Transcript of Michael Kay's Talk on XSLT Streaming at XML Prague 2014

The video is here: <http://www.youtube.com/watch?v=kAUPzeeW4Xg&t=186m18s>

Streaming is for handling large documents.

Streaming is the ability to process large input documents without first building a tree in memory.

## Benefits of streaming:

* can transform large documents
* reduced latency (delay): you start to see results before you've read the whole input
* resource reduction when handling many small documents
* can handle real-time data feeds: streaming can be used for data that is infinite, that never stops. Example: consider a satellite. If you wait until the satellite breaks down to report its results, that's not a good idea. You want to process the data as it's arriving and deliver real-time reporting of the feed.

Does streaming help when handling small documents? Streaming doesn't improve the performance of an individual transformation. Streaming doesn't make things faster. What it does is it reduces the resources needed. So if you are running a web service that is doing, say, 3 million transformations per second, then the reduction in memory resources may be useful to you. But streaming is not going to speed up the individual transformations.

## Design Approach for Streaming

* Define a subset of the existing XSLT language that is streamable and is statically detectable
* Define some new streamable constructs to handle as many streaming use cases as possible
* Allow streamed processing to coexist with unstreamed processing
  + Burst mode streaming
  + Lookup documents

The design approach taken for streaming was first to define a streamable subset of the XSLT language. It's not just a syntactic subset, it's a set of rules and constraints. If you are going to write streamable code, it has to satisfy the rules. However, if we just did that, it would restrict the developers too much; there would be too many use cases that couldn't be satisfied. So as well as restricting the language to say, "You have to follow these rules" we provided new constructs that are designed for streaming to actually make real practical use cases possible to write. These new constructs aren't confined to streaming, but they were particularly designed to make streaming possible.

We haven't just defined a streaming transformation language. What we've done is to allow streaming alongside non-streaming. Very often you will have a streamed document and other documents that aren't streamed. Most real-world document transformations involve multiple documents, many of them are quite small. So you can mix streaming and non-streaming together.

There is a mode of mixed streaming and non-streaming processing that is often very suitable which I call "*burst mode streaming*" and others call "*windowing streaming*". To illustrate: suppose you've got a very flat document (no deep nesting) such as an employee file with 100,000 employees in it. You've got 100,000 elements at level 2 in your tree. The burst mode streaming approach is to process each employee in turn: build a tree for each employee and process that tree in the normal (non-streaming) way, then move on to the next employee. You never have to build a tree for the entire file, so the memory requirements are bounded to just the amount of space needed to hold a single employee element. This sort of partial streaming is often very useful in practice.

## Streamability Rules

* Completely rewritten in the latest XSLT working draft
* Streamability of an expression depends on:
  + SWEEP: does it consume input?
  + POSTURE: how do the nodes in a node-set relate to each other?
  + USAGE: what use does the expression make of its supplied operands?
  + TYPE: what is the statically inferred type of the result?

*Streamability rules* are the set of restrictions on your stylesheet in order for it to be streamable. These rules have been rewritten several times; they are completely rewritten in the latest draft (12 December 2013), and they are complicated, sadly. The working group didn't go out of their way to make them complicated. There is a design compromise between making the rules simple and making the rules give a predictable result. The reason the rules are complicated is to ensure that things you think are streamable, actually are. So, hopefully, most of the time your intuition of what's streamable will be correct. But in order to make your intuition correct, the rules have to be quite complicated. That was the tradeoff.

There is lots of new terminology.

*Sweep* of an expression is how much of the input the expression reads. For example: suppose you call the name() function on an element. That expression doesn't need to move the input document, the XSLT processor can just look at the start tag. If you call the string() function on an element, that expression requires the XSLT processor read everything between the start tag and the end tag. So that's the concept of sweep: how much of the input does an expression require for an XSLT processor to read.

*Posture* is a more complicated concept. When you select a set of nodes, how do they relate to each other:

* Are you selecting a set of nodes that are all on the ancestor axis?
* Are you selecting a set of nodes that are peers or children on the same level and none of them contain another one (this is the striding posture)?
* Are you reading everything, crawling the tree, including nodes that are ancestors and descendants of other nodes?

*Usage* says how the result of an expression is being used by another expression. If the reverse() function or the sort() function is used, the input nodes are reordered. If xsl:number is used, the input nodes are navigated away from to somewhere else in the document. If the sum() function is used, the nodes are atomized. So there are different ways of using the nodes supplied as input to a function; they affect the streamability.

*Type* is the static type information (same as in XSLT 2.0).

# Understanding The Rules

* The rules are complex and very difficult to apply by hand
* Most cases come with some simple guidelines, e.g.,
  + Templates can only see descendants and ancestors of the context node
  + Only one downward selection allowed
  + Iteration over "striding" node-sets

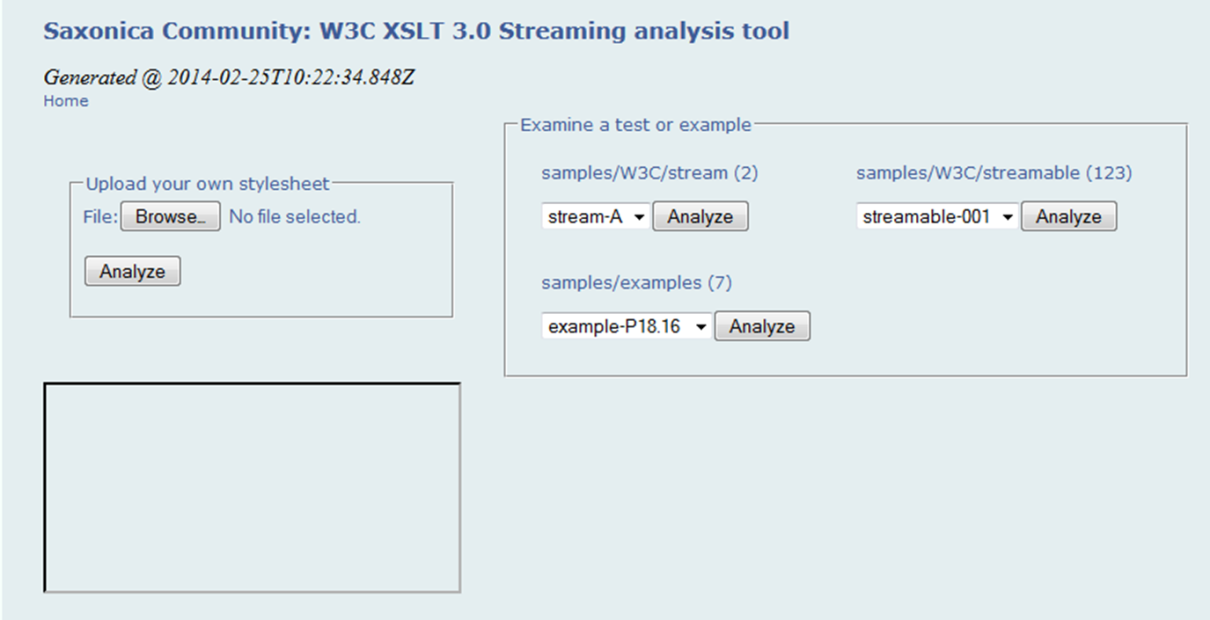
The streamability rules are complex and difficult to apply by hand. But the rules can be reduced to a set of simple guidelines.

# Saxonica Online Streamability Evaluation Tool

Saxonica has implemented a tool that implements the streamability rules and allows you to evaluate the streamability of a stylesheet. It was developed by John Lumley of Saxonica. The online tool is here:

<https://dev.saxonica.com/stream/analyze.html>

Here is a screen-shot:



Roger's Note: I clicked on the Browse button in "Upload your own stylesheet", selected one of my own stylesheets, and then clicked on the Analyze button. Michael mentioned the "W3C streamability test suite"; it appears to be here: <https://dvcs.w3.org/hg/xslt30-test/>

When the Analyze button is pressed, it kicks off an XSLT application written entirely in XSLT, running on Florent Georges' servlex environment (<http://servlex.net/>). The analysis is done on the server. An XML file is returned that contains the results of analyzing the XSLT with respect to the streamability rules. The results are displayed in the browser using Saxon-CE, running in the browser. The XSLT constructs that are streamable are highlighted in **green**, constructs that violate the streamability rules are highlighted in **red**. Below we see that there are no red constructs, so the XSLT program is streamable.



Roger's Note: I selected a different XSLT program which I know violates the streamability rules (it has two downward selections, which violates the "One Downward Selection Rule"). Here is the result:



On the left side of the display are boxes which can be selected. I selected the "final" box within "posture" and the display now annotates the XSLT with the postures:



This tool helps with understanding the streamability rules.

# The Streaming Design Challenge

* Retain the functional/declarative nature of XSLT
  + No side-effects, no mutable variables
* Allow you to "remember" what you have seen earlier in a single pass of the input

The streaming processor sees the input data only once. If the data is seen only once, then the XSLT program will need to remember what it's seen. How will the program remember what it's seen if it is not allowed to update variables? This required the introduction of some new constructs.

# Meeting the Challenge

* xsl:iterate
  + for-each with parameters passed to next iteration (fold over a sequence of items)
* accumulators
  + tree walking to compute properties of nodes (fold over a tree)
* maps
  + allow complex data structures
  + immutable

The xsl:iterate is an xsl:for-each with parameters: at the end of each iteration you can set some parameters that are then passed into the next iteration. So this is a way of remembering what was seen.

Whereas xsl:iterate will iterate over a sequence, accumulators iterate hierarchically over the whole input document: as you do a document-order tree walk of the input document, each time you hit a node and fire off template rules you change the value of accumulator functions. This is still a purely functional paradigm because the function you are computing is a function of the nodes in the document and it will return the same value every time you compute it.

Both xsl:iterate and accumulators require the ability to maintain state information. The XSLT working group realized from doing use cases that the state information could be quite complex. In prior versions of XSLT the standard way of retaining complex information has been with temporary trees. But with temporary trees it is very difficult and expensive to make a small change to a large temporary tree, so it is not a suitable structure for maintaining state information. So maps have been introduced into XSLT and XPath. If you write iterators and accumulators, you can use maps to retain temporary information. Maps are flexible and can be as complex as you like. Maps are immutable (unchangeable): when you change a value in a map, you get back a new map. That sounds expensive—it sounds like the whole structure is being copied—but this is not true; there are well-known implementations of maps where (internally) the changes are made simply by implementing a delta, thus it is quite efficient.