XML Prague 2014

Conference Proceedings

University of Economics, Prague
Prague, Czech Republic

February 14–16, 2014

XML Prague 2014 – Conference Proceedings

ISBN 978-80-260-5712-3

Streaming in the Saxon XSLT Processor

Michael Kay
Saxonica
<mike@saxonica.com>

Abstract

Streaming is a major new feature of the XSLT 3.0 specification, currently a Last Call Working Draft. This paper discusses streaming as defined in the W3C specification, and as implemented in Saxon.[1] *Streaming refers to the ability to transform a document that is too big to fit in memory, which depends on transformation itself being in some sense linear, so that pieces of the output appear in the same order as the pieces of the input on which they depend. This constraint is reflected in the W3C specification by a set of streamability rules that determine statically whether a stylesheet is streamable or not.*

This paper gives a tutorial introduction to the streamability rules and they way they are implemented in Saxon. It then does on to describe the implementation architecture for implementing streaming in the Saxon run-time, by means of push pipelines, and gives rationale for this choice of architecture.

1. Introduction

Even seasoned readers of W3C specifications may find it bewildering to read, in the Last Call draft of XSLT 3.0[7], that "if exactly one operand O of a construct C is potentially consuming, and if the operand usage of O is absorption or inspection, then the posture of C is grounded and the sweep of C is consuming". Welcome to the language of streamability. This talk has two purposes: firstly to give an introduction to these concepts, and secondly to explain how they relate to the challenge of actually building a streaming implementation of XSLT.

Streaming is one of the main planks of XSLT 3.0 (the other is stylesheet modularity). Streaming is rather informally defined in the specification as "a manner of processing in which documents are not represented by a complete tree of nodes [in memory], but rather... as a sequence of events". The definition is deliberately fuzzy, to give maximum scope for implementors to innovate around it. Despite this, the specification gives a very precise definition of a subset of the language that is deemed to be "guaranteed streamable", which means that every processor that claims to

[1] References to Saxon in this paper refer to the current development snapshot, that is, to the state of the code base some time after the release of version 9.5 and some time before version 9.6. There may therefore be no public release of Saxon that corresponds in every respect to the description herein.

implement streaming at all must be capable of streaming this subset (which in turn means that when this subset is used, the processor is expected to be capable of handling indefinitely large source documents.)

Inevitably, in formulating the rules that define this subset, the working group had in mind ideas as to how streaming might be implemented in real processors, and the kind of constraints they might be operating under. Some of the constraints can be formalized, at least in principle: for example, to be streamable, there must be some kind of ordered correspondence between the events representing the source tree and the events representing the result tree. But the WG has not attempted to articulate the constraints in these terms; rather it has used intuitive reasoning to recognize that some functions and operators (such as `min()`, `max()`, and `sum()`) can be evaluated in a forwards pass through the document, and others (such as sorting, or `reverse()`) can not.

The *general streamability rules* that emerge are derived essentially from a process of abstracting these observations into a set of general rules. How do `max()` and `count()` differ, for example? The answer is that `count()` has no problems handling an input sequence that contains overlapping nodes, whereas `max()` cannot handle overlapping nodes without buffering. The streamability of a function like `count()` or `max()` thus depends on two factors: the nature of the supplied argument (does it contain streamed nodes, and if so, can they overlap?) and the way in which the function uses the items supplied as the argument. The first property is called "posture", the second (more intuitively) is called "usage". When the posture is striding, overlapping streamed nodes are allowed, when it is crawling, they are not. From this we get rules that say, for example, that (for an expression to be streamable), if the operand usage is inspection then the posture can be striding or crawling, but if the operand usage is absorption then the posture must be striding.

Streaming in Saxon[6] divides into two parts. The first part is static analysis to determine whether a construct is streamable and to devise the streamed execution plan. This follows the W3C analysis very closely, though Saxon implements some extensions, for example where it is able to take advantage of optimizations such as function and variable inlining. The second part is the actual streamed evaluation at run-time. Streamed execution is in principle possible using either a pull or push approach. The merits of the two approaches were described in [5]. To summarise the conclusions of that paper, the main advantages of a pull approach are (a) the ability to merge two streamed inputs (for constructs such as `deep-equal()`, the union operator, or the new `<xsl:merge>` instruction), and (b) easier coding, because most of the state of the processing can be kept on the programming language stack. By contrast, the advantage of push processing is that input events can be directed to more than one destination, which is essential for constructs such as `<xsl:fork>`. Saxon's streaming implementation is based largely on push processing, because although the implementation is more work, the architecture is more flexible. The

fact that significant components of Saxon have always used push processing (for example, the schema validator and the serializer) is another contributory factor.

The push pipelines used for streamed evaluation in Saxon are interesting because they include a mix of fine-grained events (startElement, endElement), and complete items (including complete trees). The paper will include some examples of how a few simple streamable expressions translate into such hybrid-granularity pipelines, and how the structure of these pipelines relates to the classifications established by the W3C streamability model.

2. Streamability

The static analysis performed by Saxon is modelled very closely on the rules in the W3C specification. The main concern in these rules is to show that

- the body of an `<xsl:stream>` instruction, and
- the body of an `<xsl:template>` whose mode is declared with `streamable="yes"`

are in fact streamable.

2.1. The W3C Streamability Rules

The rules (given in section 19 of the XSLT 3.0 specification) appear complex but once formulated, they are not in fact difficult to implement. Most of the apparent complexity is not in the logic of the rules, but in understanding the abstractions used in the rules, and understanding why the rules work.

The rules start with the idea of modelling a stylesheet (or at least, the parts of it that need to be analysed) as a tree of constructs. *Construct* is our first new technical term: it's a generalisation of an XPath expression, an XSLT instruction, and a few other things that are capable of being evaluated, like sequence constructors and patterns. The children of a construct in the construct tree are called its operands. The result of evaluating a construct is always a value. (Which sounds obvious, but we would have to modify this to handle FLWOR expressions in XQuery, which deliver not values but tuple streams.)

The sweep indicates how much of the input document is needed to evaluate the construct. The values are:

- *Motionless*: the construct either doesn't look at the input document at all, or it only needs to look at the place where the input document is currently positioned. Examples are `2+2`, `name()`, and `@status`. This relies on an assumption that as the input document is read, the system maintains a stack holding the names and attributes of the current node and all its ancestors, and the contents of this stack are always available without moving the input position.

- *Consuming*: the construct needs to read everything between the current start tag and the corresponding end tag. Examples are `string()`, `data()`, `number()`, `<xsl:value-of>`, `<xsl:copy-of>`.
- *Free-ranging*: the construct potentially needs to read outside the slice of the document represented by the current element and its ancestors. Examples are preceding-sibling::x, and xsl:number. Such constructs are never streamable. (But note, this doesn't prevent preceding-sibling::x or xsl:number appearing in a streamable stylesheet; the construct is free-ranging only if it operates on the streamed input document.)

The other property of a construct that affects streamability is a bit harder to visualize, and is referred to as the *posture* of the construct. Posture is concerned with determining whether an expression returns nodes from the streamed input document, and if so, where these nodes come from. There are five values:

- *Grounded*: this means that the expression doesn't return nodes from the streamed input. It either returns atomic values (or function items), or it returns nodes from non-streamed documents only.
- *Striding*: this means that the expression returns a set of nodes from the streamed input document, in document order, and that none of these nodes will contain another node in the result (none is an ancestor or descendant of another). A typical example is an axis expression using the child axis.
- *Crawling*: again, the expression returns a set of nodes from the streamed input document, in document order, but this time some of the nodes may be ancestors or descendants of others. A typical example is an axis expression using the descendant axis.
- *Climbing*: The spec assumes that when an input document is streamed, a stack of information is retained containing details of the names and attributes of all ancestor elements of the element at which the stream is currently positioned. Any expression that accesses ancestor nodes or their attributes from this stack has a posture of climbing. The key thing to remember about climbing expressions is that you can go upwards to ancestors of the current node, but you can't then navigate downwards again, because the children/descendants of these nodes are not retained in memory.
- *Roaming*: This indicates that an expression navigates off to parts of the document that aren't accessible when streaming, such as preceding or following siblings. This always makes the containing expression non-streamable.

Although some constructs have their own special rules, it's worth summarising and explaining the general streamability rules that apply to most instructions and expressions. The rules aim to determine the sweep and posture of a construct. The rules depend on identifying the operands (subexpressions) of a construct; for each operand you potentially need to know:

- its static type (this in fact is not often used)
- the sweep and posture of the operand (which you get by applying the rules recursively)
- the way in which the value of the operand is used, called the operand usage. This is one of the following:
  - *Absorption*: the parent expression makes use of information from the entire subtree rooted at nodes returned by the operand expression. Examples: `string()`, `data()`, `.//descendant::x`
  - *Inspection*: the parent expression makes use of properties of the nodes returned by the operand expression that can be established while positioned at a node's start tag. Examples: `name()`, `base-uri()`, `@status`, `../@status`.
  - *Transmission*: the parent expression returns nodes delivered by the operand expression. Examples: `A|B`, `tail(X)`, filter expressions.
  - *Navigation*: the parent expression performs arbitrary reordering of the returned nodes, or navigates away from them in arbitrary ways. Examples: `reverse()`, `<xsl:number>`.

The general streamability rules start by refining the sweep and usage of the operands by taking additional information into account. Specifically:

- If the type of the operand is a childless node kind, for example `text()`, then usage *absorption* is changed to *inspection*, because the entire subtree of such nodes is a simple property of the node and doesn't involve advancing the input stream.
- If the usage of the operand is *absorption* (for example if the parent expression atomizes the value of the operand), then the sweep of the operand may have to be increased. For example given the expression `contains(., "e")`, the sweep of the first operand is *consuming*, not because "." is intrinsically consuming, but rather because the `contains()` function performs atomization and this involves reading the whole subtree of the context node.

Once the properties of all the operands have been established in this way, the properties of the parent expression can be established:

- If there aren't any operands, the expression is *grounded* and *motionless*. This applies for example to simple literals like `"London"`, and also to the empty sequence `()`. It doesn't apply to axis expressions such as `child::*`, because axis expressions have special streamability rules. The general streamability rules described here are only the default.
- If any operand is non-streamable (technically, if it is *free-ranging* or *roaming*) then the parent expression is also non-streamable.
- If several operands are *consuming*, then in general the parent expression is not streamable (it is free-ranging and roaming). We'll discuss this important rule

below, There are exceptions for conditional expressions, where both branches can be consuming.

- If exactly one operand is *consuming*, then the parent expression will usually have the sweep of that operand. An exception is where the consuming operand is evaluated more than once (consider an expression such as `(1 to 5)!child::x`) in which case the result is not streamable. The posture of the result depends on the operand usage of this operand. If the usage is *transmission* (for example `X[@a = 3]`) then the posture of the result is the same as the posture of the operand. If the usage is *inspection* or *absorption* (for example `name()` or `data()`), then the posture of the result is *grounded*, because the result does not include any streamed nodes.

So there's a general rule that (with a few exceptions), no construct can have two operands that are both consuming. This rule is fairly easy to learn, and it's fairly easy for an implementation to give good diagnostics that explain when it's been violated. it's also fairly easy to understand why it should be true: you can only scan the input file once, and unless you're pretty smart, you can only evaluate one expression while doing so.

The exceptions are cases where the implementation is expected to be smart enough to evaluate both operands during a single pass:

- The `<xsl:fork>` instruction is explicitly introduced to request evaluation of two or more instructions during a single pass. One can imagine this being done by two parallel threads, but in fact it doesn't need true parallelism: as we'll see later in the paper, Saxon implements it simply by passing each parsing event to several expression evaluators in turn.
- Union expressions such as `a|b`, and map expressions such as `map{'a': price, 'b': discount}` can also have multiple consuming operands.
- A rather different case is conditional expressions (`<xsl:choose>`, or XPath `if-then-else`) where both branches can be consuming. This is a bit different because only one of the branches is actually evaluated.

If implementations have to be smart enough to evaluate `<xsl:fork>` and map constructors, then one might reasonably ask why we don't require them to evaluate multiple consuming operands wherever they occur, rather than treating these constructs as a special case. Perhaps some of the reason is pure caution; if there were no constraints at all, the number of parallel evaluations could run completely out of control. This reflects a recognition that forked evaluation has a cost, and indeed, that it's not really pure streaming, because although the input is streamed, the output has to be buffered so that the results of the separate construct evaluations can be assembled in the right order on completion. XSLT has a tradition of not leaving everything to the optimizer but allowing programmers to get involved in some of the key performance trade-offs, and this is an example of this philosophy.

The rules given above (the general streamability rules) apply to most kinds of expression, but they don't apply to the important case of path expressions and axis expressions. For axis expressions, the posture of the result depends on the posture of the context item and the choice of axis, using transition rules like the following:

- striding + child => striding
- striding + parent => climbing
- grounded + any => grounded
- climbing + child => roaming
- crawling + child => roaming

This last rule is one of the trickiest to get used to. The rule in its simplest form can be stated as "if you reached a node via the descendant axis, then you can't select downwards from it".

The reason for this rule is as follows. Suppose you select a sequence of nodes using the descendant axis. Then, in general, this sequence can contain two nodes where the first is an ancestor of the second. Suppose you want to process all the nodes in this sequence in turn. When you evaluate a consuming expression while positioned at the first node (the ancestor), this will move the position in the input stream to the end of that node, by which time you will have moved past the second node (the descendant), which is the next one you want to process.

The way that posture is used in determining the streamability of path expressions gives us another way of thinking about what posture actually means. Suppose that all the navigation in a template is reduced to a simple path, then that path has to match the regular expression `C*D?A*`, where `C` is a child step, `D` is a descendant step, and `A` is an ancestor or attribute step. It turns out that the rules for permitted posture transitions effectively define a finite state automaton that is equivalent to this regular expression; the posture values, with their fanciful names such as striding, crawling, and climbing can be seen as labels for the states in this automaton.

Note that the use of the descendant axis does not have to be explicit to fall foul of this rule. Operations such as taking the string value or typed value of a node, which are used all the time in XSLT programming, implicitly make a downward selection and are therefore not allowed on nodes that were reached via the descendant axis.

The rule disallowing multiple descendant steps is without doubt a great inconvenience. There are a number of workarounds:

- If you know, for example, that `<title>` elements will not be nested, then you can use the function `outermost(//title)` to select those titles that do not contain other titles. This expression, because it always selects nodes with disjoint subtrees, is deemed striding rather than crawling, and therefore allows further downward selection.

- If you only need a single node, you can write this in various ways: `head(//title)`, or `(//title)[1]`, or `zero-or-one(//title)`. Again these expressions cannot return nested nodes, so they are deemed striding rather than crawling.
- Similarly, text nodes are never nested, so the expression `//text()` is also striding.
- If several downward steps occur in a simple path expression such as `//section/title`, the specification says this is to be treated as equivalent to `//title[parent::section]` – that is, it is crawling rather than roaming. The reason here is that path expressions select nodes in document order, so it's always possible to evaluate the entire path in a single scan of the subtree under the current node, making it equivalent to a single use of the descendant axis.

The problem with this rule, as it appears in the W3C spec, is that it is very rigid. A great deal of the time, it prevents you writing constructs that you, with knowledge of the data, know will actually be streamable in practice even though they are not streamable in the worst case. Saxon therefore takes a more pragmatic view here (the spec permits this). Given a construct like the one above, Saxon will attempt an optimistic streamed implementation. If while processing one `<section>` element it encounters another nested `<section>` element, then it will process both of them in parallel during the same pass over the input. Any output produced from the inner, nested `<section>` will be buffered and emitted only when processing of the outer `<section>` is complete. So in the worst case, the process is not fully streamed, but it will still produce the right answer if enough memory is available for the buffered results. In effect, Saxon is doing an implicit `<xsl:fork>`: when it finds that a crawling expression produces two nodes where one contains the other, and there is then a further downward selection from these nodes, then it evaluates these two downward selections in parallel, buffers the results, and assembles the output in the correct order at the end. The beauty of this is that in the common case where elements are not in fact nested (as would typically be the case for `<xsl:value-of select="//title"/>`), no buffering is ever necessary, and the convenience of being able to write the expression in the natural way is delivered without any performance penalty and with no risk of running out of memory.

2.2. Visualising the Streamability Rules

Evaluating the streamability rules by hand for anything but trivial examples is challenging; the detail quickly becomes overwhelming, especially as the rules are highly recursive. This is of course a serious usability problem since stylesheet authors need to know whether they are writing streamable code or not.

With experience, the effect of the rules starts to become more predictable. One quickly develops an eye for coding patterns where the result of applying the rules is immediately obvious. Two of these patterns (expressions with multiple consuming

operands, and downward selection from a node reached using the descendant axis) have already been discussed.

However, for cases where the behaviour of the rules is less obvious, and for the benefit of users who have not yet formed the ability to predict the effect of the rules, Saxonica has developed a tool that allows the construct tree to be visualized, with all the properties of each construct that are relevant to streaming (sweep, posture, usage, type, context item posture, context item type) explicitly displayed.

The tool can be found at `http://dev.saxonica.com/stream`. At the time of writing it does not handle all the rules in the W3C specification, but it handles all the most frequently-encountered ones.

The tool is implemented using Saxon-CE[1] (XSLT logic running client-side in the browser).

2.3. Implementation of the Streamability Rules in Saxon

Saxon internally implements the streamability rules by means of a method `getSweepAndPosture()` on its `Expression` class (which corresponds to what the specification calls a *Construct*). The general streamability rules are defined on the class `Expression` itself, and constructs that have their own special rules override the method as required. The method takes a parameter to indicate whether the evaluation should proceed strictly according to W3C rules, or whether Saxon extensions are permitted. This allows the user to decide whether to take advantage of Saxon extensions or to prefer portability.

Implementation of the rules is not difficult. The expression class provides a method `operands()` which returns the operands of an expression together with their usage; it also provides static type information. So all the input to the W3C rules is readily available. Once the sweep and posture of an expression have been computed, the results are saved in the expression tree to avoid the costs of multiple computation.

Users don't only want to know whether an expression is streamable, they also want to know why not. So the method `getSweepAndPosture()` also takes an (output) parameter called reasons, which on return, if the expression is not streamable, contains messages explaining which rules were violated; these messages are used as the basis for compiler diagnostics.

Saxon performs the streamability analysis after all type-checking and optimization is complete. This creates the possibility that non-streamable code will be rewritten by the optimizer as streamable, or vice-versa.

The first case is not a problem, except for the rather stringent requirement in the W3C specification that an implementation should be capable of distinguishing stylesheets that are "guaranteed streamable" according to the spec, from those that rely on implementation extensions for their streamability. The only way to achieve that with Saxon is to switch optimization off.

Rewriting streamable code as non-streamable would be more of a problem for users. The problem is avoided by ensuring that the optimizer is aware of the need for streaming. In most cases this merely suppresses a rewrite that would otherwise take place, for example the use of indexing to support filter expressions such as `//emp[@id=$id]`.

In a few cases the optimizer deliberately tries to turn a non-streamable expression into one that is streamable. An example is the expression `for $x in //emp return ($x/@name, $x/@salary)`. This is not streamable as written because it is not permitted to bind a variable to a node in a streamed document. However, it can be rewritten as `//emp/(@name, @salary)`, which is indeed streamable.

On other occasions streamability is achieved as an unintended consequence of optimization. For example, the streamability rules don't allow streamed nodes to be bound to variables, passed as arguments to functions, or returned from functions (this is primarily to avoid the need for complex data-flow analysis). The Saxon optimizer will bypass this rule when it does variable and function inlining (replacing a variable reference or function call by the body of the variable or function). For example, a call to the function

```
<xsl:function name="inc">
  <xsl:param name="n"/>
  <xsl:sequence select="$n + 1"/>
</xsl:function>
```

is not streamable according to the W3C rules, simply because it fails to declare the type of its argument (and could therefore be processing a streamed node). After optimization, however, this function call will have been expanded inline, and the expanded code will satisfy all the streamability rules.

3. Run-time execution

While the W3C specification has a lot to say about how a stylesheet is analyzed to classify its constructs as streamable or not streamable, it says nothing at all about how to actually organize evaluation at run-time in a streaming manner.

The architecture of a typical XSLT 1.0 processor [3] is shown in Figure 1. The data flow is from left to right, but the control flow is more complex. In fact there are two control modules: the XML parser reads (pulls) data from a lexical XML input stream and writes (pushes) it to a tree in memory. The XSLT transformer, via its XPath engine, reads (pulls) data from this tree, and then writes (pushes) events down a pipeline which constructs events representing nodes in the result tree, which are in turn pushed to the serializer.

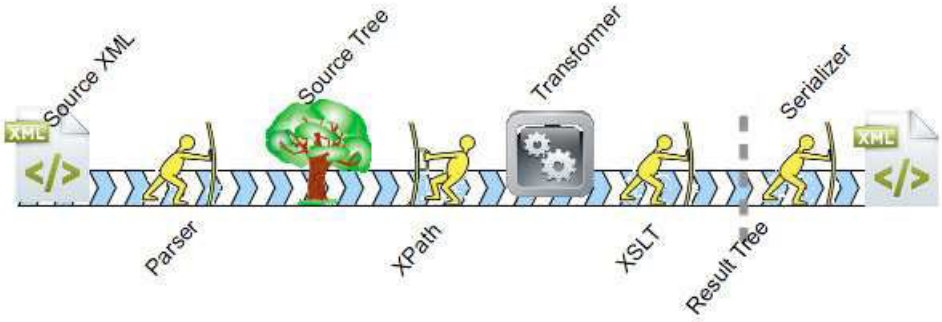

Figure 1. The architecture of a typical XSLT 1.0 processor

Note that the source tree is materialized in memory, but the result tree is not. Evaluation of XSLT instructions that construct nodes, and the serialization of those nodes, operate in a seamless push pipeline.

XSLT 1.0 famously does not allow a stylesheet to create temporary trees and then process them further using XPath; but in practice all 1.0 processors implement the EXSLT `node-set()` extension which circumvents this restriction. A typical XSLT 1.0 processor with the `node-set()` extension operates as shown in Figure 2:

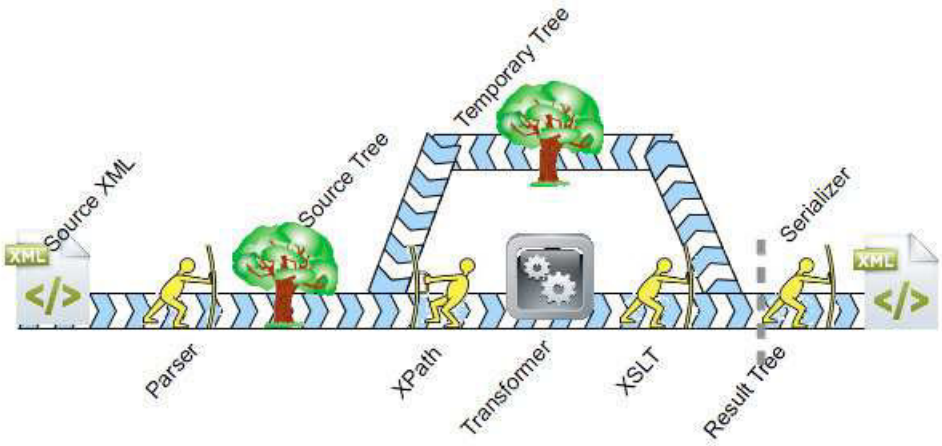

Figure 2. An XSLT 1.0 processor with the node-set() extension

Here variables containing temporary trees are materialized as trees in memory by XSLT instructions operating in push mode, and they are read by XPath expressions operating in pull mode.

XSLT 2.0 adds the possibility of schema validation, which can be applied to source trees, result trees, and also to temporary trees. The places where a schema validator can be invoked are shown with red tick-marks in Figure 3:

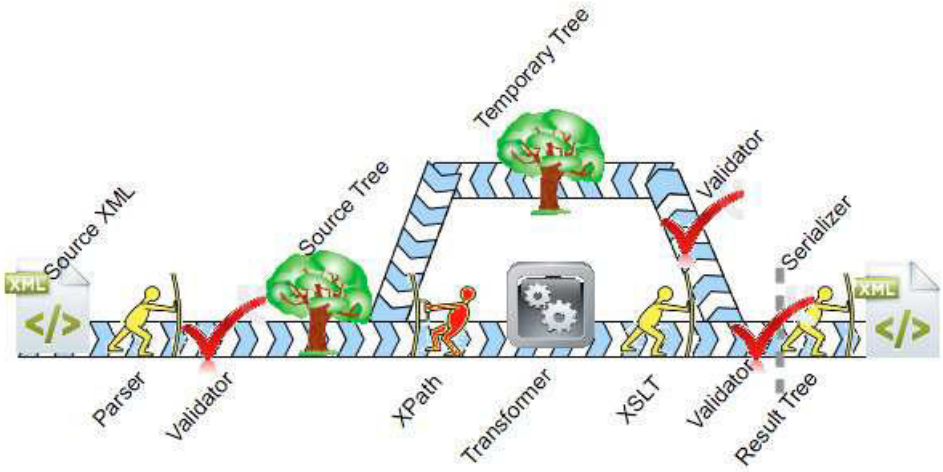

Figure 3. An XSLT 2.0 processor with schema validation

The XML Schema specification was designed to allow validation to be streamed: that is, one can determine schema validity over a stream of events representing the instance document, without needing to materialize the instance document as a tree in memory. Although one could envisage schema processors operating in either pull or push mode, in practice all the ones I know of work in push mode, and it can be seen in this diagram that this is rather convenient because in each case we have added the schema processor to a push pipeline.

In this architecture there are two kinds of pipeline: a push pipeline for the parsing and source validation, a pull pipeline for XPath evaluation, and another push pipeline for result tree consruction, result tree validation, and serialization.

A simple pipeline contains one control module which pulls data from the source end of the pipeline and pushes it to the result end of the pipeline. Data can flow naturally from a pull pipeline to a push pipeline, but the opposite is more difficult. There are essentially two ways to do it. One is to buffer the data into a reservoir from where another pipeline can read it; that needs memory. The other is to have to control modules that operate in some kind of synchrony so that data pushed by one is pulled by the other. This can be achieved by running the two control modules in separate threads under some kind of synchronization control, or with appropriate support from a programming language it can be achieved in a single thread by use of co-routines.

For more detail on these concepts, see [5]. The basic ideas are not at all new. Until the advent of large online disc storage, most data processing was done with magnetic tapes, and a major objective was to perform streamed processing of hierarchic data held in sequential form to transform it to another hierarchic data set also held in sequential form, with minimal use of tapes for holding intermediate data. Michael Jackson built many of the ideas of Jackson structured programming

[2] around these concepts, and the ideas are fully applicable to pipelines of XML transformations today.

Both pull and push pipelines can perform well, but a turbulent pipeline that has to switch between push and pull mode is likely to be less efficient. Some measurements demonstrating this effect can be found in [4].

To eliminate the need for interrupting the transformation pipeline with a reservoir that holds everything in memory, one can envisage a number of possible architectures:

- a single pipeline that operates in push mode from end to end.
- a single pipeline that operates in pull mode from end to end.
- a pipeline that has both push and pull sections, with the push-pull transitions being handled through multithreading (the co-routine alternative can probably be eliminated because of the paucity of modern programming languages that support the concept).

In Saxon's first forays into streaming, the third approach was adopted. This had the advantage that it was least disruptive to the existing architecture of the product; in particular, the XPath engine could continue to operate in pull mode.

In a pull mode XPath engine, evaluation of XPath expressions operates top-down. A parent expression controls the evaluation of its child expressions, typically asking child expressions to deliver their results as a stream of items which can be read as required. There is usually no need for the entire result of a child expression to be stored in memory, because each item can be processed as it becomes available. For example, the `sum()` function might be coded like this:

```
Iterator sum() {
  int total = 0;
  for item i in argument[0].evaluate() {
    total += i;
  };
  return monoIterator(total);
}
```

In general, every XPath construct is implemented by a function that consumes iterators representing the results of its subexpressions, and that itself delivers an iterator over its results. That is, each XPath construct is implemented by a component of a pull pipeline. This design approach is common in the implementation of functional programming languages; it can be seen as a combination of the Interpreter and Iterator design patterns in [Gamma et al].

In particular cases, it is possible to recognize XPath expressions where the tree does not need to be materialized. An example might be `sum(doc('employee.xml')//employee/salary)`. This could be implemented either by having the XPath engine make calls on an XML pull-mode parser as each salary element is required; or, as in the Saxon case, it could be implemented by having a push-mode XML parser

deposit a sequence of salary elements in a cyclic buffer, to be picked up by the XPath engine running in a separate thread. In Saxon this style of processing is implemented using the `saxon:stream()` extension function.

While this approach allows some useful applications to be written, it has many serious limitations. In particular, it does not allow for streaming applications that need to process the input data hierarchically. it's very hard to see how the XSLT 3.0 mechanism for streamable template rules, which perform a top-down hierarchic (but sequential) processing of the source tree, could be implemented using this kind of architecture.

Instead, Saxon is moving inexorably towards the first approach: an end-to-end push pipeline, illustrated by Figure 4.

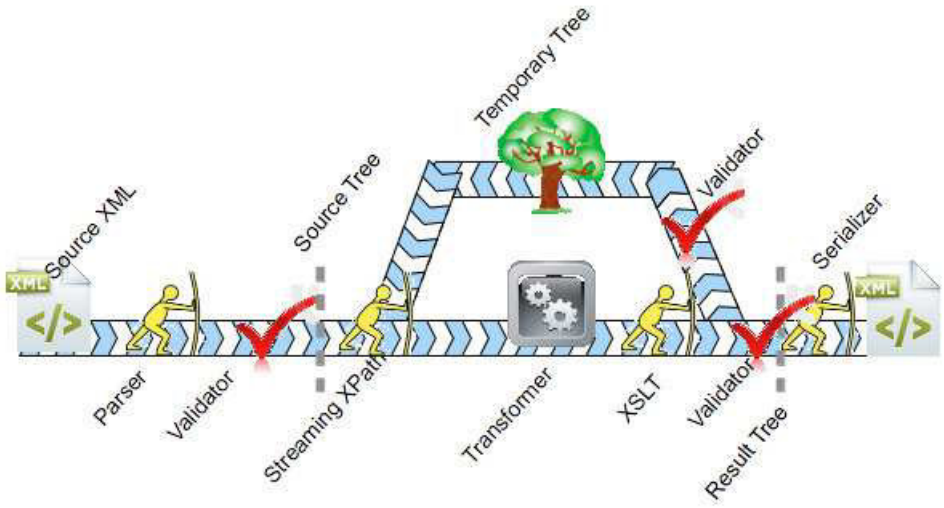

Figure 4. An XSLT 3.0 streamed processor using a pure push pipeline

Many components of Saxon have always been implemented as push pipelines, notably the XSLT instruction engine, node construction, serialization, and schema validation. The serializer alone contains around 30 components which are available to be assembled into a push pipeline based on the serialization options selected; the schema validator also has around 30 components each performing separate tasks. So the only part of the run-time engine which needs to be re-engineered for this streaming architecture is the XPath engine, where the existing pull-mode components need to be replaced by components that work in push mode. The next section of the paper explains how this works.

It's worth noting that where expressions operate on singleton values (for example, arithmetic expressions), the same code can be used in either a pull or push pipeline; the entire input value is available in materialized form, so pipelined evaluation becomes meaningless. By contrast, there are a few expressions (an example is the `insert-before()` function) that have more than one sequence-valued operand, and

where the roles of these operands are not symmetric. In a push pipeline multiple push-based implementations of such an expression are needed, depending on which operand is streamed (the rule that requires at most one operand to be consuming ensures that we can always choose one or the other, and of course we can select which one statically, because we know statically which operand is consuming).

3.1. Example of Push-based Expression Implementation

Perhaps some code would make these ideas more concrete. Here is an example that shows a simplified implementation of the `sum()` function in push mode. (it's simplified by only handling integers, by ignoring the second operand which gives a zero value, and by ignoring conditions such as overflow). Note that `IntegerValue` is a subclass of Item.

```
IntegerValue total;
void open() {
  total = 0;
  getResult().open();
}

void processItem(Item it) {
  total = total.add((IntegerValue)it);
}

void close() {
  getResult().processItem(total);
  getResult().close();
}
```

In push mode, evaluation is bottom-up, so these methods are called by whatever component it is that is evaluating the argument to the `sum()` function. That component is responsible for delivering a sequence of items; each one is delivered by calling `processItem()`, and the sequence is topped and tailed by calls of `open()` and `close()`. The component implementing `sum()` delivers a singleton sequence to the next component in the pipeline (available as `getResult()`), and this too is delivered using a sequence of three calls: `open()`, `processItem()`, and `close()`.

In this example, the things passed from one component to another are complete items. This is always the case for functions that operate on atomic values, which is the case for `sum()` and for a great many other expressions. In fact, more generally it is true whenever the operand is *grounded* or *climbing*. For functions that operate on streamed nodes, however (specifically, *striding* and *crawling* expressions), we don't always want to assemble the items before we can process them. Consider the expression `count(//*)`: the first item to be counted is the outermost element of the document, and we don't want to construct this as an object just so that it can be

counted. For this example, the pipeline needs to operate at a finer level of granularity: it needs to be notified of `startElement` and `endElement` events. The push code for `count()` looks like this:

```
IntegerValue count;
 void open() {
   count = 0;
   getResult().open();
 }

 void startElement(FleetingNode node) {
   count = count.add(1);
 }

 void processItem(Item it) {
   count = count.add(1);
 }

 void close() {
   getResult().processItem(count);
   getResult().close();
 }
```

This implementation can handle both complete items and fine-grained events. Often it will only have to handle one or the other: if the operand is striding (e.g. `child::X`) or crawling(e.g. `descendant::X`) then it will be notified of `startElement` events, while if it is grounded (e.g. `data(X)`) or climbing (e.g. `ancestor::A/@B`) then it will be notified of complete items. Nodes other than element or document nodes are also notified via the `processItem()` method, so with an expression such as `count(//node())` the two methods will both be called, for different kinds of nodes.

The class `FleetingNode` used in the `startElement` call is a representation of a node within a streamed document. It implements Saxon's `NodeInfo` interface, which is the standard way that nodes in trees are represented, but it only supports operations that are permitted when positioned at the start tag in a streamed document (classified in the spec as *inspection* operations). That is, you can call methods such as `name()`, `localName()`, and `baseUri()`; you can determine the type annotation; you can navigate the attribute, ancestor, and namespace axes; but you cannot make a downwards or sideways selection either explicitly by following axes such as `preceding-sibling`, `child`, or `descendant`, or implicitly by getting the string value or typed value. For the `count()` function, of course, we don't need to know any properties of the node, we just need to note its existence by incrementing the tally.

As well as constructs like `count()` that process fine-grained events representing element start and element end, there are also constructs that create new element nodes (for example, the `<xsl:element>` and `<xsl:copy>` instructions). Again, we don't want to materialize these nodes in memory; instead such constructs need to

generate startElement and endElement events which can then filter their way down the output pipeline, usually ending up in the serializer where they can be turned directly into start and end tags. Alternatively, if the streamed output is being captured in a variable, they might end up being passed to a tree builder that constructs a tree in memory, but this will only happen if the tree is actually needed.

3.2. Why not pull?

It would be possible to build a streaming processor that used a uniform pull model throughout, rather than Saxon's push model. Indeed, in some ways it would be easier.

As explained in [5] the primary benefit of a push model is that it allows events to be sent to more than one destination. This is used when implementing the expressions that explicitly allow more than one consuming operand, such as union expressions, `<xsl:fork>`, and map constructors, and it is also used for the Saxon extensions that permit downward selection from nodes reached using the descendant axis.

The other significant reason for using this model for Saxon streaming is that it is already used in significant parts of Saxon such as the serializer and schema validator, and this model therefore permits better re-use of existing code.

The main drawbacks of a push model are (a) that the coding required to implement it is probably more complex, and (b) that it is more difficult to handle constructs that merge two independent streamed inputs. There is only one such construct in XSLT 3.0, the `<xsl:merge>` instruction; Saxon has yet to implement streamed merging, but when it comes, it will probably use multiple threads.

4. Constructing the Push Pipeline

In the previous section we looked at how individual constructs in the expression tree are represented by push-evaluation components feeding events to each other in a bottom-up manner. This is bottom-up in the sense that subexpressions are processed before their parent expressions, and that the code for a subexpression calls the code for its parent expression to supply data, contrasted with top-down evaluation where the control flow is in the opposite direction. The relationship is very much the same as that between a top-down parser and a bottom-up parser, and the process of converting from one form to the other is known in Jackson Structured Programming [2] as *inversion*. Jackson showed that the process of inversion can be automated by a compiler.

It would be nice to think that we could invert Saxon's pull-based implementations of operations like `count()` and `sum()` to push-based implementations by an automated process; unfortunately doing so would require creating something akin to a new Java compiler, so instead we have done the process by hand. However, the assembly of these individual expression implementations into a working push

pipeline is of course fully automated. This process is essentially performing a Jackson inversion of the streamable XSLT templates in the stylesheet, and inversion at this level is greatly simplified because XSLT is a functional language free of side-effects. It is akin to the process of creating a bottom-up parser from a top-down BNF description, which is well-understood technology.

Consider first a simple template rule such as the following:

```
<xsl:template match="employee">
   <e><xsl:value-of select="name"/></e>
 </xsl:template>
```

Saxon will build an expression tree representing this template rule. The expression tree is a little more complex than might be imagined, because it contains nodes representing all the internal operations implied by the semantics of xsl:value-of: specifically a call to `data()` to atomize the `<name>` element; a call to `string-join()` to handle the case where there are multiple `<name>` elements.

Working top-down, the rule that a construct is only permitted one consuming child allows us to construct a path through this tree that contains all the consuming operations from bottom to top: this is called the streaming route, and the expressions on the streaming route provide the raw material for assembling the push pipeline that evaluates the template rule. Expressions on the streaming route may of course have other non-consuming operands; these are evaluated top-down in the usual way, at the point where their values are needed.

At the bottom of the streaming route there is always a pattern, which identifies which nodes the template is interested in (this is not the pattern that the template matches; it is a pattern that matches the nodes which the template reads from the streamed input). In this case the first operation in the push pipeline is to atomize `<name>` elements, so the relevant pattern is `name`. When the template is activated, it constructs a `Watch`; the `Watch` is a combination of the pattern, and the pipeline to be invoked when the `Watch` is matched. This `Watch` is registered with a `WatchManager`, which receives all events emanating from the XML parser, and tests each one against a list of registered `Watch`es, to see who needs to be notified. When the start tag for `<name>` is encountered, the `WatchManager` calls the `startElement()` method for the first component of this pipeline. This component is an atomizer, so it responds to this `startElement()` call by telling the `WatchManager` that it wants to know about all events up to the corresponding `endElement()` call. As these events arrive, it constructs the typed value of the element. The typed value of an element is in general a sequence of atomic values (though in the absence of a schema, this sequence will always be of length one). Each value in this sequence is notified to the next step in the push pipeline by a call on `processItem()`, so this next step (in our example, it represents the implicit `string-join()` operation) sees the sequence of atomic values comprising its streamed input. The `string-join()` operation in our example probably does nothing very interesting, because it's likely that employees only have one name; it

passes this name on to the next operation, which converts the string to a text node as required by the xsl:value-of instruction. The next operation after this in the streaming route is the literal result element that creates the <e> element. This emits events corresponding to the <e> start and end tags as part of its open() and close() calls, while the processItem() call that supplies the text node is passed on unchanged. So the template rule as a whole delivers a sequence of three calls (startElement, text node, endElement) and these become the result of the <xsl:apply-templates> instruction in the calling template rule, to be passed on up that template rule's pipeline.

The situation becomes a little more complex, of course, with template rules that involve loops and conditionals. The logic, however, is very similar to what would happen if all loops and conditionals were translated into apply-templates calls. Saxon doesn't quite go as far as doing that literally (it probably wouldn't be very efficient), but in terms of defining patterns and registering them with the WatchManager, it gives a good picture of what is going on.

As mentioned earlier, there are some constructs like union expressions, <xsl:fork> and <xsl:map> that explicitly allow multiple consuming operands. Saxon handles these by registering with the WatchManager one pattern (and corresponding pipeline) for each consuming operand. The pipelines operate in parallel as matching nodes are encountered, and the final step in each pipeline is to leave the result somewhere (in memory) where it is available to be assembled into the final result of the <xsl:fork> or <xsl:map> instruction.

5. Early exit, and error handling

A feature of functional languages like XPath is that it is not always necessary to evaluate the whole of an operand sequence in order to establish the result of the parent expression. For example, given the expression exists(child::author) it is not necessary to find all the author children; as soon as one is found, the expression can return true.

In a pull model, this is handled naturally by the parent expression iterating over the nodes returned by the operand expression, and simply not reading any further once it knows the answer.

This is less easy to achieve in a push (bottom-up) evaluation model, because the child expression knows nothing of its parent, so it will keep supplying new author elements until it reaches the end of the sequence of children. Of course, the parent expression can simply ignore them, but there are sometimes performance benefits to be gained by avoiding the unnecessary computation.

With streaming, particular gains are possible if the result to the entire transformation can be delivered before the whole input document has been parsed. This is more likely with some XPath or XQuery scenarios than with XSLT itself, but the same principles apply. For example, one can imagine a phase of a publishing pipeline

that is merely interested to read the value of the expression `/article/@version` — that is, an attribute of the outermost element of the document, which can be found very near the start of the file. Delivering this without parsing the rest of the file is potentially a big win.

To achieve this in a push pipeline, Saxon adds a parameter to the `open()` method for each each expression, whose value is a `Terminator` object. If the expression does not need any more input, it can call the `Terminator` object to say so. This enables expressions further up the pipeline to respond by themselves terminating; potentially the `WatchManager` itself is able to recognize that no more input is needed from the XML input stream, and it can then terminate the parse by throwing a (recognizable) exception, which is then caught so the user never knows about it. This does have the side-effect that XML well-formedness errors appearing subsequently in the file will never be detected (something that can be regarded as a bug or a feature).

A related issue is the implementation of the XSLT 3.0 try/catch facility. A conventional top-down implementation of try/catch can take advantage of the exception handling provided by the implementation language, which in Saxon's case is Java. With bottom-up evaluation, however, throwing a Java exception is no use, because the Java call stack is inverted so the exception will never reach the try/catch expression that is watching for it. Instead it is necessary to notify exceptions up the push pipeline in the same way as success results.

As it happens, the XSLT 3.0 try/catch capability is not streamable according to the specification; this is because it requires either output buffering or some kind of rollback capability to ensure that output produced before a failure that is caught does not make it into the result. Saxon however is more liberal here than the specification: it does allow try/catch with streamed input, and buffers the output in case an error occurs. This extension to the streamability rules can be justified on the basis that the output is sometimes much smaller than the input; indeed, in some cases its size is independent of the input, which would make it truly streamable even with buffering.

6. Conclusions

In this paper I have given a brief introduction to the concepts that are used in defining the streamability rules in the XSLT 3.0 specification, and have outlined some of the more important rules that determine whether constructs are or are not considered streamable; I have also explained some of the circumstances in which Saxon achieves streaming even in cases where this is not guaranteed by the specification. I then went on to outline how streaming is implemented in Saxon using an end-to-end push pipeline in which both whole items and fine-grained events can be notified in a bottom-up data flow from called expressions to calling expressions, and I explained some of the consequences of this architecture.

In 2001 [3] I wrote "Perhaps the biggest research challenge is to write an XSLT processor that can operate without building the source tree in memory. Many people would welcome such a development, but it certainly isn't an easy thing to do." I was right: it took a dozen years, but it has now been achieved.

References

[1] Delpratt, O'Neil, and Kay, Michael. Multi-user interaction using client-side XSLT Presented at XML Prague 2013. http://archive.xmlprague.cz/2013/files/xmlprague-2013-proceedings.pdf

[2] Jackson, Michael A. JSP in Perspective. (A retrospective look at Jackson Structured Programming, more accessible to a modern audience than the original publications from the 1970s). SD&M Pioneers' Conference, Bonn, 2001. http://mcs.open.ac.uk/mj665/JSPPers1.pdf

[3] Kay, Michael. Anatomy of an XSLT Processor. Published online by IBM DeveloperWorks https://www.ibm.com/developerworks/library/x-xslt2/

[4] Kay, Michael. Ten Reasons why Saxon XQuery is Fast. Bulletin of the IEEE Technical Committee on Data Engineering. http://sites.computer.org/debull/A08dec/saxonica.pdf

[5] Kay, Michael. You Pull, I'll Push: On the Polarity of Pipelines. Presented at Balisage: The Markup Conference 2009, Montréal, Canada, August 11 - 14, 2009. In Proceedings of Balisage: The Markup Conference 2009. Balisage Series on Markup Technologies, vol. 3 (2009). doi:10.4242/BalisageVol3.Kay01. http://www.balisage.net/Proceedings/vol3/html/Kay01/BalisageVol3-Kay01.html

[6] Saxonica: XSLT and XQuery Processing http://www.saxonica.com/

[7] XSL Transformations (XSLT) Version 3.0. W3C Last Call Working Draft, 12 December 2013. Ed. Michael Kay. http://www.w3.org/TR/xslt-30

Jiří Kosek (ed.)

XML Prague 2014
Conference Proceedings

Published by
Ing. Jiří Kosek
Filipka 326
463 23 Oldřichov v Hájích
Czech Republic

PDF was produced from DocBook XML sources
using XSL-FO and XEP.

1st edition

Prague 2014

ISBN 978-80-260-5712-3