Using XSLT and XQuery for life-size applications

Michael Kay

Saxonica Limited

mike@saxonica.com

Overview

This paper discusses the role of the XSLT 2.0 and XQuery 1.0 languages when it comes to writing real-life, sizeable applications for performing data transformations: especially factors such as error handling, debugging, performance, reuse and customization of code, relationships with XML Schema and other technologies such as XForms, and the use of pipeline-based application architectures. It will be based on actual case studies of real applications built using both technologies by the speaker's consulting clients, suitably abstracted.

There's a lot more practical experience of building large applications with XSLT than there is with XQuery, because XSLT has a head start of six years or more. One can also expect XQuery to improve in this area over the next 12 months, because facilities needed for programming-in-the-large will tend to appear as the products acquire greater maturity. Nevertheless, it's starting to become possible to collect experience of how XQuery application development scales up.

The paper starts with an overview of the kind of applications we are talking about, and some general points about the architecture of such applications and the choice of technologies to implement them. It identifies four main areas of technology where decisions need to be made: the database engine, the application development tools, the user interface technology, and the middleware that binds the rest together.

The paper then homes in on the application development part, with an analysis of the relative merits of XQuery and XSLT for implementing business logic.

In some of respects XSLT and XQuery show quite similar characteristics, in other areas they have marked differences. The aim of the paper is not simply to stage a beauty contest between the two languages, rather it is aimed at showing where both languages still have limitations, and the kind of ways in which developers can work around these limitations to create robust and maintainable applications using either language - or both in combination.

Setting the Scene

I'll start by stating some assumptions about the kind of application we're interested in.

- They're XML-based, of course. As far as possible, in my view, that means wall-to-wall XML: XML in the database, XML-based object representations for the "business logic", XML-based user interface components. Of course, there will be components and interfaces using other technologies, but we'll try and use XML wherever we can.

- They are not just "data plus presentation": they involve some real information processing, with multiple users, multiple business processes, in many cases multiple collaborating enterprises.

- They deliver significant value and therefore justify significant investment.

The factors that encourage people to develop business applications using an XML-centric architecture are varied. Very often the use of XML starts because it's being used for information interchange between multiple applications, perhaps in different organizations, and it grows from there. Sometimes there's an existing business process implemented using an "email and spreadsheet" approach, and the aim is to preserve the flexibility of such a system while providing management with the ability to monitor and control what's going on. Sometimes information is the end-product, and the purpose of the application is to gather, digest, and analyze the information for delivery to the customer.

Most of these applications are somewhere in the middle of a spectrum from pure "document-oriented" applications at one end to pure "data-oriented" applications at the other. At the extreme document end of the spectrum you're only interested

in cataloguing, storing, and transmitting documents and you don't care about the information they contain — so you don't need XML. At the pure data end, you're only interested in tabular data, so you might as well use SQL. Most real applications are somewhere between those extremes, and that's why XML has become so popular.

I find there are many applications where one can think of the data in terms of business documents. Apart from the obvious examples of invoices and purchase orders, we can describe most business processes in terms of the documents that flow around the system: an employee appraisal, an insurance claim, a tax return, a safety inspection report, an investment case. Data modelling often encourages you to analyze such documents to determine the underlying entities, attributes, and relationships; but I find that with an XML-based application architecture, the objects in the system sometimes mirror exactly the documents in the user view of the business process. This leads to a situation where the application is much more comprehensible to users, because the internal view matches the external view very closely, and it also makes things easier to change, because document designs can evolve over time. Many of the most successful XML applications that I've seen are document-based in this sense: they appear to the user, and are implemented internally, as processes that manage a document-based workflow. Even a fairly simple transaction such as an online purchase from a web site can be readily understood in these terms.

The effect of thinking this way is that the data in the system tends to be oriented around "the things that happen in the world" rather than "the things that exist in the world". Instead of doing the kind of traditional analysis that asks "what are all the attributes of an employee", an employee as represented in the computer system becomes simply a set of documents defining everything that has ever happened to the employee — which makes the database look much more like the contents of the filing cabinet held in the personnel department, and greatly increases the flexibility of what can be stored and queried. Saxonica's customer database is exactly like that: it's simply a record of all the interactions with a customer over time. This approach greatly simplifies database design: there's no need to worry, for example, about whether a customer can only have one address or phone number. Everything is naturally multi-valued, and just as in the real world, internal inconsistency in the data doesn't bring the system to a halt.

There's often a need for some reference data as well, of course: the business process for issuing quotations probably relies on having an up-to-date price list. Sometimes it will make sense to hold the reference data in SQL form rather than

in XML.

Technical Components

What technical components do we need to build this kind of application architecture? The main ones, I think, are:

- Database technology: used for long-term storage of information, supporting query access.
- Application development tools, in particular programming languages, for coding the business logic.
- User interface technology for displaying information to users and gathering their input.
- Middleware, for binding the components and allowing them to interoperate in a distributed environment.

On the database side, there's a choice between relational engines, XML engines, and XML-enabled relational engines. There are also other technologies that we sometimes forget about, such as LDAP directories, which often play an important role. Personally I'm a fan of what I still like to call "native XML databases" – a term which to me means databases that were born to do XML, as distinct from databases that underwent a mid-life conversion to do XML. I think you get a level of elegance and simplicity in a database that supports a single data model which you can never match in a system that tries to support multiple data models. With a system that supports both relations and XML documents, you spend your life trying to decide which kind of representation to use for each kind of data object, and probably regretting the choice later. I think it's better to use a single data model, and if you're going to process the data as XML then I think it's better to store it and query it as XML. But as I've already mentioned, there will sometimes be good reasons (if only legacy reasons) for holding reference data such as price lists in a relational database.

For application development, the first choice is between declarative languages (XSLT or XQuery) and procedural languages (Java or C#). I would put XML-enabled procedural languages such as Linq in the second category. This can sometimes be a bloody battleground, where developers have strong opinions one way or the other. I have strong opinions too: I think there are substantial benefits

to be gained from using higher-level languages to express the business logic. XSLT has a steep learning curve, and one can make many criticisms of its usability: the surface syntax is verbose, many common tasks require arcane workarounds (at least in version 1.0), processing can be slow. But in the end, those obstacles turn out to be superficial, and when it comes to the bottom line, my experience is that XSLT delivers the goods. We'll look in the next section at how it stacks up against XQuery. By contrast, I find use of Java with XML very frustrating. You either go for a generic API such as DOM or XOM, in which case you are constantly writing low-level navigation code (even if you combine it with XPath), or you use a data-binding approach, in which case you are for ever fighting the mismatch between the XML data model and the Java object model. The data-binding approach can also lose all the flexibility benefits that an XML-based application architecture should bring; changes to the document schema cause too much of a ripple effect in the applications.

For User Interface technology there are also a number of choices. XSLT is pretty well established as the chosen way of rendering XML documents for the browser screen, though it's by no means the only option. This creates an odd asymmetry, however, because XSLT is a one-way technology: it does nothing to enable user interaction or to help you get the user's input back to the application. XForms was invented to fill this gap, and I've seen people using it very successfully, especially as a server-side technology. What happens here is that your presentation XSLT constructs a document containing XHTML mixed with XForms controls; this is passed to a server-side component that translates the XForms into HTML and Javascript (and these days, Ajax) to execute in the browser. The user-supplied data that comes back with the next HTTP GET or POST request is then intercepted by the XForms engine, which packages it into an XML document that's passed to the appropriate module of the business application to process. The business application thus takes XML as input and produces XML as output, pausing on the way to read or write some XML data from or to the database.

Finally, middleware. I think there's a danger with XSLT and perhaps also with XQuery of making applications too monolithic. The best way to do a complex transformation is one step at a time, and it's best for each step to be a separate stylesheet or query. This makes the individual stylesheets and queries easier to debug and easier to maintain, and it makes it much easier to reuse code. The pipeline processing model works extremely well here: each component of the application transforms an XML document into another XML document, and the application as a whole is constructed as a pipeline of such transformations.

To take a simple example, suppose you have to handle RSS as your input. There are different versions of RSS that use different namespaces. Writing a stylesheet that can handle both versions is tedious and error-prone. Instead, write your code to handle only one of the versions, and create a separate transformation that converts one version into the other. Not only have you simplified your logic, you have created a component – an RSS 0.9 to RSS 1.0 converter, say – which can be reused any number of times in other processing pipelines. (In fact, you probably don't have to write it at all, the chances are someone has already done the job.)

Managing a pipeline of transformations requires some kind of infrastructure. You can write this yourself in Java; or you can use a product that specializes in XML pipelines, such as Orbeon's presentation server (http://www.orbeon.com/) or the MT Pipeline product from Markup Technologies (http://www.markup.co.uk/). Alternatively, similar capabilities are sometimes available in content management products, or you might prefer to use something more general-purpose such as Ant. A new W3C activity has started whose ultimate aim is to standardize this level of technology: I will be watching its progress with interest.

One of the advantages of a pipeline-based application architecture is that different steps in the pipeline (that's a mixed metaphor, but I don't know how to avoid it!) can be implemented using different technologies. For example, one step might be written in XSLT, another in XQuery, and a third in Java. This allows you to use the most appropriate tool for each task, and to reuse components regardless of the technology that was used to implement them.

XSLT vs XQuery

XSLT has become well-established as the preferred tool for rendering XML. Similarly, few people question that XQuery is the preferred tool for getting XML data out of XML databases. In between, the implementation of business logic is disputed territory. Which language works best in this area?

Many people appear to be much more comfortable with one language than the other. This is perhaps a little surprising, because the languages are remarkably similar: they are both declarative functional languages, they share a common data model (XDM), a common sublanguage (XPath), and a common function library, and they share very similar syntax for constructing new XML documents.

The are several things that make a straight comparison difficult. One is that there are two versions of XSLT to consider: XSLT 1.0, which most people are actually

using today, is the version that is likely to influence perceptions, but it's fairer to compare XSLT 2.0 with XQuery 1.0 since they are at the same point on the standards track. (Or is it? There are a many XQuery 1.0 implementations to choose from, whereas my own product, Saxon, still has rather a monopoly of the XSLT 2.0 space.) More importantly, you can only evaluate a tool in relation to a particular task.

There was a recent usability study conducted by Joris Graaumans [1] which ran some careful experiments designed to compare the performance and satisfaction of a set of trial users of both languages ("performance" here measures both the quality of the code written and the time taken to write it). Joris concluded:

> This study shows that the usability of XQuery is higher compared to XSLT. XQuery is easier to use because the expressions in the language are more compact and because XQuery embeds XPath in a more simple way compared to XSLT. Furthermore, query complexity is a good predictor of performance on a query task. Finally, user experience has a large influence on the performance with a query language. The influence of different XML structures on performance with a query language are not shown in this study. The query tasks and the subjects in the experiments are representative for the start of the learning curve of these languages.

The trouble about this kind of study is that the users are students and the tasks are very small. Within its scope, I would not dispute the conclusions: XQuery is a smaller and more compact language, it is easier to learn, and it is easier to use when doing simple tasks. The question is, what results do you get when you apply the same tests to experienced users and to life-size applications? Unfortunately, no-one can afford to do controlled experiments at this level, so one has to fall back on anecdotal experience.

There's another problem with this study, which is betrayed by its title: *Usability of XML Query Languages*. When you look at the tasks in detail, they are all query tasks, not transformation tasks. You would expect XQuery to be better at query, and XSLT to be better at transformation, so the results should not be surprising.

Are the two tasks really different? I think they are. When you look at something which is described as a transformation task (like the example earlier, convert RDF0.9 to RDF1.0) then in my experience XSLT often has a much more elegant solution to the problem. With problems expressed as queries (*How many tomatoes did we export to Uruguay last year?*) the XQuery formulation is usually much simpler.

It's interesting that Graaumans found the conclusions equally valid when applied to different kinds of XML structure (that is, "documents" versus "data"). This tends to counter the common perception that XQuery is designed more for "data", and XSLT more for "documents". However, if you examine the tasks more closely, the tasks performed on "documents" are still very much queries: *Select the chapter that has a number higher than one and the title 'Getting to know SGML'.*

Graaumans has shown that XQuery is better than XSLT at query tasks. My experience (both direct and hearsay) of projects that have tried to use XQuery for writing the business logic of an application, however, is less positive. I think this is partly because, given a pipeline architecture as discussed in the previous section, many of the components in the pipeline will be transformations.

Here are some of the problems that I've experienced when trying to use XQuery for larger applications. (I've deliberately not included applications whose primary task is document rendition, because I think XSLT's advantages in that area are clear-cut.)

- Many transformations involve leaving large chunks of the document unchanged, while modifying other parts. There's a natural design pattern for this in XSLT: write an identity template rule that matches everything and copies it unchanged; then override this with specific template rules for the elements that need to be modified. There's no equivalent to this in XQuery, and this makes it surprisingly difficult to perform a simple task like "copy the document, deleting all the NOTE elements". In time, XQuery will probably handle this kind of requirement through an update mechanism: rather than copying the document with changes, you will modify it in-situ. But the update facilities are not yet available.

- In an application of any size, there's a need to create versions and variants. An online banking service for private customers is likely to be very similar to the service for small businesses, but with differences. The lack of any polymorphism in XQuery makes this kind of situation very difficult to handle. You're back to the old pre-OO approach of injecting conditional statements wherever the logic varies. By contrast, XSLT has a great deal of flexibility in this area, even in version 1.0: not only the dynamic invocation of template rules, but the ability to override templates, variables, and other definitions in an importing stylesheet module. These facilities are missing from XQuery for a good reason: polymorphism and dynamic binding make

static optimization difficult, and in a database query scenario the ability to optimize queries is a very high priority. But the effect of this decision is to make XQuery less suitable as a transformation language.

- XQuery is not XML. The fact that XQuery's syntax is not an XML syntax (though it gets close) doesn't matter when writing small queries, whether these are ad-hoc queries entered interactively, or canned queries embedded in a Java application or shell script. In fact, in those situations it's a positive advantage, because it makes the query more compact and more readable. Once you start programming in-the-large, however, the fact that XSLT is an XML-based language starts to become useful. A remarkably high proportion of the applications I see actually manipulate stylesheet source code in some way, either by reading it, modifying it, or generating it. Life-size applications often include an element of introspection in their design. The granularity at which XSLT uses XML syntax in fact appears to have been a good design choice from this point of view: although the split between the XML-based structure of XSLT instructions and the micro-syntax used for XPath expressions appears arbitrary, it provides a good compromise between making the language readable and writable by humans and making it accessible to machine processing. By contrast, neither of the two syntaxes available for XQuery – the ordinary "human readable" syntax and the very fine-grained XQueryX representation – manage to hit this sweet spot.

  A surprising consequence of this is that it's very straightforward to store your XSLT stylesheets in an XML database and to use XQuery to retrieve them. You can then easily find out, for example, all the modules that contain calls to a particular named template. It's much harder to store your XQuery code in an XML database: although you can store the modules in XQueryX form, the schema is so fine-grained that it's not easy to formulate meaningful queries.

- Compared with XSLT 2.0 in particular, XQuery is weak at tasks that involve grouping or pattern matching. The availability of the `distinct-values()` function is an improvement on XSLT 1.0, but it doesn't come close to the ease-of-use of XSLT 2.0's `<xsl:for-each-group>` construct, and in many use cases it's likely to suffer $O(n^2)$ performance because having found the distinct values present in a set of nodes, you then need to go back and locate the nodes that match each of these values. There's nothing at all available for positional grouping (identifying groups

of nodes that are related by position rather than by sharing common values) beyond the ability to write low-level recursive functions. Similarly, for processing structured text, the capabilities of XSLT 2.0's `<xsl:analyze-string>` go well beyond anything available in XQuery. These are examples of areas where XSLT wins by being several years ahead of XQuery in its life-cycle: there are many features in XSLT 2.0 that respond to the feedback obtained from widespread use of version 1.0 of the language, whereas far less experience is available for XQuery, and even where the shortcomings are known, the requirements have been put to one side because of the urgency of getting a 1.0 specification finished.

- XQuery places much more reliance on a good optimizer. Slow-running XSLT applications can almost invariably be speeded up by judicious use of keys. While some XQuery applications may have "out of band" options to control tuning, there's nothing comparable in the language.

  There's a real clash of cultures here about whether such features are desirable. In the database world, there seems to have been a consensus for about 30 years that it's OK for users to say which fields they want to be indexed, but it's not OK for them to say in their queries when they want the index to be used. XSLT was designed with no such inhibitions. Which is right? When I see some of the articles in database system documentation explaining to users the tricks they can use to change the behaviour of the optimizer, I sometimes wonder.

  In fact, it's possible to do a lot more optimization on XSLT code than is sometimes thought. Because expectations are higher in the XQuery world, I only started doing things such as join optimization in Saxon to improve my rating in XQuery benchmarks, but the changes actually benefit many XSLT users as well. But I don't feel that there's anything wrong with the XSLT language providing an explicit way for users to say "I want to do an index loop-up at this point rather than a sequential search." When you're developing life-sized applications, I think it's useful to have explicit control over big performance trade-offs rather than having to constantly second-guess the optimizer.

- Schema-awareness. The facilities for schema-aware processing in XSLT 2.0 and XQuery 1.0 were developed very much in parallel, so you might expect that there are few differences. There's relatively little "public" experience of schema-aware processing at the moment: Saxonica has

customers developing big applications both with XSLT and with XQuery, but the people who recognize the benefits that schema-aware processing brings to "life-size" application development (see [2]) tend to be enterprises who keep their new knowledge to themselves. I did find one minor difference between XSLT and XQuery schema-awareness, however, that turned out to make a big difference to one particular application. XQuery allows you to validate the content of a constructed element only against a global element definition in the schema. XSLT allows you also to validate it against a global type definition. In the real world there are some big schemas – FPML is an example – that define lots of global types but very few global elements. If you're working with that kind of schema, then XSLT gives you much more fine-grained type checking of the results of individual functions or templates in your stylesheet than you get with XQuery. This in turn gives you a better chance that your mistakes will be found at compile-time, and therefore a better chance that the bugs will be found before you go live.

Conclusions

Firstly, I've tried to persuade you of a few beliefs about the design of XML-based applications:

- Try and use XML wall-to-wall rather than mixing different models
- Use XML databases wherever it makes sense.
- Think in terms of designing your applications around business documents, storing XML documents that describe workflow as it happens rather than trying to model the state of the world at an instant in time.
- Consider use of a pipeline architecture for connecting the components of your application, to achieve component modularity and reuse. One of the benefits is that different steps in the pipeline can be implemented using different technologies.
- Consider using XForms to close the loop with the user: XSLT to produce the output that the user sees, XForms (with Ajax) to capture the user's response
- Use the declarative languages XSLT and XQuery for writing the business

logic, rather than coding it in procedural languages like Java and C# (with or without data binding)

Then I spent some time discussing the merits of XSLT and XQuery.

- XQuery wins when doing queries, especially small queries. When you're extracting a small amount of information from a large amount of data, XQuery is the tool for the job

- XSLT was initially designed with a strong focus on publishing and rendition tasks, and it's definitely the preferred tool for that job

- When you're writing the business logic of the application, there's an overlap in functionality and personal preferences play a strong part in making your choice. However, I've tried to show some of the areas in which XSLT is more mature as a language for writing life-size applications. XQuery may catch up and overtake it in future, but at present, I would usually recommend XSLT in this role.

References

[1] Usability of XML Query Languages, Joris Graaumans, Ph.D Thesis, University of Utrecht, 2005. ISBN 90-393-4065-X

[2] Schema-Aware Queries and Stylesheets. Michael Kay, 2006. Published online at http://www.stylusstudio.com/schema_aware.html