimension of LDAP and to use the RDF features to describe the graph dimension of LDAP.

This third way could look like:

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://xml.insee.intra/schema/annuaire/"
  xmlns:a="http://xml.insee.intra/schema/annuaire/"
  xmlns:l="http://xml.insee.intra/schema/ldap/">
<l:root>
  <l:node rdf:parseType="Resource">
    <c>fr</c>
    <l:node rdf:parseType="Resource">
      <o>insee</o>
      <l:node rdf:parseType="Resource">
        <ou>Personnes</ou>
        <inseePerson rdf:parseType="Resource">
          <telephoneNumber>0123456789</telephoneNumber>
          <cn>Laurent Dupondt</cn>
          <inseeFonction
```

```
            rdf:resource="http://xml.insee.fr/ldap/uid=GS10,ou=Fonctions,
            o=insee,c=fr"/>
            <inseeTimbre>SED</inseeTimbre>
            <uid>R2D2</uid>
            <inseeNomGIP>DUPONDT</inseeNomGIP>
            <inseeDomaineNT>DR40A</inseeDomaineNT>
            <sn>Dupondt</sn>
            <inseeGroupeDefaut>MVS:SE40</inseeGroupeDefaut>
            <employeeType>Interne</employeeType>
            <inseePrenomGIP>LAURENT</inseePrenomGIP>
            <inseeServeurExchange>S40X01</inseeServeurExchange>
            <roomNumber>113</roomNumber>
            <inseeGrade>Attach
            <personalTitle>M</personalTitle>
            <mail>laurendt.dupondt@pas-de-pourriel.fr</mail>
            <givenName>Laurent</givenName>
            <inseeUnite
              rdf:resource="http://xml.insee.fr/ldap/ou=DR54-SED,
              ou=Unit%C3%A9s,o=insee,c=fr"/>
            <objectClass>top</objectClass>
            <objectClass>person</objectClass>
            <objectClass>organizationalPerson</objectClass>
            <objectClass>inetOrgPerson</objectClass>
            <objectClass>InseePerson</objectClass>
            <employeeNumber>12345</employeeNumber>
            <ou>DR54-SED</ou>
          </inseePerson>
        </l:node>
      </l:node>
    </l:node>
  </l:root>
</rdf:RDF>
```

Being a brand new idea coming very late after the vocabulary has been more or less finalized, this snippet isn't something polished at all but just there to give you an idea of where that could lead!

The problem

So far, so good and I think that we've made a decent job in our quest to define a "low RDF tax" RDF/XML vocabulary. This has been a fun small project which could justify a proposal to another XML conference but probably not to Extreme!

The problem became more challenging when we've started to address the second request: define an XML vocabulary to express queries against the LDAP repository using a syntax that was coherent with what we'd done so far.

We had basically three types of standards on which we could have relied...

LDAP filters (RFC 2254)

The first one was the RFC 2254 that defines a text format to express LDAP filters.

LDAP queries are specified by defining a search base and a LDAP filter. For instance, the filter `"(inseeRoleApplicatif=RP$$P$SUP*)"` search for objects which attribute `"inseeRoleApplicatif"` matching the value `"RP$$P$SUP*"`. The star character `*` being a wildcard that replaces any character, this match means that the attribute has to begin with `"RP$$P$SUP"`.

As a vocabulary, we could have embedded such filters into a wrapper element together with the search base:

```
<?xml version="1.0" encoding="utf-8"?>
<ldapSearch>
  <base scope="subTree">Personnes,o=insee,c=fr</base>
  <filter>(inseeRoleApplicatif=RP$$P$SUP*)</filter>
</ldapSearch>
```

That would have been handy for people familiar with LDAP, but we wanted to provide something usable for users knowing our RDF/XML vocabulary without requiring that they learn LDAP is they didn't know it.

W3C XQuery

XQuery was another natural candidate. The same query would give:

```
xquery version "1.0";
declare namespace a = "http://xml.insee.intra/schema/annuaire/";
<rdf:RDF xmlns:rdf="#">
{
  for $x in /rdf:RDF/a:inseePerson
  where $x[a:inseeRoleApplicatif[ starts-with(., 'RP$$P$SUP') ]]
  return $x}
</rdf:RDF>
```

That's handy for XML heads who see the RDF/XML serialization as pure XML but where are the triples that RDF heads see?

Furthermore, XQuery is fairly complex and this complexity would create two issues in our context:

- Users would have to learn XQuery.
- Implementing an XQuery engine on top of the LDAP gateway would be a daunting task.

We've thus considered that while XQuery didn't preserve the balance between XML and RDF that we had carefully built, it was also be an overkill!

W3C SPARQL (or any other RDF query language)

I am not a SPARQL guru, so please be kind with me if the following snippet contains errors, but the same query expressed in SPARQL would look like:

```
PREFIX : http://xml.insee.intra/schema/annuaire/
SELECT ?inseePerson
WHERE {
  ?inseePerson rdf:type :inseePerson;
  ?inseePerson :inseeRoleApplicatif ?inseeRoleApplicatif.
  FILTER regex(str(?inseeRoleApplicatif), "^RP$$P$SUP ")
}
```

Some RDF heads would be much happier with this expression but not all of them would agree that this is the way to go: RDF query languages are still very young and many different approaches have been proposed.

Furthermore, XML heads would be totally lost, users would have to learn SPARQL (and before that they would have to learn how to distinguish triples in XML fragments) and implementing a SPARQL engine on top of our LDAP gateway would be almost as challenging as implementing an XQuery engine!

Home bred

After having rejected the three standards on which we could have relied, we came to the conclusion that we would have to create our own query language that should be both coherent with the syntax used by our RDF/XML vocabulary and easy to use by non-guru-XML-literates.

These requirements rang two bells...

When I started giving presentations on XML schema languages, I used to say that XML schemas are more complex than the instances they describe and ended up proposing Examplotron, a schema language based on examples.

Before that, one of the easiest database I have ever used had been Borland's Paradox, and its query language was a Query By Example (QBE) language.

Query by example (QBE)

In his "Principles of Database Systems", Jeffrey D. Ullman defines QBE as a "domain calculus language" developed at IBM that "contains a number of features not present in relational algebra or calculus, or in any of the implemented query languages we have discussed."

The QBE we had in mind is not that ambitious, but that statement is a good indication that, while we would probably limit the features we would implement, we would not be limited by the method itself.

Since we are not designing a user interface but an XML/RDF query language, our QBE would naturally be based on a RDF/XML syntax that would

be as close as possible to the XML/RDF vocabulary used to formalize the results. Being a XML vocabulary, it was also natural enough to use a specific namespace to distinguish the instructions of the query language from the actual examples.

Also, as already mentioned, one of our goals is to define a query language that can be read both as XML documents but also as valid XML/RDF models and processed both from its XML infoset or from its sets RDF triples. Basic query structure

In IBM's original QBE, the same set of columns were used to indicate the list of columns to print and the conditions on the columns. The distinction between these two features was done using different sets of operators.

This reduces the "legibility as examples" of the values placed in the columns and we have considered that it would be more readable to separate the definition of what needs to be returned from the definition of the conditions.

This led us toward a structure that looks like the classical `select X from Y` structure of a SQL select statement:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:q="http://xml.insee.intra/schema/qbe/">
  <q:select>
    <q:what>
      .../...
    </q:what>
    <q:where>
      .../...
    </q:where>
  </q:select>
</rdf:RDF>
```

Typical XML/RDF documents have a relatively flat structure and we can follow this style to express the what and where clauses of our request:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://xml.insee.intra/schema/annuaire/"
  xmlns:q="http://xml.insee.intra/schema/qbe/">
  <q:select>
    <q:what rdf:resource="#what"/>
    <q:where rdf:resource="#where"/>
  </q:select>
  <inseePerson rdf:ID="what"/>
  <inseePerson rdf:ID="where">
    <mail>jean.dupondt@insee.fr</mail>
  </inseePerson>
</rdf:RDF>
```

The what clause is optional and when it is omitted, the query returns the complete element specified in the where clause (this would be equivalent to a SQL `select * from ...` query). In this example, that would give:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://xml.insee.intra/schema/annuaire/"
  xmlns:q="http://xml.insee.intra/schema/qbe/">
  <q:select>
    <q:where rdf:resource="#where"/>
  </q:select>
  <inseePerson rdf:ID="where">
    <mail>jean.dupondt@insee.fr</mail>
  </inseePerson>
</rdf:RDF>
```

In these two equivalent queries, we are requesting *inseePerson* elements that have mail addresses equal to `jean.dupondt@insee.fr`.

While this flat style looks XML/RDFish, it is not that usual to XML heads and we accept an alternative Russian doll syntax (using what RDF heads call a blank node):

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://xml.insee.intra/schema/annuaire/"
  xmlns:q="http://xml.insee.intra/schema/qbe/">
  <q:select>
    <q:what>
      <inseePerson/>
    </q:what>
    <q:where>
      <inseePerson>
        <mail>jean.dupondt@insee.fr</mail>
      </inseePerson>
    </q:where>
  </q:select>
</rdf:RDF>
```

This alternative syntax produces a very similar set of RDF triples and we consider it as strictly equivalent.

The last alternative for this same query would be to omit the what clause:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://xml.insee.intra/schema/annuaire/"
  xmlns:q="http://xml.insee.intra/schema/qbe/">
  <q:select>
    <q:where>
      <inseePerson>
        <mail>jean.dupondt@insee.fr</mail>
      </inseePerson>
    </q:where>
  </q:select>
</rdf:RDF>
```

Introducing functions

What happens if we want to express something a little bit more complex and write that we want the `<inseePerson/>` whose email address ends with `@insee.fr`?

Since we are in design mode, we have many possibilities to implement that feature.

The first one which is the one that has been adopted by IBM's original QBE would be to use functions in literals:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://xml.insee.intra/schema/annuaire/"
  xmlns:q="http://xml.insee.intra/schema/qbe/">
  <q:select>
    <q:where>
      <inseePerson>
        <mail>ends-with(@insee.fr)</mail>
      </inseePerson>
    </q:where>
  </q:select>
</rdf:RDF>
```

I don't like this solution for a couple of reasons:

- The functions are called using a plain text syntax that requires a parsing which is not obvious to do with "pure" XML tools such as XSLT 1.0.
- If you look at this document with RDF glasses, the triple "*_:genid3* <`http://xml.insee.intra/schema/annuaire/mail`> `ends-with(@insee.fr)`" should mean "*_:genid3* has a mail property equal to `ends-with(@insee.fr)`". Here we are interpreting it as "*_:genid3* has a mail property matching the condition `ends-with(@insee.fr)`" and that just doesn't sound right.

The way to avoid these plain text function calls is to replace them by XML elements (or RDF triples if you prefer). The first (naive) attempt to do so would be to write:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://xml.insee.intra/schema/annuaire/"
  xmlns:q="http://xml.insee.intra/schema/qbe/">
  <q:select>
    <q:what>
      <inseePerson/>
    </q:what>
    <q:where>
      <inseePerson>
        <mail>
          <q:ends-with>@insee.fr</q:ends-with>
```

```
        </mail>
      </inseePerson>
    </q:where>
  </q:select>
</rdf:RDF>
```

This fine looks nice to XML heads, but makes Jena scream:

```
_:jA3 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://xml.insee.intra/schema/qbe/ends-with> .
Error:  file:///home/vdv/cvs-private/presentations/en/extreme2005/
  query4.rdf[11:43]:
E202 Expected whitespace found:  '@insee.fr'.  Maybe a missing
rdf:parseType='Literal', or a striping problem.
```

Why is that? In fact, the rules that bind XML nodes to RDF triples in the XML/RDF syntax are well designed enough that they've produced what we'd expected so far but they're biting us in this last example! Let's think more about these rules...

When we write:

```
<rdf:RDF>
  <foo>
    <bar>
      <baz>
        <bat>XXX</bat>
      </baz>
    </bar>
  </foo>
<rdf:RDF>
```

the different elements are not treated as equal by the XML/RDF binding rules and the triples that a RDF parser will extract from this document are:

```
Subj.   Pred.   Obj.
<foo>   <bar>   <baz>
<baz>   <bat>   "XXX"
```

`<foo>` and `<baz>` are resources that are used as subject and cannot have literals directly attached to them while `<bar>` and `<bat>` are resources that are used as predicates and can have literals directly attached to them.

In our naive attempt that made Jena scream, we've attached the literal `@insee.fr` to the resource `<q:ends-with>` that was in a position where it was considered as a subject.

There are a couple of solutions to work around this error.

The first one is to use the `rdf:parseType` attribute to specify that `<mail>` should not be considered only as a predicate but also as a resource that can be the subject of a triple:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://xml.insee.intra/schema/annuaire/"
  xmlns:q="http://xml.insee.intra/schema/qbe/">
  <q:select>
    <q:where>
      <inseePerson>
        <mail rdf:parseType="Resource">
          <q:ends-with>@insee.fr</q:ends-with>
        </mail>
      </inseePerson>
    </q:where>
  </q:select>
</rdf:RDF>
```

The trick, here, is that since `<mail>` is now a subject, `<q:ends-with>` becomes a predicate that can have a literal directly attached to it.

Technically speaking, this does the job of making Jena happy, but I don't like it that much since it requires a good understanding of the XML/RDF binding rule to be able to tell when and where you need to add `rdf:parseType="Resource"` attributes. Requiring such an understanding doesn't meet our goal to define a query language that is understandable by "pure XML heads".

The second solution (which we've adopted) is to add a level of hierarchy:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://xml.insee.intra/schema/annuaire/"
  xmlns:q="http://xml.insee.intra/schema/qbe/">
  <q:select>
    <q:where>
      <inseePerson>
        <mail>
          <q:conditions>
            <q:ends-with>@insee.fr</q:ends-with>
          </q:conditions>
        </mail>
      </inseePerson>
    </q:where>
  </q:select>
</rdf:RDF>
```

The trick here is that since `<q:ends-with>` needs to be a predicate, we've added `<q:conditions>` to serve has its subject. RDF parsers are happy and that seems to be cleaner than the previous workaround because that does not change the nature of `<mail>` which is and stays a predicate.

Of course, the same query can be written with a flat style with or without an explicit what clause:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://xml.insee.intra/schema/annuaire/"
  xmlns:q="http://xml.insee.intra/schema/qbe/">
```

```
  <q:select>
    <q:where rdf:resource="#where"/>
  </q:select>
  <inseePerson rdf:ID="where">
    <mail>
      <q:conditions>
        <q:ends-with>@insee.fr</q:ends-with>
      </q:conditions>
    </mail>
  </inseePerson>
</rdf:RDF>
```

Joins

So far, we've queried the `<mail>` predicate of `<inseePerson>` resources. Now, let's see what happens if we want to get `<inseePerson>` resources that have an `<inseeUnite>` which `<ou>` predicate is equal to `DG75-C460`. In XML, that would mean that we are looking for situations such as:

```
<rdf:RDF>
  <insee:Person>
    <inseeUnite rdf:resource="xxx"/>
  </insee:Person>
  <inseeUnite rdf:about="xxx">
    <ou>DG75-C460</ou>
  </inseeUnite>
</rdf:RDF>
```

But, since we are also thinking to RDF heads, such a situation needs to be equivalent to:

```
<rdf:RDF>
  <insee:Person>
    <inseeUnite rdf:parseType="Resource">
      <ou>DG75-C460</ou>
    </inseeUnite>
  </insee:Person>
</rdf:RDF>
```

And we'll leave people use indifferently the first or the second style. Following a Russian doll style, our query can thus be written:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://xml.insee.intra/schema/annuaire/"
  xmlns:q="http://xml.insee.intra/schema/qbe/">
  <q:select>
    <q:where>
      <inseePerson>
        <inseeUnite rdf:parseType="Resource">
          <ou>DG75-C460</ou>
```

```
        </inseeUnite>
      </inseePerson>
    </q:where>
  </q:select>
</rdf:RDF>
```

Why do we accept `rdf:parseType="Resource"` in this case when we've rejected it in the previous section? That's a good question...

In the previous section, we would have been adding `rdf:parseType="Resource"` in a `<mail>` element if and only if we wanted to add a condition and that seemed arbitrary. Here we are adding `rdf:parseType="Resource"` to `<inseeUnite>` because we want to specify that `<inseeUnite>` is a resource and not only a property and that should be much easier to swallow by anyone, XML or RDF head!

Join and condition

So far so good, but how does that scale? Can we, for instance, add a condition after a join and search the `<inseePerson>` which `<inseeUnite>` have a `<ou>` that starts with `DDEQ`?

That's easy enough and what we've seen so far works well together:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://xml.insee.intra/schema/annuaire/"
  xmlns:q="http://xml.insee.intra/schema/qbe/">
  <q:select>
    <q:where>
      <inseePerson>
        <inseeUnite rdf:parseType="Resource">
          <ou>
            <q:conditions>
              <q:starts-with>DG75</q:starts-with>
            </q:conditions>
          </ou>
        </inseeUnite>
      </inseePerson>
    </q:where>
  </q:select>
</rdf:RDF>
```

What about several conditions?

What if I want to meet two different conditions?

To keep it simple we've adopted the principle that by default, all the conditions grouped under a single where clause are anded:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:q="http://xml.insee.intra/schema/qbe/"
```

```
  xmlns="http://xml.insee.intra/schema/annuaire/">
  <q:select>
    <q:where>
      <inseePerson>
        <cn>
          <q:conditions>
            <q:contains>a</q:contains>
          </q:conditions>
        </cn>
        <mail>
          <q:conditions>
            <q:contains>o</q:contains>
          </q:conditions>
        </mail>
      </inseePerson>
    </q:where>
  </q:select>
</rdf:RDF>
```

This simple rule is enough to express set of conditions that must all be verified. For other cases, we have introduced two sets of elements that need to be used in conjunction:

- `<q:if>` and `<q:if-not>` are containers for sets of conditions that must or must not be verified.
- `<a:any>` and `<q:all>` are containers for sets of conditions that perform a logical "or" or a logical "and" between these conditions.

An `<q:if>` or a `<q:if-not>` must always be used together with an embedded `<q:all>` or `<q:any>`. This allows to represent the four possible combinations and also to meet the XML/RDF triple binding rules without having to require additional `rdf:parseType` attributes.

The last query is thus a shortcut for:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:q="http://xml.insee.intra/schema/qbe/"
  xmlns="http://xml.insee.intra/schema/annuaire/">
  <q:select>
    <q:where>
      <inseePerson>
        <q:if>
          <q:all>
            <cn>
              <q:conditions>
                <q:contains>a</q:contains>
              </q:conditions>
            </cn>
            <mail>
              <q:conditions>
```

```
              <q:contains>o</q:contains>
            </q:conditions>
          </mail>
        </q:all>
      </q:if>
    </inseePerson>
  </q:where>
 </q:select>
</rdf:RDF>
```

The other combinations would be:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:q="http://xml.insee.intra/schema/qbe/"
  xmlns="http://xml.insee.intra/schema/annuaire/">
  <q:select>
    <q:where>
      <inseePerson>
        <q:if-not>
          <q:all>
            <cn>
              <q:conditions>
                <q:contains>a</q:contains>
              </q:conditions>
            </cn>
            <mail>
              <q:conditions>
                <q:contains>o</q:contains>
              </q:conditions>
            </mail>
          </q:all>
        </q:if-not>
      </inseePerson>
    </q:where>
  </q:select>
</rdf:RDF>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:q="http://xml.insee.intra/schema/qbe/"
  xmlns="http://xml.insee.intra/schema/annuaire/">
  <q:select>
    <q:where>
      <inseePerson>
        <q:if>
          <q:any>
            <cn>
              <q:conditions>
                <q:contains>a</q:contains>
              </q:conditions>
            </cn>
            <mail>
              <q:conditions>
                <q:contains>o</q:contains>
```

```
            </q:conditions>
          </mail>
        </q:any>
      </q:if>
    </inseePerson>
  </q:where>
 </q:select>
</rdf:RDF>
```

and

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:q="http://xml.insee.intra/schema/qbe/"
  xmlns="http://xml.insee.intra/schema/annuaire/">
  <q:select>
    <q:where>
      <inseePerson>
        <q:if-not>
          <q:any>
            <cn>
              <q:conditions>
                <q:contains>a</q:contains>
              </q:conditions>
            </cn>
            <mail>
              <q:conditions>
                <q:contains>o</q:contains>
              </q:conditions>
            </mail>
          </q:any>
        </q:if-not>
      </inseePerson>
    </q:where>
  </q:select>
</rdf:RDF>
```

The same `<q:if>`, `<q:if-not>`, `<q:any>` and `<q:all>` elements can be used with the same meaning under `<q:conditions>` elements, for instance:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:q="http://xml.insee.intra/schema/qbe/"
  xmlns="http://xml.insee.intra/schema/annuaire/">
  <q:select>
    <q:where>
      <inseePerson>
        <mail>
          <q:conditions>
            <q:if-not>
              <q:all>
                <q:contains>a</q:contains>
                <q:contains>@insee.fr</q:contains>
                <q:starts-with>laurent</q:starts-with>
```

```
            </q:all>
          </q:if-not>
        </q:conditions>
      </mail>
    </inseePerson>
  </q:where>
 </q:select>
</rdf:RDF>
```

Search base

We've seen how what LDAP would call query filters can be expressed using our query language, what about LDAP search base?

I have already said that we had added the following properties to our RDF/XML vocabulary to support this feature:

```
<l:dn>uid=R2D2,ou=Personnes,o=insee,c=fr</l:dn>
<l:parent rdf:resource="http://xml.insee.fr/ldap/ou=Personnes,o=insee,c=fr"/>
<l:ancestorOrSelf
  rdf:resource="http://xml.insee.fr/ldap/uid=R2D2,
  ou=Personnes,o=insee,c=fr"/>
<l:ancestorOrSelf
  rdf:resource="http://xml.insee.fr/ldap/ou=Personnes,o=insee,c=fr"/>
<l:ancestorOrSelf rdf:resource="http://xml.insee.fr/ldap/o=insee,c=fr"/>
<l:ancestorOrSelf rdf:resource="http://xml.insee.fr/ldap/c=fr"/>
<l:ancestorOrSelf rdf:resource="http://xml.insee.fr/ldap/"/>
```

These properties will be used in our queries like any other property and will match the three different types of search base.

The first one, most commonly used at the INSEE is a search base with a scope equal to "subTree". That means that the search is performed in the object itself and all its descendants. To express that, we will use the `<l:ancestorOrSelf>` property, for instance:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:q="http://xml.insee.intra/schema/qbe/"
  xmlns="http://xml.insee.intra/schema/annuaire/"
  xmlns:l="http://xml.insee.intra/schema/ldap/">
  <q:select>
    <q:where>
      <inseePerson>
        <l:ancestorOrSelf rdf:parseType="Resource">
          <l:dn>o=insee,c=fr</l:dn>
        </l:ancestorOrSelf>
      </inseePerson>
    </q:where>
  </q:select>
</rdf:RDF>
```

The second case is a scope equal to “oneLevel” and this one means that the search is to be performed among the immediate children of the search base. To express that, we will use the `<l:parent>` property:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:q="http://xml.insee.intra/schema/qbe/"
  xmlns="http://xml.insee.intra/schema/annuaire/"
  xmlns:l="http://xml.insee.intra/schema/ldap/">
  <q:select>
    <q:where>
      <inseePerson>
        <l:parent rdf:parseType="Resource">
          <l:dn>o=insee,c=fr</l:dn>
        </l:parent>
      </inseePerson>
    </q:where>
  </q:select>
</rdf:RDF>
```

The third and last case is a scope equal to “base” and in that case, the search has to be performed on the object itself. For that last case, we will be using the `<l:dn>` property:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:q="http://xml.insee.intra/schema/qbe/"
  xmlns="http://xml.insee.intra/schema/annuaire/"
  xmlns:l="http://xml.insee.intra/schema/ldap/">
  <q:select>
    <q:where>
      <inseePerson>
        <l:dn>o=insee,c=fr</l:dn>
      </inseePerson>
    </q:where>
  </q:select>
</rdf:RDF>
```

Current status

The query language is now more or less stabilized. A first proof of concept has been implemented with Orbeon PresentationServer that transforms these queries into XQuery queries using a XSLT 1.0 transformation and queries an XML database (eXist).

The final implementation is now under development. It will transform these queries into LDAP searches sent to the LDAP repository and serialize the answers as RDF/XML.

The main difficulty in this implementation is to support the joins that do exist in LDAP filters and have to be performed by the gateway itself.

Conclusion

While this project is being developed for a very specific target, the design decisions have not been influenced by the context and I believe that this experience could be generalized to any project needing a query language matching these three conditions :

- The source to query is expressed (or can be expressed) as RDF/XML.
- The query language needs to make sense for both XML and RDF heads.
- The query language needs to look like the source to query (QBE).

I'd like to thank my customer (INSEE) for having funded this work and Franck Cotton who is leading this project.

xq2xml: Transformations on XQueries

David Carlisle (NAG Limited)

Abstract

xq2xml consists of a set of XSLT2 stylesheets to manipulate XQuery expressions. The two main applications are converting to XQueryX and XSLT.

The initial parsing of the XQuery expression is performed by the W3C Working Group test parser (maintained by Scott Boag). This presents an XML view of the parse tree, which may then be manipulated by XSLT.

The transformation to XSLT2 was mainly designed to highlight the similarities and differences between XQuery and XSLT, and this talk will discuss the relationship between the two languages. However it may also be seen as the first stage in an XQuery implementation, which is completed by executing the generated XSLT with an XSLT2 processor. xq2xml has been regularly tested in this mode with the XQuery test suite.

Other transformations are provided to (for example) rewrite expressions using the optional axes in XQuery to equivalent expressions that do not use optional axes.

More details, and full source of xq2xml, may be obtained from `http://monet.nag.co.uk/xq2xml/index.html`.

Transformations on Queries

XQuery is a new Query language being specified by a W3C Working Group which is designed to Query XML documents. It is strongly related to XSLT, and has been developed in parallel with XSLT2 and XPath2.

One of the strengths of XSLT is that it utilises an XML syntax which allows XSLT files to be used as data for queries and transformations. Unfortunately XQuery uses a non-XML syntax which means that it is not directly available to be used as the source of such queries. There is an XML Syntax for XQuery, XQueryX, but this is not widely supported, in fact the xq2xml suite was (as far as I am aware) the first tool set to offer any support for generating XQueryX.

So the original aim of xq2xml was to offer an XML view of an XQuery expression to allow queries and transformations on XQuery source files. Before looking at the details of the implementation, it is probably worthwhile to list some of the kinds of operations that one may wish to do on a Query (or set of Queries) considered as data:

Query for specific language constructs.

For example one may wish to query for extension pragmas, or use of the `doc()` function, which may not be safe in certain environments, and either transform the query not to use these constructs, or to warn of their use.

Another example (that is provided as an example in the distribution) is to transform any Query using the optional axes that are in XPath but may not be supported by all XQuery implementations into a Query that does not use these axes.

Query for the declaration of external variables which may be declared in a Query.

Any such variables need to be initialised by the external environment before executing the Query, and in many environments it is useful to be able to extract the variable names (and types if declared) from the Query text.

Convert to another syntax.

A conversion to XQueryX is provided as an example.

Convert to a different language.

A conversion to XSLT is provided in the xq2xml suite. Another interesting project would be to convert (at least parts of) an XQuery expression to SQL.

An XML view of XQuery: the parser

The first stage of processing XQuery is to parse it. Fortunately The Scott Boag, on behalf of W3C Working Groups, makes available an open source (Java) XQuery parser. This is based on a JavaCC grammar that is automatically extracted from the XML source files of the XQuery specification, and is used by the working groups to test the feasibility of the grammar being specified.

As originally supplied, this parser emitted a textual display of the parse tree produced by parsing a supplied XQuery. A trivial modification, originally

supplied as part of xq2xml, but now incorporated into the parser provides a Java class that displays the parse tree as XML. The XQuery:

```
1 + 2
```

is reported as the XML:

```
<XPath2>
  <QueryList>
    <Module>
      <MainModule>
        <Prolog/>
        <QueryBody>
          <Expr>
            <AdditiveExpr><data>+</data>
              <PathExpr>
                <IntegerLiteral><data>1</data></IntegerLiteral>
              </PathExpr>
              <PathExpr>
                <IntegerLiteral><data>2</data></IntegerLiteral>
              </PathExpr>
            </AdditiveExpr>
          </Expr>
        </QueryBody>
      </MainModule>
    </Module>
  </QueryList>
</XPath2>
```

Apart from the two outer elements, XPath2 and QueryList, the element names in this representation are all directly taken from the production names in the EBNF defining XQuery. It is this XML view of the XQuery expression that forms the basis of all the transformations provided by xq2xml.

XQuery to XQueryX

Originally, when contemplating basing Query transformations on an XML view of XQuery, I had hoped to be able to use XQueryX, and XML syntax for XQuery being specified by the W3C. Unfortunately at the time there appeared to be no available (or even announced) tools to generate XQueryX, and so I decided to make generation of XQueryX the first transformation to be provided by xq2xml. The XSLT2 stylesheet, xq2xsl, takes input the XML generated by the XQuery parser and generates XQueryX. This transformation is surprisingly intricate, as XQueryX differs from a "natural" encoding of the XQuery grammar in several respects.

The XQueryX encoding of the above example produced by the xq2xqx stylesheet is as follows:

```
<xqx:module xmlns:xqx="http://www.w3.org/2005/XQueryX">
  <xqx:mainModule>
    <xqx:queryBody>
      <xqx:addOp>
        <xqx:firstOperand>
          <xqx:integerConstantExpr>
            <xqx:value>1</xqx:value>
          </xqx:integerConstantExpr>
        </xqx:firstOperand>
        <xqx:secondOperand>
          <xqx:integerConstantExpr>
            <xqx:value>2</xqx:value>
          </xqx:integerConstantExpr>
        </xqx:secondOperand>
      </xqx:addOp>
    </xqx:queryBody>
  </xqx:mainModule>
</xqx:module>
```

I decided to base other transformations in the xq2xml suite directly on the XML produced by the parser rather than going through the XQueryX syntax, even though that is more standardised. The XQueryX encoding is surprisingly complex, and yet loses many features that may be important to an author of a Query, for example abbreviated syntax is not supported, so a Query such as `//abc` if encoded as XQueryX and transformed back to XQuery would be represented as `/descendent-or-self::node()/abc`. This may not be desirable if for example the transform is just intended to expand optional axes, a user may not be expecting other parts of the Query to change. Also, while the expanded form of the abbreviated syntax has the same semantics, it may have different run time behaviour, Query optimisers may find it easier to recognise the syntactic construct using `//` and optimise it to use database indexes or other techniques.

XQuery to XSLT

The initial parsing of the XQuery is as described in the previous section. However the resulting XML is then transformed with `xq2xsl.xsl` rather than `xq2xqx.xsl` which results in XSLT rather than XQueryX. The simple test query above results in:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:fn="http://www.w3.org/2005/xpath-functions"
            xmlns:local="http://www.w3.org/2005/XQuery-local-functions"
            xmlns:xq="java:Xq2xml"
            xmlns:xdt="http://www.w3.org/2005/xpath-datatypes"
            version="2.0"
```

```
            extension-element-prefixes="xq"
            exclude-result-prefixes="xq xs xdt local fn">
  <xsl:param name="input" as="item()" select="1"/>
  <xsl:output indent="yes"/>
  <xsl:template name="main">
    <xsl:for-each select="$input">
      <xsl:sequence select="( 1 + 2 )"/>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

Most of this is just standard boilerplate declaring some namespaces that are predefined in XQuery but must be explicitly declared in XSLT. The actual expression (which is also valid XPath in this case) appears as the select attribute to `xsl:sequence`.

However not every XQuery consists of a single XPath expression, so consider a slightly more complicated example:

```
declare variable $x := <a> <b>zzz</b> </a>;
$x/b
```

This is converted to:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:fn="http://www.w3.org/2005/xpath-functions"
            xmlns:local="http://www.w3.org/2005/XQuery-local-functions"
            xmlns:xq="java:Xq2xml"
            xmlns:xdt="http://www.w3.org/2005/xpath-datatypes"
            version="2.0"
            extension-element-prefixes="xq"
            exclude-result-prefixes="xq xs xdt local fn">
  <xsl:param name="input" as="item()" select="1"/>
  <xsl:output indent="yes"/>
  <xsl:variable name="x" as="item()*">
    <xsl:for-each select="$input">
      <xsl:element name="a">
        <xsl:element name="b">
          <xsl:text>zzz</xsl:text>
        </xsl:element>
      </xsl:element>
    </xsl:for-each>
  </xsl:variable>
  <xsl:template name="main">
    <xsl:for-each select="$input">
      <xsl:sequence select="$x/ b "/>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

XQuery variable declarations and element constructors map naturally to equivalent XSLT instructions. When a subexpression is required to be eval-

uated as XPath rather than XSLT then (as before) `xsl:sequence` is used to switch to XPath evaluation.

The remaining complication is if the XQuery expression that is being evaluated as XPath contains a sub expression that is mapped to XSLT. XML (and XSLT) rules mean that it is not possible to directly embed the XSLT instructions in XPath. So in this case a function definition is constructed to hold the XSLT sub expression, as in the following example:

```
count((<a/>,1+2,<b/>))
```

Which is converted to the following XSLT:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:fn="http://www.w3.org/2005/xpath-functions"
            xmlns:local="http://www.w3.org/2005/XQuery-local-functions"
            xmlns:xq="java:Xq2xml"
            xmlns:xdt="http://www.w3.org/2005/xpath-datatypes"
            version="2.0"
            extension-element-prefixes="xq"
            exclude-result-prefixes="xq xs xdt local fn">
  <xsl:param name="input" as="item()" select="1"/>
  <xsl:output indent="yes"/>
  <xsl:template name="main">
    <xsl:for-each select="$input">
      <xsl:sequence select="count((xq:xpath_d1e27(.)))"/>
    </xsl:for-each>
  </xsl:template>

  <xsl:function name="xq:xpath_d1e27" as="item()*">
    <xsl:param name="xq:here"/>
    <xsl:for-each select="$xq:here">
      <xsl:element name="a"/>
      <xsl:sequence select="( 1 + 2 )"/>
      <xsl:element name="b"/>
    </xsl:for-each>
  </xsl:function>
</xsl:stylesheet>
```

These two techniques, switching from XPath to XSLT via a function call, and from XSLT to XPath via `xsl:sequence` allow the whole of XQuery to be more or less directly mapped to XSLT in a simple manner. The two exceptions are typeswitch and "order by" which do not have direct equivalents in XSLT. Typeswitch is fairly simply mapped to an `xsl:choose` expression testing types with the instance of operator. Order by is rather more complicated as described below. Minor discrepancies between default handling of namespaces and white space account for much of the complication in the code (and probably most of any remaining bugs).

As with any conversion between two programming languages, one possible view of the converter is that it is the first stage of an XQuery implementation, one just needs to pass the output of the converter to a suitable XSLT engine to execute the code. At the time of writing the only publicly available XSLT2 processor that is complete enough to run these transformations is Saxon 8, which includes a native implementation of XQuery, so the usefulness of xq2xsl as a stand-alone XQuery implementation maybe somewhat limited at present, however it does provide a demonstration of an alternative implementation strategy, which has been tested against all releases of the XQuery test suite (up to 0.8.6), and hopefully version 1 of the test suite will be released, and tested against xq2xsl before the conference date. Even with Saxon, the transformations may have some practical use, converting function libraries written in XQuery into a form that may be directly included in XSLT using `xsl:import`, without having to use any proprietary extension functions to call XQuery from XSLT. As XSLT2 matures and more implementations are made available, it may be that xq2xsl does in fact become a practical implementation of XQuery in contexts where there is a readily available XSLT engine but no direct access to XQuery.

XQuery and XSLT compared

XQuery and XSLT clearly have a lot in common. They are developed in parallel and share a data model (XDM) used to describe all inputs and results, and share, in XPath2, the majority of the syntax used in either language to select nodes and to evaluate function calls and other expressions.

The major syntactic differences are in node construction, where XSLT uses XML syntax, literal result elements and `xsl:element` instructions, whereas XQuery uses non-XML syntax both for computed element constructors (comparable to `xsl:element`) and direct element constructors (modelled on literal XML elements, but using a syntax that mimics XML rather than being XML).

Ignoring these syntactic differences, the major differences are in the features that are in one language but not the other:

- XQuery has no analogue of template application, so no analogue of `xsl:apply-templates`, it does not have any grouping facility comparable to `xsl:for-each-group`, and its regular expression support is more limited, having the XPath regular expression functions, which are limited to doing replacement on strings, but does not have an analogue of `xsl:analyze-string` which may be used to construct nodes as a result of regular expression matching.
- XSLT lacks XQuery's typeswitch expression and it does not have a direct

analogue of FLWOR expressions.

As described below (and implemented in xq2xsl) it is possible to implement the "missing" features from XQuery in XSLT, although, especially in the case of FLWOR, not in a particularly natural. It is not really practical to implement the "missing" XSLT features in XQuery as grouping at regular expression matching, at least really require the use of higher order functions and neither XQuery nor XSLT provide any general syntax for manipulating functions as objects, and so rely on special syntactic forms being present in the language to handle these special cases.

It is not really surprising, nor a criticism of XQuery, that it is effectively a profile of XSLT. It is designed to be more amenable for use as a database query language, and in that mode XSLT's traditional "push" mode, driven by the implicitly recursive calls on `apply-templates`, isn't really a good fit for the underlying architecture, (or, it appears, with the mindset of the typical database user!). Template application is far more natural (and was really designed for) traditional "document" uses. The fact that XQuery also drops support for grouping is rather more surprising, especially given the pain that has been experienced by XSLT1 users (which also had no explicit support for grouping) who have to learn the various idioms that have been developed. Hopefully a future version of XQuery will add a grouping construct.

Typeswich

xq2xsl maps typeswitch to xslt's `xsl:choose` expression, testing on the value of "instance of" expressions.

```
typeswitch (2)
  case $zip as element(*)
      return 7
  case $postal as element(ll )
      return 8
  default $x return 9

<xsl:variable as="item()*" name="xq:ts" select="( 2 ,$xq:empty)"/>
  <xsl:choose>
    <xsl:when test="$xq:ts instance of element( * )">
      <xsl:variable as=" element( * )" name="zip" select="$xq:ts"/>
      <xsl:sequence select=" 7 "/>
    </xsl:when>
    <xsl:when test="$xq:ts instance of element( ll )">
      <xsl:variable as=" element( ll )" name="postal" select="$xq:ts"/>
      <xsl:sequence select=" 8 "/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:variable as="item()*" name="x" select="$xq:ts"/>
```

```
    <xsl:sequence select=" 9 "/>
   </xsl:otherwise>
 </xsl:choose>
```

The spurious trailing `",$xq:empty"` in the definition of the main variable `xq:ts` whose type is to be tested is there to ensure that the XSLT compiler does not statically determine its type (which would allow it to raise static errors on branches of the `xsl:choose` that could not arise). Early releases used `,()` but at some point Saxon's optimiser recognised this expression and still statically evaluated the type of `xq:ts` in some cases. `xq:empty` is a parameter with default definition of `()`. It is never given a different value, but it prevents the expression from being evaluated at compile time.

FLWOR expressions

The order by clause in a FLWOR expression is the one XQuery feature that does not easily translate to XSLT. This is essentially because it doesn't easily fit in the XQuery/XPath data model as its natural semantics requires nested sequences (called tuples in the XQuery documentation) but XPath sequences can not contain sequences, only atomic values. The XQuery Formal semantics also explicitly omits to give semantics to Order By for the same reason.

The xq2xsl translation does (baring bugs) provide a full translation of a FLWOR expression into XSLT, however as translation of the most general case is rather verbose, the system detects some special cases and translates them more directly. These special cases will be discussed first.

FLWOR: XPath

If the converter is in XPath mode, and detects a FLWOR expression that corresponds to an XPath For expression (specifically it has no "at" clause, no variable type declaration, no let clause, where or order by clauses) then the expression essentially translates to itself.

If any other FLWOR expression occurs in XPath mode, then the converter switches to XSLT mode (which typically means generating a new function call).

FLWOR: XSLT, no tuples

If the converter is in XSLT mode and the FLWOR expression has no order by clause, or if there is an order by clause but only one *for*-variable (and no *let*-variables) then the FLWOR expression is translated fairly naturally to a nested set of `xsl:for-each` instructions, and each clause of the order by is converted to an `xsl:sort` instruction (or two `xsl:sort` instructions if empty-first ordering is required).

FLWOR: XSLT, CARTESIAN PRODUCT

Note this case is not currently detected by the system, and FLWOR expressions that could be converted as described here are converted as described in the next section, which produces a more verbose and possibly less efficient translation.

Consider a general *2*-variable for expression, in which the sequence over which the inner variable ranges does not depend on the outer variable. This means that we can assume that each *for*-variable is ranging over a sequence contained in a variable that has been calculated before the FLWOR expression.

```
for $i in $is, $j in $js
order by f($i,$j)
return
g($i,$j)
```

for some functions `f` and `g` then this can be rewritten as:

```
let $ci :=count($is) return
let $cj :=count($js) return
for $n in (0 to $ci * $cj)
let $i :=$is[$n mod $ci)+1]
let $j := $js[($n idiv $ci) +1]
order by f($i, $j)
return
g($i,$j)
```

This rewrite has produced a FLWOR expression with just one variable, so may be translated to XSLT `for-each` as described in the previous section.

Cases with more variables could be handled in exactly the same way, with appropriate `div` and `mod` expressions to calculate the required indices.

FLWOR: XSLT, DEPENDENT PRODUCT (GENERAL CASE)

The general two variable case is:

```
for $i in $is,
  $j in F($i)
  order by f(i,j)
  return
  g($i,$j)
```

However this may be rewritten as:

```
let $one := (
  for $i at $ip in $is,
  $j at $jp in F($i)
  return
  encode($ip,$jp)
```

```
)
  return

  for $z in $one
  let $indexes:=decode($z)
  let $ip:=$indexes[1]
  let $i:= $is[$ip]
  let $jp:=$indexes[2]
  let $j:= (F($i))[$jp]
  order by f($i,$j)
  return
  g($i,$j)
```

where `encode` is anything that encodes a sequence of integers as a single item and `decode` is anything that gets the sequence of integers back. specifically the current converter encodes a sequence of integers as a string using:

```
codepoints-to-string(($x1+32,$x2+32,...))
```

and decodes using

```
for $i in string-to-codepoints(.)  return($i - 32)
```

Note this uses a single character per integer, which limits the method to a million or so entries per sequence, but could easily use more characters per integer, so this is not really a real restriction in the method

The main disadvantage of all this is that the expressions `F($i)` above that generate the sequences get evaluated multiple times and might be expensive. This could be optimised by the converter (for example detecting the case that the sequences do not depend on the range variables, and so using the method in the previous section, but some of the optimisations are probably more easily done by the XSLT compiler. For example I believe Saxon does not evaluate variables that are never used, so there is no real need for the converter to analyse the expressions and see if all the variables defined are needed.

Other differences

FLWOR and typeswitch expressions are the only major structural differences that ought to affect a translation from XQuery to XSLT, however as the languages were not designed to be explicitly equivalent, there are many places where a translator has to do far more work to produce a fully equivalent expression. Unfortunately this often impacts on the readability of the generated code.

To give one example, XQuery allows elements to be constructed using a "direct element constructor" that mimics XML syntax:

```
<elem a="2">...</elem>
```

Constructs an element with name *elem* and attribute a with value 2. If you were translating this to XSLT in order to teach the general principles or to produce easily maintainable XSLT code, one would hope to translate this to an XSLT literal result element (which would look identical). Unfortunately XQuery and XSLT have different behaviour if two attributes of the same name either appear in the literal element, or are generated in the element content. In XSLT this is not an error (any earlier attribute nodes of the same name are silently discarded, but in XQuery this is an error condition.) In order to fully comply with the XQuery specification xq2xsl never uses the literal attribute syntax to generate attribute nodes, the entire sequence generated by any attribute constructors and the element content is first generated into a temporary variable, and then this sequence queried for duplicate attributes before being copied into the constructed element.

```
<xsl:element name="elem">
  <xsl:variable name="content" as="item()*">
    <xsl:attribute name="a">
      <xsl:text>2</xsl:text>
    </xsl:attribute>
    <xsl:text>...</xsl:text>
  </xsl:variable>
  <xsl:variable name="att" select="$content[.  instance of
  attribute()]/node-name(.)"/>
  <xsl:if test="count($att)!=count(distinct-values($att))">
    <xsl:sequence
    select="error(QName('http://www.w3.org/2005/xqt-errors',
    'XQDY0025'),'duplicated attribute')"/>
  </xsl:if>
  <xsl:sequence select="$content"/>
</xsl:element>
```

The number of such "unnatural" translations is growing steadily (largely driven by the XQuery Test suite, which while it has some problems, does cover a large part of the XQuery language). It is not currently implemented but it would be possible, and perhaps useful, to have a switch to turn off these transformations that are there to gain strict XQuery conformance, and instead produce more natural, and most likely more efficient, XSLT code. Especially in cases such as this, where the only non-conformance in the "natural" code is in an error condition.

FullAxis: XQuery to XQuery

The stylesheet `fullaxis.xsl` is a small stylesheet that imports `xq2xq.xsl` (a stylesheet that just performs an "identity transformation", writing the

Query back in XQuery syntax) and adds a few simple templates to rewrite any use of the optional axes in XQuery (ancestor, ancestor-or-self, preceding, preceding-sibling, following, following-sibling). This should be useful if you have to use a system that inconveniences its users by not providing these axes.

Presumably there will be such systems as the Working Group have gone to the trouble of making these axes optional, although not supporting the axes would be a surprising choice for any implementor as (as demonstrated here) removing the axes does not limit the expressive power (so doesn't offer any new implementation strategies as would perhaps be the case if path expressions were strictly limited to forward searches). It just inconveniences the user by making them type a more unwieldy expression (which is probably harder to optimise as it is harder to spot a general expression than the restricted form of an axis). This stylesheet can't help with the optimisation (or lack thereof) but does at least produce an equivalent expression.

So for example:

```
$y/preceding::*[2]/string(@i)
```

is rewritten to:

```
  $y/
(let $here := .  return
  reverse(root()/descendant::*[.  << $here][not(descendant::node()[.  is
$here])]))
[2]/string(@i)
```

Licence, support, availability

xq2xsl is a personal project by the author, David Carlisle, it is released under the W3C free software licence. It is not a supported product of my employer NAG Ltd, but it is distributed with their support (and currently from a web site controlled by NAG). There is no guarantee of support, however I am always pleased to receive comments and will try to respond to them in good time.

Web 2.0: Myth and Reality

Eric van der Vlist (Dyomedea)

Abstract

The Web 2.0 is both a new buzzword and a real progress. In this article, I'll to separate the myth from the reality.

Definition

The first difficulty when we want to make an opinion about Web 2.0 is to distinguish its perimeter.

When you need to say if an application is XML or not, that's quite easy: the application is an XML application if and only if it conforms to the XML 1.0 (or 1.1) recommendation.

That's not so easy for Web 2.0 since Web 2.0 is not a standard but a set of practices.

In that sense, Web 2.0 can be compared to REST [1] (Representational State Transfer) which is also a set of practices. Fair enough will you say, but it's easy to say if an application is RESTfull. Why would that be different with Web 2.0?

REST is a concept that is clearly described in a single document: Roy Fielding's thesis [2] which gives a precise definition of what REST is.

On the contrary, Web 2.0 is a blurred concept which aggregates a number of tendencies and everyone seems to have his own definition of Web 2.0 as you can see by the number of articles describing what the Web 2.0 is.

If we really need to define Web 2.0, I'll take two definitions.

The first one is the one given by the French version of Wikipedia [3]:

> "Web 2.0 is a term often used to describe what is perceived as an important transition of the World Wide Web, from a collection of web sites to a computing platform providing web application to users. The proponents of this vision believe that the services of Web 2.0 will come to replace traditional office applications."

This article also gives an history of the term:

> "The term was coined by Dale Dougherty of O'Reilly Media during a brainstorming session with MediaLive International to develop ideas for a conference that they could jointly host. Dougherty suggested that the Web was in a renaissance, with changing rules and evolving business models."

And it goes on by giving a series of examples that illustrate the difference between good old "Web 1.0" and Web 2.0:

> "DoubleClick was Web 1.0; Google AdSense is Web 2.0. Ofoto is Web 1.0; Flickr is Web 2.0."

Google who has launched AdSense in 2003 was doing Web 2.0 without knowing it one year before the term has been invented in 2004!

Technical layer

Let's focus on the technical side of Web 2.0 first.

One of the characteristics of Web 2.0 is to be available to today's users using reasonably recent versions of any browser. That's one of the reasons why Mike Shaver said in its opening keynote [4] at XTech 2005 that "201cWeb 2.0 isn't a big bang but a series of small bangs".

Restricted by the set of installed browsers, Web 2.0 has no other choice than to rely on technologies that can be qualified of "matured":

- HTML (or XHTML pretending to be HTML since Internet Explorer doesn't accept XHTML documents declared as such) − the last version of HTML has been published in 1999.
- A subset of CSS 2.0 supported by Internet Explorer − CSS 2.0 has been published in 1998.
- Javascript − a technology introduced by Netscape in its browser in 1995.
- XML − published in 1998.
- Atom or RSS syndication − RSS has been created by Netscape in 1999.
- HTTP protocol − the latest HTTP version has been published in 1999.
- URIs − published in 1998.
- REST − a thesis published in 2000.
- Web Services − XML-RPC APIs for Javascript were already available in 2000.

The usage of XML over HTTP in asynchronous mode has been given the name "Ajax".

Web 2.0 appears to be the full appropriation by web developers of mature technologies to achieve a better user experience.

If it's a revolution, this is a revolution in the way to use these technologies together, not a revolution in the technologies themselves.

Office applications

Can these old technologies really replace office applications? Is Web 2.0 about rewriting MS Office in Javascript and could that run in a browser?

Probably not if the rule was to keep the same paradigm with the same level of features.

We often quote the famous "80/20" rule after which 80% of the features would require only 20% of the development efforts and sensible applications should focus on these 80% of features.

Office applications have crossed the 80/20 border line years ago and have invented a new kind of 80/20 rule: 80% of the users use probably less than 20% of the features.

I think that a Web 2.0 application focussing on the genuine 80/20 rule for a restricted application or group of users would be a tough competition to traditional office applications.

This seems to be the case for applications such as Google Maps (that could compete with GIS applications on the low end market) or some of the new wysiwyg text editing applications that flourish on the web.

A motivation that may push users to adopt these web applications is the attractiveness of systems that help us manage our data.

This is the case of Gmail, Flickr, del.icio.us or LinkedIn to name few: while these applications relieve us from the burden of the technical management of our data they also give us a remote access from any device connected to the internet.

What is seen today as a significant advantage for managing our mails, pictures, bookmarks or contacts could be seen in the future as a significant advantage for managing our office documents.

Social layer

If the French version of Wikipedia [3] has the benefit of being concise, its is slightly out of date and doesn't describe the second layer of Web 2.0, further developed during the second Web 2.0 conference in October 2005.

The English version of Wikipedia [5] adds the following examples to the

list of Web 1.0/Web 2.0 sites:

> Britannica Online (1.0)/ Wikipedia (2.0), personal sites (1.0)/ blogging (2.0), content management systems (1.0)/ wikis (2.0), directories (taxonomy) (1.0) / tagging ("folksonomy") (2.0)

These examples are interesting because technically speaking, Wikipedia, blogs, wikis or folksonomies are mostly Web 1.0.

They illustrate what Paul Graham is calling Web 2.0 "democracy" [6].

Web 2.0 democracy is the fact that to "lead the web to its full potential" (as the W3C tagline says) the technical layer of the internet must be complemented by a human network formed by its users to produce, maintain and improve its content.

There is nothing new here either and I remember Edd Dumbill launching WriteTheWeb [7] in 2000, "a community news site dedicated to encouraging the development of the read/write web" because the "tide is turning" and the web is no longer a one way web.

This social effect was also the guide line of Tim O'Reilly in his keynote session [8] at OSCON 2004, one year before becoming the social layer of Web 2.0.

Another definition

With a technical and a social layer, isn't Web 2.0 becoming a shapeless bag in which we're grouping anything that's looking new on the web?

We can see in the technical layer a consequence of the social layer, the technical layer being needed to provide the interactivity required by the social layer.

This analysis would exclude from Web 2.0 applications such as Google Maps which have no social aspect but are often quoted as typical examples of Web 2.0.

Paul Graham tries to find common trends between these layers in the second definition that I'll propose in this article:

> "Web 2.0 means using the web the way it's meant to be used. The 'trends' we're seeing now are simply the inherent nature of the web emerging from under the broken models that got imposed on it during the Bubble."

This second definition reminds me other taglines and buzzword heard during these past years:

- The W3C tagline is "Leading the Web to Its Full Potential". Ironically, Web 2.0 is happening, technically based on many technologies specified

by the W3C, without the W3C. It is very tempting to interpret the recent announcement of a "Rich Web Clients Activity" [9] as an attempt to catch a running train.

- Web Services are an attempt to make the web available to applications which was meant to be from the early ages of Web 1.0.
- The Semantic Web − which seems to have completely missed the Web 2.0 train − is the second generation of the web seen by the inventor of Web 1.0.
- REST is the description of web applications using the web as it is meant to be used.
- XML is "SGML on the web" which was possible with HTTP as it was meant to be used.

Here again, Web 2.0 appears to be the continuation of the "little big bangs" of the web.

Technical issues

In maths, continuous isn't the same as differentiable and in technology too, continuous evolutions can change direction.

Technical evolutions are often a consequence of changes in priorities that lead to these changes of direction.

The priorities of client/server applications that we developed in the 90's were:

- the speed of the user interfaces,
- their quality,
- their transactional behaviour,
- security.

They've been swept out by web applications which priorities are:

- a universal addressing system,
- universal access,
- globally fault tolerant: when a computer stops, some services might stop working but the web as a whole isn't affected,
- scalability (web applications support more users than client/server ones dreamed to support),

- a user interface relatively coherent that enables sharing services through URIs,
- open standards.

Web 2.0 is taking back some of the priorities of client/server applications and one needs to be careful that these priorities are met without compromising what is the strength of the web.

Technically speaking, we are lucky enough to have best practices formalized in REST and Web 2.0 developers should be careful to design RESTfull exchanges between browsers and servers to take full advantage of the web.

Ergonomic issues

Web 2.0 run in a web browsers and they should make sure that users can keep their Web 1.0 habits, especially with respect to URIs (including the ability to create bookmarks, send URIs by mail and use their back and forward buttons).

Let's take a simple example to illustrate the point.

Have you noticed that Google, presented as a leading edge Web 2.0 company is stubbornly Web 1.0 on its core business: the search engine itself?

It is easy to imagine what a naive Web 2.0 search engine might look like.

That might start with a search page similar to the current Google suggest [10]. When you start writing your query terms, the service suggests possible completions of you terms.

When you would send the query, the page wouldn't move. Some animation could keep you waiting even if that's usually not necessary with a high speed connection on Google. The query would be sent and the results brought back asynchronously. Then, the list of matches would be displayed in the same page.

The user experience would be fast and smooth, but there are enough drawbacks with this scenario that Google doesn't seem to find it worth trying:

- The URI in the address bar would stay the same: users would have no way to bookmark a search result or to copy and past it to send to a friend.
- Back and forward buttons would not work as expected.
- These result pages would be accessible to crawlers.

The web developer who would implement this Web 2.0 application should take care to provide good workarounds for each of these drawbacks. This is certainly possible, but that requires some effort.

Falling into these traps would be really counter-productive to Web 2.0 since we have seen that these are ergonomic issues that justify this evolution to make the web easier to use.

Development

The last point on which one must be careful when developing Web 2.0 applications are development tools.

The flow of press releases made by software vendors to announce development tools for Ajax based applications may put an end to this problem, but Web 2.0 often means developing complex scripts that are subject to interoperability issues between browsers.

Does that mean that Web 2.0 should ignore declarative definitions of user interface (such as in XForms, XUL or XAML) or even in the 4GL's that had been invented for client/server applications in the early 90's?

A way to avoid this regression is to use a framework that hides most of the Javascript development.

Catching up with the popular "Ruby on Rails", web publications frameworks are beginning to propose Web 2.0 extensions.

This is the case of Cocoon which new version 2.1.8 includes a support of Ajax but also of Orbeon PresentationServer which includes in its version 3.0 a fully transparent support of Ajax through its Xforms engine.

This features enables to write user interfaces in standard XForms (without a single line of Javascript) and to deploy these applications on todays browsers, the system using Ajax interactions between browsers and servers to implement XForms.

Published in 2003, XForms is only two years old, way too young to be part of the Web 2.0 technical stack. Orbeon PresentationServer is a nifty way to use XForms before it can join the other Web 2.0 technologies!

Business model

What about the business model?

The definition of Paul Graham for whom Web 2.0 is a web rid of the bad practises of the internet bubble is interesting when you know that some analysts believe that a Web 2.0 bubble is on its way.

This is the case of Rob Hof (Business Week) who deploys a two step argumentation [11]:

1. "It costs a whole lot less to fund companies to revenue these days", which Joe Kraus (JotSpot) explains [12] by the facts that:

- Hardware is 100x cheaper,
- Infrastructure software is free,
- Access to Global Labor Markets,
- Internet marketing is cheap and efficient for niche markets.

2. Even though venture capital investment seems to stay level, cheaper costs mean that much more companies are being funded with the same level of investment. Furthermore, cheaper costs also means that more companies can be funded by non VC funds.

Rob Hof also remarks that many Web 2.0 startups are created with no other business model than being sold in the short term.

Even if it is composed to smaller bubbles, a Web 2.0 bubble might be on the way.

Here again, the golden rule is to take profit of the Web 1.0 experience.

Data Lock-In era

If we need a solid business model for Web 2.0, what can it be?

One of the answers to this question was in the Tim O'Reilly keynote at OSCON 2004 [8] that I have already mentioned.

Giving its views on the history of computer technologies since their beginning, Tim O'Reilly showed how this history can be split into three eras:

- During the "Hardware Lock-In" era, computer constructors ruled the market.
- Then came the "Software Lock-In" era dominated by software vendors.
- We are now entering the "Data Lock-In" era.

In this new era, illustrated by the success of sites such as Google, Amazon, or eBay, the dominating actors are companies that can gather more data than their competitors and their main asset is the content given or lent by their users for free.

When you outsource your mails to Google, you publish a review or even buy something on Amazon, upload your pictures to Flickr or add a bookmark in del.icio.us, you tie yourself to this site and you trade a service against their usage of your data.

A number of people are talking against what François Joseph de Kermadec [13] is calling the "fake freedom" [14] given by Web 2.0.

Against this fake freedom, users should be careful:

- to trade data against real services,

- to look into the terms of use of each site to know which rights they grant in exchange if these services,
- to demand technical means, based on open standards, to get their data back.

So what?

What are the conclusions of this long article?

Web 2.0 is a term to qualify a new web that is emerging right now.

This web will use the technologies that we already know in creative ways to develop a collaborative "two way web".

Like any other evolution, Web 2.0 comes with a series of risks: technical, ergonomic, financial and threats against our privacy.

Beyond the marketing buzzword, Web 2.0 is a fabulous bubble of new ideas, practices and usages.

The fact that its shape is still so blurred shows that everything is still open and that personal initiatives are still important.

The Web 2.0 message is a message of hope!

References

1. REST
   http://www.ics.uci.edu/%7Efielding/pubs/dissertation/rest_arch_style.htm
2. Roy Fielding's thesis
   http://www.ics.uci.edu/%7Efielding/pubs/dissertation/top.htm
3. Web 2.0 definitions on Wikipedia [French]
   http://fr.wikipedia.org/wiki/Web_2.0
4. Mike Shaver's opening keynote at XTech 2005
   http://xmlfr.org/actualites/ tech/050531-0001#N117
5. Web 2.0 definitions on Wikipedia [English]
   http://en.wikipedia.org/wiki/Web_2.0
6. Web 2.0 seen by Paul Graham
   http://www.paulgraham.com/web20.html
7. WriteTheWeb
   http://writetheweb.com/about/
8. Tim O'Reilly's keynote session at OSCON 2004
   http://conferences.oreillynet.com/cs/os2004/view/e_sess/5515at
9. Rich Web Clients Activity
   http://www.w3.org/2006/rwc/
10. Google suggest
    http://www.google.com/webhp?complete=1&hl=en
11. Rob Hof analysis
    http://www.businessweek.com/the_thread/techbeat/
    archives/2005/10/no_web_20_bubbl.html
12. Joe Kraus explanation
    http://bnoopy.typepad.com/bnoopy/2005/06/its_a_great_tim.html
13. François Joseph de Kermadec
    http://www.oreillynet.com/pub/au/1339
14. A fake freedom by François Joseph de Kermadec
    http://www.oreillynet.com/lpt/wlg/7977

XML Data – The Current State of Affairs

Kamil Toman and Irena Mlýnková (Charles University)

Abstract

At present the eXtensible Markup Language (XML) is used almost in all spheres of human activities. Its popularity results especially from the fact that it is a self-descriptive metaformat that allows to define the structure of XML data using other powerful tools such as DTD or XML Schema. Consequently, we can witness a massive boom of techniques for managing, querying, updating, exchanging, or compressing XML data.

On the other hand, for majority of the XML processing techniques we can find various spots which cause worsening of their time or space efficiency. Probably the main reason is that most of them consider XML data too globally, involving all their possible features, though the real data are often much simpler. If they do restrict the input data, the restrictions are often unnatural.

In this contribution we discuss the level of complexity of real XML collections and their schemes, which turns out to be surprisingly low. We involve and compare results and findings of existing papers on similar topics as well as our own analysis and we try to find the reasons for these tendencies and their consequences.

Introduction

Currently XML and related technologies [7] have already achieved the leading role among existing standards for data representation and are used almost in all spheres of human activities. They are popular for various reasons, but especially because they enable to describe the allowed structure of XML documents using powerful tools such as DTD [7] or XML Schema [9, 20, 6]. Thus we can witness a massive boom of various XML techniques for managing, processing, exchanging, querying, updating, and compressing XML data that mutually compete in speed, efficiency, and minimum space and/or memory requirements.

On the other hand, for majority of the techniques we can find various critical spots which cause worsening of their time and/or space efficiency. In the worst and unfortunately quite often case such bottlenecks negatively influence directly the most interesting features of a particular technique.

If we study the bottlenecks further, we can distinguish two typical problematic situations. Firstly, we can distinguish a group of general techniques that take into account all possible features of input XML data – an approach that is at first glance correct. Nevertheless the standards were proposed as generally as possible enabling future users to choose what suits them most, whereas the real XML data are usually not as "rich" as they could be – they are often surprisingly simple. Thus the effort spent on every possible feature is mostly useless and it can even be harmful.

Secondly, there are techniques that do restrict features of input XML data in some way. Hence it is natural to expect the bottlenecks to occur only in situations when given data do not correspond to the restrictions. The problem is that such restrictions are often "unnatural". They do not result from inherent features of real XML data collections but from other, more down-to-earth, reasons, e.g. limitations of the basic proposal of a particular technique, complexity of such solution etc.

A solution to the given problems could be a detailed analysis of real XML data and their classification. Up to now, there are several works which analyze real XML collections from various points of view. All the papers have the same aim – to describe typical features and complexity of XML data – and all conclude that the real complexity is low indeed. In this paper we briefly describe, discuss, and compare results and findings of the papers as well as our own analysis. We try to find the reasons for these tendencies, their consequences and influence on future processing.

The paper is structured as follows: The first section introduces theconsidered problems. The following, second, section contains a brief overview of formalism used throughout the paper. Section 4 classifies, describes, and discusses XML data analyses. The last, fifth, section provides conclusions.[1]

Formal definitions

For structural analysis of XML data it is natural to view XML documents as ordered trees and DTDs or XSDs (i.e. XML Schema definitions) as sets of regular expressions over element names. Attributes are often omitted for simplicity. We use notation and definitions for XML documents and DTDs

[1]We will not describe neither the basics of XML, DTD, or XML Schema. We suppose that XML and DTD have already become almost a common knowledge whereas description of XML Schema is omitted for the paper length.

from [8] and [5]. (For XSDs are often used the same or similar ones − we omit them for the paper length.)

Definition 1. An XML document *is a finite ordered tree with node labels from a finite alphabet* Σ. *The tree is called* Σ-tree.

Definition 2. A DTD *is a collection of element declarations of the form* $e \rightarrow \alpha$ *where* $e \in \Sigma$ *is an element name and* α *is its content model, i.e. regular expression over* Σ. *The content model* α *is defined as* $\alpha := \epsilon \mid pcdata \mid f \mid \alpha_1, \alpha_2, ..., \alpha_n \mid \alpha_1|\alpha_2|...|\alpha_n \mid \beta* \mid \beta+ \mid \beta?$, *where* ϵ *denotes the empty content model, pcdata denotes the text content, f denotes a single element name, ","* *and "|" stand for concatenation and union (of content models* $\alpha_1, \alpha_2, ...\alpha_n$*), and "*", "+", and "?" stand for zero or more, one or more, and optional occurrence(s) (of content model* β*). One of the element names* $s \in \Sigma$ *is called* a start symbol.

Definition 3. *A* Σ*-tree* satisfies the DTD *if its root is labeled by start symbol s and for every node n and its label e, the sequence* $e_1, e_2, ...e_k$ *of labels of its child nodes matches the regular expression* α, *where* $e \rightarrow \alpha$.

Basic analyses of XML data usually focus on depth of content models and/or XML documents, reachability of content models and/or elements, types of recursion, types of paths and cycles, fan-ins and fan-outs. They are usually similar for both XML documents and XML schemes (regardless the used language).

Definition 4. Depth of a content model α *is inductively defined as follows:*
depth(ϵ*) = 0;*
depth(pcdata) = depth(f) = 1;
depth($\alpha_1, \alpha_2, ..., \alpha_n$*) = depth(*$\alpha_1|\alpha_2|...|\alpha_n$*) = max(depth(*$\alpha_i$*)) + 1;* $1 \leq i \leq n$
depth($\beta*$*) = depth(*$\beta+$*) = depth(*$\beta?$*) = depth(*β*) + 1.*

Definition 5. Distance of elements e_1 *and* e_2 *is the number of edges in* Σ*-tree separating their corresponding nodes.*

Level of an element *is distance of its node from the root node. The level of the root node is 0.*

Depth of an XML document *is the largest level among all the elements.*

Definition 6. *An element name* e' is reachable from *e, denoted by* $e \Rightarrow e'$, *if either* $e \rightarrow \alpha$ *and* e' *occurs in* α *or* $\exists\ e''$ *such that* $e \Rightarrow e''$ *and* $e'' \Rightarrow e'$.

A content model α *is* derivable, *denoted by* $e \Rightarrow \alpha$, *if either* $e \rightarrow \alpha$ *or* $e \Rightarrow \alpha'$, $e' \rightarrow \alpha''$, *and* $\alpha = \alpha'[e'/\alpha'']$, *where* $\alpha'[e'/\alpha'']$ *denotes the content model obtained by substituting* α'' *for all occurrences of* e' *in* α'.

An element name e is reachable, *if r* $\Rightarrow$ *e, where r is the name of root*

element. Otherwise it is called unreachable.

Definition 7. *An element e is* recursive *if there exists at least one element d in the same document such that d is a descendant of e and d has the same label as e.*

The element-descendant association is called an ed-pair.

Definition 8. *An element e is called* trivially recursive *if it is recursive and for every α such as $e \Rightarrow \alpha$ e is the only element that occurs in α and neither of its occurrences is enclosed by "*" or "+".*

An element e is called linearly recursive *if it is recursive and for every α such as $e \Rightarrow \alpha$ e is the only recursive element that occurs in α and neither of its occurrences is enclosed by "*" or "+".*

An element e is called purely recursive *if it is recursive and for every α such as $e \Rightarrow \alpha$ e is the only recursive element that occurs in α.*

An element that is not purely recursive is called generally recursive *element.*

Definition 9. Simple path *(in a non-recursive DTD) is a list of elements e_1, e_2,... e_k, where $e_i \rightarrow \alpha_i$ and e_{i+1} occurs in α_i for $1 \leq i < k$. Parameter k is called* length of a simple path.

Simple cycle *is a path in the form e_1, e_2,... e_k, e_1, where e_1, e_2,... e_k are distinct element names.*

Chain of stars *is a simple path of elements e_1, e_2,... e_{k+1}, where e_{i+1} is in the corresponding α_i followed by "*" or "+" for $1 \leq i \leq k$. Parameter k is called* length of a chain of stars.

Definition 10. Fan-in *of an element e is the cardinality of the set $\{e' \mid e' \rightarrow \alpha'$ and e occurs in $\alpha'\}$. An element name with large fan-in value is called* hub.

Definition 11. Element fan-out *of element e is the cardinality of the set $\{e' \mid e \rightarrow \alpha$ and e' occurs in $\alpha\}$.*

Minimum element fan-out *of element e is the minimum number of elements allowed by its content model α.*

Maximum element fan-out *of element e is the maximum number of elements allowed by content model α.*

Attribute fan-out *of element e is the number of its attributes.*

There are also XML constructs, that can be called advanced, such as types of mixed content, DNA patterns, or relational patterns.

Definition 12. *An element is called* trivial *if it has an arbitrary amount of attributes and its content model $\alpha := \epsilon \mid pcdata$.*

A mixed content of element is called simple *if it consist only of trivial elements. Otherwise it is called* complex.

Definition 13. *An nonrecursive element e is called* DNA pattern *if its content model α is not mixed and consists of a nonzero amount of trivial elements and just one nontrivial and nonrecursive element which is not enclosed by "*" or "+". The nontrivial subelement is called degenerated branch.*

Depth *of a DNA pattern e is the maximum depth of its degenerated branch.*

Definition 14. *A nonrecursive element e is called* relational pattern *if it has an arbitrary amount of attributes and its content model $\alpha := (e_1, e_2, ..., e_n)* \mid (e_1, e_2, ..., e_n)+ \mid (e_1|e_2|...|e_n)* \mid (e_1|e_2|...|e_n)+$ and it is not mixed.*

A nonrecursive element e is called shallow relational pattern *if it has an arbitrary amount of attributes and its content model $\alpha := f* \mid f+$ and it is not mixed.*

Analyses and results

Up to now several papers have focused on analysis of real XML data. They analyze either the structure of DTDs, the structure of XSDs, the structure of XML data regardless their schema, or the structure of XML documents in relation to corresponding schema. The sample data usually essentially differ.

DTD analysis

Probably the first attempt to analyze the structure of XML data can be found in [18]. The paper is relatively old (especially with regard to the fast development of XML standards) and it contains a quantitative analysis of 12 DTDs and a general discussion of how they are (mis)used.

The analysis involves:

- the size of DTDs, i.e. the number of elements, attributes, and entity references,
- the structure of DTDs, i.e. the number of root elements and depth of content models, and
- some specific aspects, i.e. the use of mixed-content, `ANY`, `ID`s and `IDREF`s, or the kind of attribute decorations used (i.e. `implied`, `required`, and `fixed` attributes).

The discussion of current (mis)using of DTDs brings various conclusions, involving especially shortcomings of DTD. Most of them have already been overcome in XML Schema – e.g. the necessity to use XML itself for description of the structure of XML data, the missing operator for unordered

sequences, insufficient tools for inheritance and modularity, the requirement for typed `IDREF`s (i.e. those which cannot refer to any `ID`) etc.

There are also interesting observations concerning structure of the data, especially the finding that content models have the depth less than 6 and that `ID`s and `IDREF`s are not used frequently (probably due to the above mentioned problem with typing). According to the author the most important conclusion is that DTDs are usually incorrect (both syntactically and semantically) and thus are not a reliable source of information.

Second paper [8] that also focuses on DTDs describes analyses which are more statistical than in the previous case. It analyzes 60 DTDs further divided according to their intended future use into three categories:

- *app*, i.e. DTDs designed for data interchange,
- *data*, i.e. DTDs for data that can easily be stored in a database, and
- *meta*, i.e. DTDs for describing the structure of document markup.

The statistics described in the paper focus on graph theoretic properties of DTDs and can be divided into:

- *local*, i.e. describing kinds of content models found at individual element declarations (e.g. the number of mixed-content elements) and
- *global*, i.e. describing graph structure of the DTD (e.g. the maximum path length allowed by the DTD).

Local properties focus on four types of features − content model classifications, syntactic complexity, non-determinism, and ambiguity. The classification of content models involves *pcdata*, ϵ, *any*, mixed content, "|" only (but not mixed) content, "," only content, complex content (i.e. with both "|"s and ","s), list content (i.e. the usage of "+" or"*" for one element), and single content (i.e. the optional usage of "?" for one element); the syntactic complexity is expressed by the previously defined *depth* function. The question of both non-determinism and ambiguity (i.e. a special case of non-determinism) of content models is a bit controversial since non-deterministic content models are not allowed by the XML standards. The most important findings for local properties are that the content model of DTDs is usually not complex (the maximum depth is 9, whereas its mode is even 3) and that despite the standards, there are both non-deterministic and ambiguous content models.

Global properties discussed in the paper involve reachability, recursions, simple paths and cycles, chains of stars and hubs. The most important findings are listed below.

- Unreachable elements are either root elements or useless, whereas the mode of their number is 1, i.e. the root element is usually stated clearly.
- There are no linear recursive elements, whereas the number of non-linear recursive elements is significant (i.e. they occur in 58% of all DTDs).
- The maximum length of simple path is surprisingly small (mostly less than 8), whereas on the other hand the number of simple paths as well as simple cycles is either small (less than 100) or large (more than 500).
- The length of the longest chain of stars is usually small (its mode is 3).
- Hubs exist in all categories od DTDs and their number is significant.

Last found paper [12] which focuses on DTD analysis is trying to adapt software metrics to DTDs. It defines five metrics, also based on their graph representation − i.e. size, complexity, depth, fan-in, and fan-out, whereas all of them were already defined and discussed. Regrettably, there are just 2 DTD examples for which the statistics were counted.

DTD vs. XML schema

With the arrival of XML Schema, as the extension of DTD, has arisen a natural question: Which of the extra features of XML Schema not allowed in DTD are used in practise? Paper [5] is trying to answer it using analysis of 109 DTDs and 93 XSDs. Another aim of the paper is to analyze the real structural complexity for both the languages, i.e. the degree of sophistication of regular expressions used.

The former part of the paper focuses on analysis of XML Schema features. The features and their resulting percentage are:

- extension[2] (27%) and restriction (73%) of simple types,
- extension (37%) and restriction (7%) of complex types,
- `final` (7%), `abstract` (12%), and `block` (2%) attribute of complex type definitions,
- substitution groups (11%),
- unordered sequences of elements (4%),
- `unique` (7%) and `key`/`keyref` (4%) features,
- namespaces (22%), and

[2]Extension of a simple type means adding attributes to the simple type, i.e. creating a complex type with simple content.

- redefinition of types and groups (0%).

As it is evident, the most exploited features are restriction of simple types, extension of complex types, and namespaces. The first one reflects the lack of types in DTD, the second one confirms the naturalness of object-oriented approach (i.e. inheritance), whereas the last one probably results from mutual modular usage of XSDs. The other features are used minimally or are not used at all.

Probably the most interesting finding is, that 85% of XSDs define so called *local tree languages* [17], i.e. languages that can be defined by DTDs as well, and thus that the expressiveness beyond local tree grammars is needed rarely.

XML schema analysis

Paper [5] mentioned in the previous section analyzed the properties of DTDs and XSDs together. Nevertheless its first part focused only on statistical analysis of real usage of new XML Schema features. Paper [14] has a similar aim − it defines 11 metrics of XSDs and two formulae that use the metrics to compute complexity and quality indices of XSDs. The metrics are:

- the number of (both globally and locally defined) complex type declarations, which can be further divided into text-only, element-only, and mixed-content,
- the number of simple type declarations,
- the number of annotations,
- the number of derived complex types,
- the average number of attributes per complex type declaration,
- the number of global (both simple and complex) type declarations,
- the number of global type references,
- the number of unbounded elements,
- the average bounded element multiplicity size, where multiplicity size is defined as *(maxOccurs - minOccurs + 1)*,
- the average number of restrictions per simple type declaration,
- *element fanning*, i.e. the average fan-in/fan-out.

On the basis of the experience in analyzing many XSDs the authors define two indices for expressing their quality and complexity.

Definition 15. Quality Index = *(Ratio of simple to complex type declarations) * 5 + (Percentage of annotations over total number of elements) * 4 + (Average restrictions per simple type declarations) * 4 + (Percentage of derived complex type declarations over total number of complex type declarations) * 3 (Average bounded element multiplicity size) * 2 (Average attributes per complex type declaration) * 2*

Complexity Index = *(Number of unbounded elements) * 5 + (Element fanning) * 3 + (Number of complex type declarations) + (Number of simple type declarations) + (Number of attributes per complex type declaration)*

Unfortunately, there is just one XSD example for which the statistics were counted.

XML DOCUMENT ANALYSIS

Previously mentioned analyses focused on descriptions of the allowed structure of XML documents. By contrast paper [15] (and its extension [2]) analyzes directly the structure of their instances, i.e. XML documents, regardless eventually existing DTDs or XSDs.[3] It analyzes about 200 000 XML documents publicly available on the Web, whereas the statistics are divided into two groups − statistics about the Web and statistics about the XML documents.

The Web statistics involve:

- clustering of the source web sites by zones consisting of Internet domains (e.g. *.com*, *.edu*, *.net* etc.) and geographical regions (e.g. Asia, EU etc.),
- the number and volume (i.e. the sum of sizes) of documents per zone,
- the number of DTD (48%) and XSD (0.09%) references,
- the number of namespace references (40%),
- distribution of files by extension (e.g. *.rdf*, *.rss*, *.wml*, *.xml* etc.), and
- distribution of document *out-degree*, i.e. the number of `href`, `xmlhref`, and `xlink:href` attributes.

Obviously most of them describe the structure of the XML Web and categories of the source XML documents.

Statistics about the structure of XML documents involve:

- the size of XML documents (in bytes),

[3]The paper just considers whether the document does or does not reference a DTD or an XSD.

- the amount of markup, i.e. the amount of element and attribute nodes versus the amount of text nodes and the size of text content versus the size of the structural part,
- the amount of mixed content elements,
- the depth of XML documents and the distribution of node types (i.e. element, attribute, or text nodes) per level,
- element and attribute fan-out
- the number of distinct strings, and
- recursion.

The most interesting findings of the research are as follows:

- The size of documents varies from 10B to 500kB; the average size is 4,6kB.
- For documents up to 4kB the number of element nodes is about 50%, the number of attribute nodes about 30%. Surprisingly, for larger documents the number of attribute nodes rises to 50%, whereas the number of element nodes declines to 38%. The structural information still dominates the size of documents.
- Although there are only 5% of all elements with mixed content, they were found in 72% of documents.
- Documents are relatively shallow − 99% of documents have fewer than 8 levels, whereas the average depth is 4.
- The average element fan-out for the first three levels is 9, 6, and 0.2; the average attribute fan-out for the first four levels is 0.09, 1, 1.5, and 0.5. Surprisingly, 18% of all elements have no attributes at all.

A great attention is given to recursion which seems to be an important aspect of XML data. The authors mention the following findings:

- 15% of all XML documents contain recursive elements.
- Only 260 distinct recursive elements were found. In 98% of recursive documents there is only one recursive element used.
- 95% of recursive documents do not refer to any DTD or XSD.
- Most elements in ed pairs have the distance up to 5.
- The most common average fan-outs are 1 (60%) and 2 (37%), the average recursive fan-out is 2.2.

Lastly, paper [13] that focuses on analysis of XML documents consists of two parts − a discussion of different techniques for XML processing and an analysis of real XML documents. The sample data consists of 601 XHTML web pages, 3 documents in DocBook format[4], an XML version of Shakespeare's plays[5] (i.e. 37 XML documents with the same simple DTD) and documents from *XML Data repository* project[6]. The analyzed properties are the maximum depth, the average depth, the number of simple paths, and the number of unique simple paths; the results are similar to previous cases.

XML documents vs. XML schemes

The work initiated in the previously mentioned articles is taken up recently by paper [16]. It enhances the preceding analyses and defines several new constructs for describing the structure of XML data (e.g. DNA or relational patterns). It analyzes XML documents together with their DTDs or XSDs eventually that were collected semi-automatically with interference of human operator. The reason is that automatic crawling of XML documents generates a set of documents that are unnatural and often contain only trivial data which cause misleading results. The collected data consist of about 16 500 XML documents of more than 20GB in size, whereas only 7.4% have neither DTD nor XSD. Such low ratio is probably caused by the semi-automatic gathering.

The data were first divided into following six categories:

- *data-centric documents*, i.e. documents designed for database processing (e.g. database exports, lists of employees etc.),
- *document-centric documents*, i.e. documents which were designed for human reading (e.g. Shakespeare's plays, XHTML [1] documents etc.)
- *documents for data exchange* (e.g. medical information on patients etc.),
- *reports*, i.e. overviews or summaries of data (usually of database type),
- *research documents*, i.e. documents which contain special (scientific or technical) structures (e.g. protein sequences, DNA/RNA structures etc.), and
- *semantic web documents*, i.e. RDF [4] documents.

The statistics described in the paper are also divided into several categories. They were computed for each category and if possible also for both XML

[4] `http://www.docbook.org/`
[5] `http://www.ibiblio.org/xml/examples/shakespeare/`
[6] `http://www.cs.washington.edu/research/xmldatasets/`

documents and XML schemes and the results were compared. The categories are as follows:

- *global statistics*, i.e. overall properties of XML data (e.g. number of elements of various types such as empty, text, mixed, recursive etc., number of attributes, text length in document, paths and depths etc.),
- *level statistics*, i.e. distribution of elements, attributes, text nodes, and mixed contents per each level,
- *fan-out statistics*, i.e. distribution of branching per each level,
- *recursive statistics*, i.e. types and complexity of recursion (e.g. exploitation rates, depth, branching, distance of ed-pairs etc.),
- *mixed-content statistics*, i.e. types and complexity of mixed contents (e.g. depth, percentage of simple mixed contents etc.),
- *DNA statistics*, i.e. statistics focussing on DNA patterns (e.g. number of occurrences, width, or depth), and
- *relational statistics*, i.e. statistics focussing on both relational and shallow relational patterns (e.g. number of occurrences, width, or fan-out).

Most interesting findings and conclusions for all categories of statistics are as follows:

- The amount of tagging usually dominates the size of document.
- The lowest usage of mixed-content (0.2%) and empty (26.8%) elements can be found in data-centric documents.
- The highest usage of mixed-content elements (77%) can be found in document-centric documents.
- Documents of all categories are typically shallow. (For 95% of documents the maximum depth is 13, the average depth is about 5.)
- The highest amounts of elements, attributes, text nodes, and mixed contents as well as fan-outs are always at first levels and then their number of occurrences rapidly decreases.
- Recursion occurs quite often, especially in document-centric (43%) and exchange (64%) documents, although the number of distinct recursive elements is typically low (for each category less than 5).
- Recursion, if used, is rather simple − the average depth, branching as well as distance of ed-pairs is always less than 10.

- The most common types of recursion are linear (20% for document-centric and 33% for exchange documents) and pure (19% for document-centric and 23% for exchange documents).
- Unlike document instances almost all schemes specify usually only the most general type of recursion.
- The percentage of simple mixed contents is relatively high (e.g. 79% for document-centric or even 99% for exchange documents) and thus the depth of mixed contents is generally low (on the average again less than 10).
- The number of occurrences of DNA patterns is rather high, especially for research, document-centric, and exchange documents. On the other hand the average depth and width is always low (less than 7).
- The number of occurrences of relational patterns is high, especially for semantic-web, research, and exchange documents. The complexity (i.e. depth and width) is again quite low.
- XML schemes usually provide too general information, whereas the instance documents are much simpler and more specific.

Discussion

The previous overview of existing analyses of XML data brings various interesting information. In general, we can observe that the real complexity of both XML documents and their schemes is amazingly low.

Probably the most surprising findings are that recursive and mixed-content elements are not as unimportant as they are usually considered to be. Their proportional representation is more than significant and in addition their complexity is quite low. Unfortunately, effective processing of both the aspects is often omitted with reference to their irrelevancy. Apparently, the reasoning is false whereas the truth is probably related to difficulties connected with their processing.

Another important discovery is that the usual depth of XML documents is small, the average number is always less than 10. This observation is already widely exploited in techniques which represent XML documents as a set of points in multidimensional space and store them in corresponding data structures, e.g. R-trees [11], UB-trees [3], BUB-trees [10] etc. The effectiveness of these techniques is closely related to the maximum depth of XML documents or maximum number of their simple paths. Both of the values should be of course as small as possible.

Next considerable fact is that the usage of schemes for expressing allowed structures of XML documents is not as frequent as it is expected to be. The situation is particularly wrong for XSDs which seem to appear sporadically. And even if they are used, their expressive power does not exceed the power of DTDs. The question is what is the reason for this tendency and if we can really blame purely the complexity of XML Schema. Generally, the frequent absence of schema is of course a big problem for methods which are based on its existence, e.g. schema-driven database mapping methods [19].

Concerning the XML schemes there is also another important, though not surprising finding, that XML documents often do not fully exploit the generality allowed by schema definitions. It is striking especially in case of types of recursion but the statement is valid almost generally. Extreme cases are of course recursion that theoretically allows XML documents with infinite depth or complete subgraphs typical for document-centric XML documents. This observation shows that although XML schemes provide lots of structural information on XML documents they can be too loose or even inaccurate.

The last mentioned analysis indicates, that there are also types of constructs (such as simple mixed contents, DNA patterns, or relational patterns etc.), that are quite common and can be easily and effectively processed using, e.g., relational databases. Hence we can expect that a method that focuses on such constructs would be much more effective than the general ones.

Last but not least, we must mention the problem of both syntactic and semantic incorrectness of analyzed XML documents, DTDs, and XSDs. Authors of almost all previously mentioned papers complain of huge percentage of useless sample data − an aspect which unpleasantly complicates the analyses. A consequent question is whether we can include schema non-determinism and ambiguity into this set of errors or if it expresses a demand for extension of XML recommendations.

Conclusion

The main goal of this paper was to briefly describe, discuss, and classify papers on analyses of real XML data and particularly their results and findings. The whole overview shows that the real data show lots of regularities and pattern usages and are not as complex as they are often expected to be. Thus there exists plenty of space for improvements in XML processing based on this enhanced categorization.

Acknowledgement

This work was supported in part by the National Programme of Research (Information Society Project 1ET100300419).

References

1. *The Extensible HyperText Markup Language (Second Edition).* W3C Recommendation, August 2002. `http://www.w3.org/TR/xhtml1/`

2. D. Barbosa and L. Mignet and P. Veltri. Studying the XML Web: Gathering Statistics from an XML Sample. In *World Wide Web*, pages 413-438, Hingham, MA, USA, 2005. Kluwer Academic Publishers.

3. R. Bayer. The Universal B-Tree for Multidimensional Indexing: General Concepts. In *WWCA '97, Worldwide Computing and Its Applications, International Conference*, pages 198-209, Tsukuba, Japan, 1997. Springer.

4. D. Beckett. *RDF/XML Syntax Specification (Revised).* W3C Recommendation, February 2004. `http://www.w3.org/TR/rdf-syntax-grammar/`

5. G. J. Bex and F. Neven and J. Van den Bussche. DTDs versus XML Schema: a Practical Study. *WebDB '04, Proceedings of the 7th International Workshop on the Web and Databases*, pages 79-84, New York, NY, USA, 2004, ACM Press.

6. P. V. Biron and A. Malhotra. *XML Schema Part 2: Datatypes Second Edition.* W3C Recommendation, October 2004, `www.w3.org/TR/xmlschema-2/`

7. T. Bray and J. Paoli and C. M. Sperberg-McQueen and E. Maler and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Third Edition).* W3C Recommendation, February 2004, `http://www.w3.org/TR/REC-xml/`

8. B. Choi. What are real DTDs like?. In *WebDB '02, Proceedings of the 5th International Workshop on the Web and Databases*, pages 43-48, Madison, Wisconsin, USA, 2002, ACM Press.

9. D. C. Fallside and P. Walmsley. *XML Schema Part 0: Primer Second Edition.* W3C Recommendation, October 2004, `www.w3.org/TR/xmlschema-0/`

10. R. Fenk. The BUB-Tree. In *VLDB '02, Proceedings of 28th International Conference on Very Large Data Bases*, Hong Kong, China, 2002, Morgan Kaufman Publishers.

11. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD'84, Proceedings of Annual Meeting*, pages 47-57, Boston, Massachusetts, 1984, ACM Press.

12. M. Klettke and L. Schneider and A. Heuer. Metrics for XML Document Collections. In *XMLDM Workshop*, pages 162-176, Prague, Czech Republic, 2002.

13. J. Kosek and M. Kratký and V. Snášel. Struktura reálných XML dokumentů a metody indexovaní. In *ITAT 2003 Workshop on Information Technologies Applications and Theory*, High Tatras, Slovakia, 2003. (in Czech)

14. A. McDowell and C. Schmidt and K. Yue. Analysis and Metrics of XML Schema. In *SERP '04, Proceedings of the International Conference on Software Engineering Research and Practice*, pages 538-544, 2004, CSREA Press.

15. L. Mignet and D. Barbosa and P. Veltri. The XML Web: a First Study. In *WWW '03, Proceedings of the 12th international conference on World Wide Web, Volume 2*, pages 500-510, New York, NY, USA, 2003, ACM Press.

16. I. Mlýnková and K. Toman and J. Pokorný. *Statistical Analysis of Real XML Data Collections.* Technical report 2006/5, Charles University, June 2006, `http://kocour.ms.mff.cuni.cz/ mlynkova/doc/tr2006-5.pdf`

17. M. Murata and D. Lee and M. Mani. Taxonomy of XML Schema Languages using Formal Language Theory. In *Extreme Markup Languages*, Montreal, Canada, 2001.

18. A. Sahuguet. Everything You Ever Wanted to Know About DTDs, But Were Afraid to Ask (Extended Abstract). In *Selected papers from the 3rd International Workshop WebDB 2000 on The World Wide Web and Databases*, pages 171-183, London, UK, 2001, Springer-Verlag.

19. J. Shanmugasundaram and K. Tufte and C. Zhang and G. He and D. J. DeWitt and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*, pages 302-314, Edinburgh, Scotland, UK, 1999, Morgan Kaufmann.

20. H. S. Thompson and D. Beech and M. Maloney and N. Mendelsohn. *XML Schema Part 1: Structures Second Edition.* W3C Recommendation, October 2004, `www.w3.org/TR/xmlschema-1/`

First eXist Workshop

Wolfgang Meier (eXist XML database)

Abstract

We would like to give eXist users and developers room to learn from each other, present projects, discuss ideas, problems and our roadmap for the future. The schedule is still very much open (time planned: 9am to 2pm). All community members are invited to contribute and propose presentations or topics to discuss. In particular, we would be interested to see presentations on concrete projects you are working at, including problems you encountered or wishes you may have. The workshop should not be too much developer-oriented.

DocBook BoF

Jirka Kosek (www.kosek.cz)

Abstract

Are you using DocBook and want to know which new features are available in a brand new DocBook version 5.0? Then come and join DocBook BoF. During this BoF Jirka Kosek will present new features of DocBook V5.0. You will see how to edit, validate and process DocBook V5.0 content and how easily you can create customized DocBook versions using RELAX NG schema language. At the end of the session there will be enough time do discuss various DocBook issues with other users.

Jirka Kosek is a member of OASIS DocBook TC and a developer of open-source XSLT stylesheets for processing DocBook content. Do not miss this unique chance to meet core DocBook developer and register for XML Prague 2006 today.

DocBook V5.0 `http://www.docbook.org/specs/wd-docbook-docbook-5.0a1.html`

Perl XML BoF

Petr Cimprich, Petr Pajas (Ginger Alliance, Charles University, Prague)

Abstract

Perl is a powerful environment to work with XML, though not always easy for new users. CPAN contains several hundreds of Perl XML modules and choosing the right modules for your project requires some experience. During this BoF, you can meet with authors and experienced users of Perl XML CPAN modules, to learn from them, and to contribute by your own knowledge.

Xdefinition BoF

Václav Trojan (Syntea)

Abstract

Xdefinition is a tool that enables the description of both the structure and the properties of data values in an XML document. Moreover, the Xdefinition allows the description of the processing methods of specified XML objects. With Xdefinition it is possible to validate XML documents and to describe most of the XML transformations.

Xdefinition enables the merging in one source of both the validation of XML documents and processing of data (using "actions"). Compared to the "classical" technologies based on DTD and XML schemas, the advantage of Xdefinition is higher readability and easier maintenance. Xdefinition has been designed for processiong very large XML data files, up to many gigabytes in size. Moreover, Xdefinition may serve as the tool for both description and implementation of metalanguges based on XML technologies.

A basic property of Xdefinition is maximum respect for the structure of the described data. The form of Xdefinition is − as described for XML data − an XML document with a structure similar to the described XML origin data. This allows you to quickly and intuitively design Xdefinitions for any given XML data. In most cases this requires just the replacement of XML data values with simple scripts. You can also gradually add to your script the required actions for data processing, so you can take a step-by-step approach to your work.