XSLT and XPath Optimization

Michael Kay Saxonica LimitedReading England http://saxon.sf.net/

Biography

> Michael Kay is the developer of the Saxon open-source XSLT and XQuery processor, and the author of *XSLT Programmer's Reference* published by Wiley. He is editor of the XSLT 2.0 specification and is also an invited expert on the XQuery working group. He has recently started his own company, Saxonica Limited, to continue the development of Saxon and provide support to Saxon users.

Table of Contents

Introduction

This paper describes the main techniques used by the Saxon XSLT and XQuery processor (http://saxon.sf.net/) to optimize the execution of XSLT stylesheets and XPath expressions, and reviews some additional XSLT and XPath optimization techniques that are not (yet) used in Saxon.

The primary focus is on XPath rather than XSLT, partly because Saxon does relatively little optimization at the XSLT level (other than in the way pattern matching works).

Recent releases of Saxon support XQuery 1.0 as well as XSLT 2.0. However, the XQuery processor is an in-memory processor, not a database query engine. Optimizing database queries is an entirely different art, because it relies so heavily on rearranging queries to exploit persistent indexes. An in-memory processor does not have this opportunity, because the only indexes available are those that are constructed transiently for the duration of a transformation or query.

Saxon's processing can be divided into three phases: parsing, static analysis, and run-time execution. These are described in the three main sections of this paper.

Parsing

The actual structure of the Saxon parser is not directly relevant to this paper, but it is worth describing it briefly, because it constructs the data structures that subsequent phases operate on.

The main point of interest is that instead of being a conventional two-stage parser (the stages being lexical analysis and syntax analysis), it is best regarded as a three-stage parser (the stages being tokenizing, symbol recognition and syntax analysis).

The tokenizer simply reads primitive tokens from the input string. At this stage there is no analysis of the roles of different kinds of name – they are not categorized as QNames, function names, operators, axis names, or keywords.

The main role of the symbol recognizer is to perform this classification. Generally this uses some local context information, specifically a token is analyzed in relation to the immediately preceding and following tokens. This process recognizes function names by the presence of a following "(" token, axis names by the presence of a following "::", and the keywords "for", "some", and "every" by the presence of a following "$". Variable names are recognized by a preceding "$", and operators (such as "and") by the fact that the preceding token was not an operator. Pairs of tokens that comprise a single

terminal symbol (such as "instance of" and "cast as") are combined to deliver a single symbol to the syntax analyser.

Parsing of XQuery is considerably more complex than XPath parsing, because of the need to switch between the different grammatical styles used in the query prolog, in conventional expressions, and in the XML-like element constructors. XSLT parsing, of course, is trivial, because the difficult part is done by an XML parser.

The syntax analyser in Saxon is a hand-written recursive descent parser. This is the result of organic growth; the grammar has really grown too large for this technique to be appropriate. It works well with XPath 1.0, but XPath 2.0 and particularly XQuery really need something more efficient.

The output of parsing is (as normal) an abstract syntax tree. Each node in this tree represents an expression. Some simple normalization is done during the parsing phase, for example expansion of "//" into "/descendant-or-self::node()/", expansion of "@" into "attribute::", and ".." into "parent::node()". References to variables and functions are fixed up during parsing where possible, but where necessary the reference is added to a fixup list, and is resolved as soon as the corresponding declaration is encountered (XSLT allows forwards references to global variables and to functions defined in the stylesheet).

All QNames (including element and attribute names, variable names, function names, and type names) are analyzed during parsing and registered in a NamePool; once registered, each name can be identified by a unique 32-bit integer. This records the prefix as well as the URI and local-name; when two names are compared, the prefix information can be masked out (the URI and local-name are indexed by the bottom 20 bits). Generally the binding of prefixes to URIs is known by the time the expression is parsed; there is one exception to this, in XQuery, where an element constructor can legally take the form:

```
<code item="{/xx:a/xx:b[@size="3.5"""] (:" means "inches":)}" xmlns:xx="some.uri"/>
```

It's not possible here to identify the end of the `item` attribute without parsing it. Saxon handles this by ignoring any namespace errors on its first attempt to parse the attribute, and reparsing it if necessary when all the namespace declarations have been read.

After any function or template declaration has been parsed, slots in a local stack frame are allocated to each of the local variables, and references to each variable are visited to mark them with the chosen slot number. There is no use of variable names or function names at run-time (even in the form of numeric NamePool codes).

Static Analysis

When processing XSLT, Saxon reads the whole stylesheet before parsing any XPath expressions. This means that references to functions and variables can always be resolved during parsing. In XQuery this is not generally possible, because there is no separate XML parsing phase, and functions can be mutually recursive. Even in XSLT, however, it is necessary to revisit variable references and function references to add type information after the variable and function declarations have been read. If the types of variables and functions are declared explicitly (using the new `as="xs:integer"` syntax available in XSLT 2.0) then this information is available during the first phase of processing, but more commonly types are left undeclared, in which case Saxon infers the type of a variable from the expression to which it is bound, and then uses this information in inferring the type of expressions in which the variable is used. Clearly this process is potentially cyclic, and in fact Saxon does not go to inordinate lengths to pursue this process to its logical conclusion: each expression is visited once only to do type inference, and it uses whatever information is available at that time about the types of variables and functions referenced in the expression.

At the time of writing, a schema-aware version of Saxon has been developed, but has not yet been released. In this first version, schema types are checked dynamically by the validator, but there is little or no use made of type information in the schema for optimization. The static type inferencing that is performed doesn't go beyond the level of built-in types: it can distinguish integers, strings, elements, and attributes, but not purchase orders and vehicle registrations.

Each XPath expression in a stylesheet goes through two stages of static analysis after parsing. Both of these work essentially by a recursive walk of the abstract syntax tree. The two stages are identified by the (not entirely appropriate) method names simplify() and analyze().

The first phase, simplify(), is largely responsible for context-independent rewriting of the tree, and other local optimization of expressions. This phase has tended to decline in importance, as the more powerful optimizations can only be done in the second phase, when more information is available. In fact, many of the rewrites done during this phase could equally well be done during parsing. For example, this phase expands function calls such as `name()` into `name(.)`, and in the case of an XPath 2.0 function such as `matches()`, it precompiles the regular expression in the common case where it is supplied as a string literal.

Generally speaking, static properties of expressions are computed lazily, the first time the property is required, and are then stored on the tree for use later. In some cases the property might not be computed until run-time, though this is rare. The most obvious static property of an expression is its type, which in the XPath 2.0 data model has two parts: the cardinality (0:1, 1:1, 1:many, or 0:many), and the item type (for example xs:integer or attribute(*,*) or comment()).

However, there are many other interesting properties of expressions that have nothing to do with the type. The most important of these are the *dependencies* of the expression, which define which parts of the evaluation context the expression depends on (including dependencies on variables). This information greatly affects the ability to rewrite the expression, for example by moving a subexpression out of an XPath predicate. In general, the dependencies of an expression are the union of the dependencies of its sub-expressions. Certain expressions have obvious direct dependencies, for example "." depends on the context item and "position()" depends on context position. Some are more subtle, for example a filter expression with a numeric predicate (or a predicate that might be numeric) also depends on the context position. An expression such as name() depends on the context node because it has an implicit argument which is the context node. This analysis can therefore only be done on an expression-by-expression basis. In some cases the expression masks dependencies in its sub-expressions, for example $x[@a] does not depend on the context node even though @a does.

Other properties include, for an expression that delivers a sequence of nodes, the fact that the nodes are (or are not) in document order, and the fact that the nodes must all belong to the same document tree (if they do, then a path expression beginning with "/" within a predicate has the same value for all the nodes in the sequence).

For path expression, two important properties are the *peer* and *subtree* properties. The *peer* property is true if the expression will never return a sequence containing two nodes, one of which is an ancestor of the other. The *subtree* property is true if every node returned by the expression will have the original context node as an ancestor. These properties are important because in a path expression `A/B`, it is possible to avoid sorting the result into document order provided that A and B both deliver results in document order, and A has the peer property, and B has the subtree property. This rule avoids sorting for the great majority of path expressions encountered in practice. For example, X/Y and X//Y and X/Y//Z never need to be sorted, but X//Y/Z does (because X//Y does not have the peer property). The rule also applies where the subexpressions are not axis steps, for example A might be a call on the document() function.

An important property that is not yet fully implemented in Saxon is the *creative* property. An expression is creative if it is capable of constructing new nodes and returning results that depend on the identity of those nodes. In XQuery, constructor expressions such as `<a/>` are obviously creative, but creative expressions can also be encountered in XSLT 2.0: any XPath function call to a stylesheet function that creates a new node and returns it is creative. An expression is not creative if it creates a new node and then atomizes it to extract its value, because when that happens, two evaluations of the expression will return indistinguishable results. Creative expressions, because they depart from the purely functional nature of XPath as an expression language, are a fly in the ointment for the optimizer. For example, path expressions such as `A/B/C` are normally associative: they can be evaluated either as `(A/B)/C` or as `A/(B/C)`. But if one of the subexpressions is creative, this rule breaks down. If `foo()` creates a new node, then `count((A/B)/foo())` gives a different answer from `count(A/(B/foo()))`.

Most of the important static optimizations are done during the second phase, analyze(). These fall into a number of categories:

1. Early evaluation of constant sub-expressions (known, for some curious reason, as "constant folding"). This is another optimization that cannot be done if the result is sensitive to node identity. Constant sub-expressions are surprisingly common in XSLT, because global variables often have a constant value; and it is often possible to get rid of large chunks of stylesheet code by pre-evaluating a condition such as `<xsl:if test="(system-property('xsl:vendor')='Xalan')">`.
2. Local rewrites such as replacing `count(X) = 0` by `empty(X)`. (This avoids the need to distinguish a sequence with a thousand items from one with a thousand and one). These rewrites are local in the sense that they only require looking in the immediate vicinity of a node in the abstract syntax tree.

3. Early binding of polymorphic operators (especially comparison operators such as "=" and arithmetic operators such as "+") based on the types of their operands.
4. Adding code to do run-time type-checking and type conversion (such as atomization and numeric promotion) if static analysis shows that it is necessary. If static analysis shows that the supplied value will always have the required type, then this code is not generated.
5. Non-local rewrites. The most significant of these in Saxon is moving an expression out of a loop if it does not depend on any variables (range variables or context variables) that are set within the loop. A "loop" here can be a predicate, the right-hand side of the "/" operator, or the action part of a "for" expression. (Saxon does not at present do this optimization at the XSLT level, only at the XPath/XQuery level). This optimization too needs to be aware that creative expressions cannot always be safely moved.
6. Ordering rewrites. These fall into two categories: adding a sort, and removing it. This relates primarily to the requirement to deliver the results of certain expressions in document order. Saxon first adds a sort operator to the tree if the semantics require it and the expression is not "naturally sorted" (as determined using rules such as the peer/subtree rule given earlier). Then it removes the sort operator if the expression is used in a context where ordering is immaterial, for example an argument of one of the functions count(), max(), or of course unordered(). However, the scope for this is limited because in many contexts where there is no dependency on ordering, there is still a requirement to eliminate duplicates, and the simplest way to eliminate duplicates is by sorting.
7. Elimination of common subexpressions. Saxon currenly does this in only a few special cases, for example it rewrites the expression `A/B/C | A/B/D` as `A/B/*[self::C or self::D]`. In general, elimination of common subexpressions is greatly complicated by the need to consider dependencies on the context.

So much for Saxon's compile-time optimizations. Let's now look at what happens at run-time.

Run-time Evaluation

Saxon uses a great variety of techniques to ensure efficient run-time evaluation. Many of these are simply good coding practice, and are not really worthy of the name "optimization". This section describes some of the more strategic aspects of the evaluation strategy.

One very important consideration at run-time is to minimize the use of memory, and in particular the creation of Java objects. Saving memory almost invariably means saving time, and not only in stylesheets that are trying to transform 100Mb documents and are therefore thrashing the virtual memory. Saxon therefore uses an internal data structure to represent the tree data model that is designed primarily for efficiency in its use of memory. Since most transformations visit the average node in the source tree about once, it's generally true that more time is spent in building the tree than in navigating it, so there is no point spending a lot of time (and memory) building data structures that make navigation more efficient. (This is very different to an XQuery system that handles persistent documents on disk.)

Saxon evaluates XSLT instructions and XPath expressions by interpreting the expression tree that's produced as the final output of the compile time analysis phases. This expression tree includes not only expressions and instructions that correspond directly to constructs written by the user, but also internal operations such as type checking, atomization, and sorting into document order.

There are really two trees: an instruction tree representing the XSLT stylesheet, and expression trees representing the XPath expressions. But since the XPath expressions are rooted at the leaves of the XSLT instruction tree, it's quite legitimate to view this as a single tree, and this is certainly the interpretation that makes most sense in the XQuery case. (The run-time code in Saxon is identical whether the tree was generated from XSLT or from XQuery.)

However, the interpreters for the two parts of the tree work in rather different ways. This is to a considerable extent a legacy of the strong distinction between XSLT instructions and XPath expressions in XSLT 1.0, a distinction which has started to blur with XSLT 2.0, and which is non-existent in XQuery. Traditionally, XSLT instructions created nodes and wrote them to the result tree, while XPath expressions retrieved data from the source tree. This means that the XPath interpreter works in pull mode, while the XSLT interpreter pushes. What the two have in common is that they are very heavily pipelined.

The XPath pipeline uses iterators to evaluate sequences. In general, an expression is evaluated by calling its iterate() method, passing an object representing the dynamic context as an argument. This returns an iterator (Saxon's SequenceIterator class is modeled on the standard Java Iterator, but with some differences). Often this iterator will act as a front-end to some other iterator: to take an obvious example, a filter expression is evaluated by iterating over the base sequence and returning only

those items that satisfy the predicate. Similar iterators are used to implement path expressions, for expressions, atomization, type checking, and so on. This pipelined (or streaming) architecture has two big advantages: it means that there is no need to allocate memory to hold the intermediate results, and it means that no wasted work is carried out if the iteration terminates early. Early termination is very much a characteristic of XPath semantics: for example, the rules for calculating effective boolean value mean that only the first two items in a sequence need to be read, and any expression of the form E[1] needs to read only the first item. There are a few constructs where the pipeline has to be broken, notably sorting, but they are relatively rare.

Closely associated with pipelining is the idea of lazy evaluation. When a variable is evaluated, the system does not immediately allocate space to hold its value. Instead, it stores a *Closure* (another odd name, but it's the one that authors of books on compiler-writing always use). The Closure in effect contains a reference to the expression and a snapshot copy of the dynamic context (that is, the context node, position, and size, and the local variables. Global variables are immutable so they don't need to be saved). When iterate() is called to evaluate the variable, the Closure obtains items from the underlying sequence on demand. These items are then saved in memory, so that the next time the variable is used, the underlying expression is not re-evaluated. At one time Saxon tried to make an intelligent choice between saving the value of the expression in memory and re-evaluating it on each reference, but it's difficult to do this, especially without any real idea as to whether it's necessary to optimize on time or on space. So now each item in the sequence is evaluated once, the first time that item is needed, but once evaluated, it is kept, until such time as the Java garbage collector decides to get rid of it.

The one thing that does seriously break the Saxon pipeline is a temporary tree: this is because instructions that create trees work in push mode rather than pull mode. The events to write the tree pass down their own pipeline, which includes separate stages for tasks such as eliminating duplicate attributes and duplicate namespaces, schema validation, and then various stages of serialization. This pipeline is based on the SAX model (though Saxon uses a slightly different interface internally). When a variable is evaluated, and its value is a temporary tree, then the tree is fully constructed using this push model, and the root node is then returned to the "pull" pipeline.

In Saxon 6.x (which implements XSLT 1.0) the run-time architecture made heavy use of a technique called *expression reduction*. This involves partial evaluation of an expression, rewriting it as a new simplified expression as soon as the value of one of its subexpressions is known, and then re-optimizing the reduced expression at run-time. This technique was effective at ensuring that expressions used within a loop, but invariant over the loop, were only evaluated once; however, rewriting the expression at compile time is even better. Expression reduction at run-time has therefore disappeared from the product.

Before we leave the discussion of run-time optimizations, one other technique needs to be mentioned: tail call optimization. Perhaps with the availability of constructs such as `for` expressions and `<xsl:for-each-group>` in XSLT 2.0 and XPath 2.0, recursive templates and functions won't be used quite as frequently as they were in XSLT 1.0. Nevertheless, they will remain an important coding technique. Saxon is one of several XSLT processors that implements the standard tail-call optimization, whereby a recursive function or template call (in fact, any call whether recursive or not) that's the last thing done in the body of a template or function is performed after unwinding the stack, not before. This doesn't really improve performance much, but it does mean that you don't run out of stack space.

Other Optimization Approaches

In the previous sections I have outlined the main techniques used in the Saxon product to optimize XSLT and XPath (and indeed XQuery) execution. Now I'd like to take a very quick look at some of the things that Saxon doesn't do, but which other people have explored.

Firstly, there's a range of techniques that come under the heading of *parallelism*. Xalan, for example, has the parser and tree builder for the source document running in parallel with the transformation engine: if the stylesheet needs access to nodes that aren't there yet, the transformation engine waits until the parser has delivered them. The real saving here would come if it was also possible to discard parts of the tree once the stylesheet has finished with them. Unfortunately no-one seems close to solving this problem, even though many stylesheets do process the source document in a broadly sequential way.

There's been a considerable amount of academic work attempting to evaluate XPath expressions against a serial data stream. Olteanu, Mirth, Furche and Bry at the Ludwig Maximilians Universität München developed a system that rewrites any expression using reverse axes into an expression that uses forwards axes only. However, as far as I can tell it doesn't handle positional predicates: It's fairly obvious that expressions using positional predicates with a reverse axis, such as `preceding-`

`sibling::x[3]` are not going to succumb readily to this technique. At best, if you only get the chance to see the node once, you can remember a list of possible candidates, and discard nodes from the list when you have enough information to see that they don't match after all. Other researchers such as Berlea and Seidl at the University of Trier with their **fxt** processor have recognized that if you want a different processing model, it might be better to invent a different language. However, I wouldn't write off such ideas. Even if the best they can do is to choose when to process a tree serially and when to build it in memory, that will be a big step forward.

Another interesting approach is lazy tree construction, which has been explored by Schott and Noga at the University of Stuttgart. Their basic idea is to extend the lazy evaluation idea, which Saxon uses only for sequences, to trees, so that a node on a tree (and the subtree underneath it) can be represented as a closure containing an expression and a context for evaluating that expression. They provide some detailed measurements which show the result you might expect: if the user of the tree ends up reading all of it, the cost is much the same, but if the accesses to the tree are sparse, there is a big saving. This idea effectively extends the "pull" model into the tree construction part of the processing.

These approaches all concentrate on the dynamic architecture. There has also been academic work that attempts to achieve greater savings by compile time optimization. As with other academic work, it often selects an interesting subset of the language, leaving out not only the parts that are of no interest to the researchers, but often the parts that are the most intractable. This can make the papers frustrating to a practical engineer.

Much of this work focuses on type inferencing. Even with XSLT 1.0, where there is very little explicit type information available, it is possible to make type inferences and use these to generate more efficient code. Generally work on type inferencing has two objectives, improved performance and improved error detection; these are sometimes confused with each other. Global analysis of a stylesheet, especially in the context of a schema, can reveal which template rules are likely to call which others; this makes it possible to infer what the types of the arguments will be when a template is called, even though the types have not been declared.

In a database environment (that is, for XQuery) type inferencing is important because it helps you to identify which indexes you can use, which is the primary goal of all database query optimization. For XSLT it's less obvious where the potential benefits are to be found. Reducing the number of times you convert a string to an integer, or speeding up the decision whether to do integer or floating point arithmetic, are not very big wins (we're not yet at the stage where a 10% speed-up will improve your profits). The opportunity that's usually cited is the ability to prune the search space when evaluating a path expression such as `//a//b`. In principle, knowledge of the schema should enable you to decide which branches of the tree need to be explored. It's worth pointing out that this doesn't have to be a decision that is made statically: given the way XSLT uses template rules, it will very often be difficult to know statically what the schema type of the context node is going to be, but even if the information is only available at run-time (via the type annotation on the context node), it can still potentially be useful. However, there is an alternative to using the schema information, and that is to build indexes based on the actual document instance. Saxon already does this in the limited case of expressions such as `//a`: the first time such an expression is evaluated against a particular document, the result is stored with the document, and on subsequent occasions the document does not need to be searched again. It would be quite possible to extend this mechanism so that this scan builds an index for every element name that appears in a path expression preceded by "//", anywhere in the stylesheet. I don't know whether this would give better results than using schema information to prune the search, but I suspect that it might.

There are a few situations where static type information is invaluable, of course. A notorious example is the overloaded predicate syntax `A[B]`. If B is a boolean value, this acts as a filter, while if it is a number, it represents the expression `position()=B`. With an expression such as `X[2]` it is of course possible to avoid evaluating the full sequence X. But it's quite legal (if perverse) to write an expression such as `X[if (*) then 3 else .=0]` where the value of the predicate is sometimes numeric and sometimes boolean. It's only permissible to truncate the evaluation of the expression if you know that the value will always be numeric (and will always be the same number). This requires static analysis: not just type inferencing but dependency analysis too.

Summary

We've reviewed some of the techniques used by Saxon for optimizing XSLT and XPath execution, and we've looked at some of the other ideas floating around in other products and in the literature. There's no doubt that further advances are possible.

The big question for XSLT optimization, I think, is to find the right balance between static optimization techniques and dynamic, adaptive approaches. These aren't necessarily in conflict with each other, but they often take the wind out of each

others' sails, so in practice it's necessary for implementations to make a choice. Static analysis is always cheaper (at least for stylesheets that are performance-critical, which are the ones we care about). But decisions made at run time can always be made with the benefit of better information. As with all performance work, it comes down to a question of measurement: finding workloads that are typical of the problem space addressed by the particular product, and then measuring how well different approaches cope with them.

I have always advised people that measurement is the key to successful performance engineering. Sadly, I have to confess that I cannot produce numbers that demonstrate the performance contribution of each of the techniques I have discussed in this paper. I will have to leave that to the researchers.