Transcript of Abel Braaksma's Talk on XSLT Streaming at XML Prague 2014

The video of Abel's talk is here: http://www.youtube.com/watch?v=kAUPzeeW4Xg&t=318m25s

Abel's slides are here: http://exselt.net/Portals/1/Streaming%20for%20the%20masses.zip

Streaming:

- Input data: tree size
- Output data: tree size
- Intrinsically streaming input/output
- Performance

Streaming is well-suited when:

- The input data is huge and doesn't fit into memory.
- The input is small but the output data is huge. For example, the output consists of all combinations of words in the input; the output is too large to fit into memory.
    - A streaming processor does not wait for the entire input to be finished processing, it produces the output incrementally.
- A twitter feed, a news feed, or a TCP/IP connection are intrinsically streaming: you get the data one element at a time, there is no way to get previous elements, and you want to process the data over days, months, or even years. In such scenarios there is only one option: streaming must be used.
- Streaming can improve performance: if the data is too large and the system starts doing lots of memory swapping, then streaming can help with performance.

Guaranteed Streamability

- Processor independent
- Guaranteed streamable
- Complex rules

The XSLT 3.0 specification says that if your stylesheet is guaranteed streamable, then regardless of what streaming processor you run the stylesheet on, the processor must process the stylesheet in a streaming manner. If your stylesheet is guaranteed streamable, you can run it on any streaming processor.

The rules for deciding if your stylesheet is guaranteed streamable are complex. The goal of this talk is to make the rules easy to understand.

Outline of Talk

- Introduction to streaming
- Ten rules for streaming in XSLT
- Complex scenarios, flowchart
- Conclusion

Below are 10 rules for writing streamable XSLT programs. These rules should cover most of what you would ever want to do.

New! xsl:mode is new in XSLT 3.0. It is used to switch on streaming. Once the mode is set to streamable, every template rule (with that mode) must be guaranteed streamable. Place the following instruction at the top of your XSLT file:

```
<xsl:mode streamable="yes" />
```

The following 10 rules assume that you have placed xsl:mode at the top of your XSLT file.

Rule #1 One downward select per template rule

To guarantee that a template rule is streamable, the template rule must not have more than one downward select.

The following template rule selects two child elements and therefore is not guaranteed streamable.

```
<xsl:template match="album">
   <xsl:value-of select="song" />
   <xsl:value-of select="writer" />
</xsl:template>
```

You may think:

> *But I know that in the input, the* song *element comes first*
> *and then the* writer *element second (in the* album *element).*

However, the XSLT processor doesn't know that. If the writer element should come first, then the processor would need to read the writer element and buffer it, read and write the song element, and then write the buffered writer element. If the writer element is large, then the processor would need to buffer a large amount of data, which defeats the purpose of streaming.

Thus, two downward selects are not allowed.

The following template rule is guaranteed streamable since it has only one downward select:

```
<xsl:template match="album">
   <xsl:value-of select="song" />
</xsl:template>
```

New! || is a new string concatenation operator in XSLT 3.0. For example, 'Singer: ' || song means that the string 'Singer: ' is to be concatenated with the value of the song child element.

The following is streamable because the template rule has just one downward selection:

```
<xsl:template match="album">
   <xsl:value-of select=" 'Singer: ' || song" />
</xsl:template>
```

'Singer: ' is just a string and is not selecting anything from the input.

Rule #2 One downward select per construct

A *construct* is an instruction (e.g., xsl:for-each), an XPath expression, part of an XPath expression (e.g., each step in an XPath path expression), and others.

What are the constructs in the following XSLT?

```
<xsl:template match="album">
   <xsl:for-each select="songs/song">
      <xsl:apply-templates select="songtext" />
   </xsl:for-each>
</xsl:template>
```

The content of xsl:template is an implicit xsl:sequence constructor. That implicit xsl:sequence constructor is a construct. Here the xsl:sequence constructor is shown explicitly:

construct

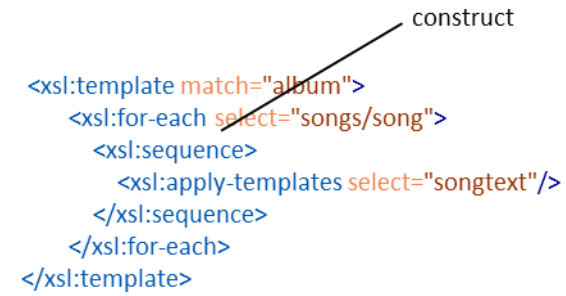
```
<xsl:template match="album">
    <xsl:sequence>
      <xsl:for-each select="songs/song">
        <xsl:apply-templates select="songtext" />
      </xsl:for-each>
    </xsl:sequence>
  </xsl:template>
```

xsl:for-each is a construct:

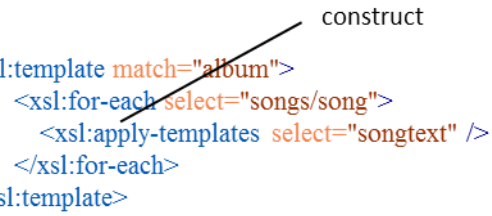
construct

```
<xsl:template match="album">
    <xsl:for-each select="songs/song">
      <xsl:apply-templates  select="songtext" />
    </xsl:for-each>
</xsl:template>
```

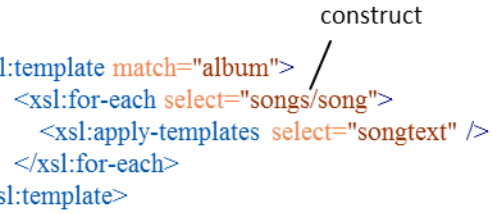
The content of xsl:for-each is an implicit xsl:sequence constructor. That implicit xsl:sequence constructor is a construct. Here the xsl:sequence constructor is shown explicitly:



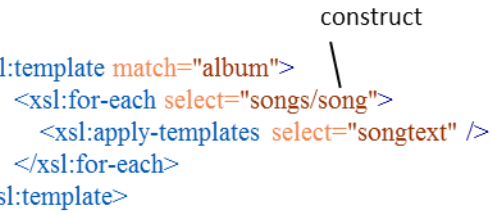
xsl:apply-templates is a construct:



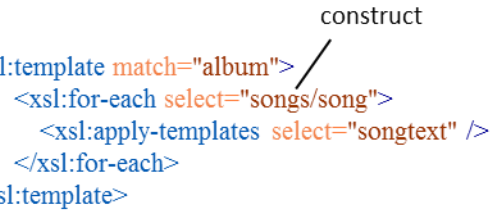
The XPath expression songs/song is a construct:



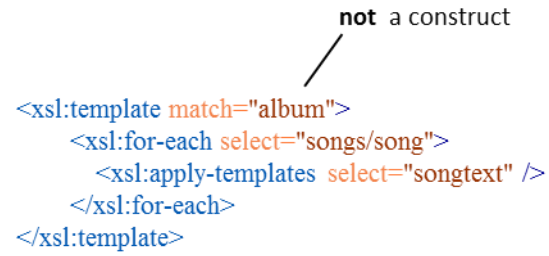
The song portion of the XPath expression is a construct:



The songs portion of the XPath expression is a construct:



xsl:template is not a construct and is not used in the analysis for streamability:



Within each construct there can be only one downward select.

Rule #3 Use motionless expressions

To the maximum extent possible, write your XPath expressions to be *motionless*.

As an XSLT processor streams through the input, it maintains a stack of ancestor information. That is, the processor maintains a stack of a node's ancestors (but not ancestor node's siblings or children). So, when a processor is positioned at a node it has this information:

- all the ancestors of the node
- all the namespaces that are in scope
- all the attributes of the node

All this information is immediately available and doesn't require the processor to move backward in the input. You can access as much of this information as you wish. An XPath expression that accesses this information is *motionless*.

An XPath expression that accesses an ancestor is called a *climbing expression*.

A *motionless expression* is one that doesn't require the processor to move forward or backward in the input. So any expression that accesses the ancestors of the current node, the in-scope namespaces, or the current node's attributes is a motionless expression. The following are examples of motionless expressions.

Access the name of the parent element:

```
parent::*/name()
```

Access two attributes on the current element

```
'Name: ' || @name || ', Age: ' || @age
```

Accessing attributes are not downward selects. Accessing attributes do not require any movement by the processor. Expressions to access the current node's attributes are motionless expressions.

Functions that don't operate on nodes are motionless so you can use them whenever and wherever you want. For example, calling the function to get the current date and time is motionless:

```
current-dateTime()
```

A processor can peek forward, without consuming input, to see if the current node has children, using this new function:

```
has-children(.)
```

The XPath doc() function returns an entire document. The processor does not process the returned document in a streaming fashion. Therefore the doc() function is motionless:

```
doc('Book.xml')
```

Now let's look at some illegal expressions. Consider this XML:

```
<author name="Tolkien">
  <books>
    <book>...</book>
    <book>...</book>
    <book>...</book>
  </books>
</author>
```

Suppose the processor is positioned at the <author> element. We now know that the processor can access @name. Consider this template:

```
  <xsl:template match="author">
    <xsl:apply-templates select="@name" />
    <xsl:copy-of select="books"/>
  </xsl:template>
```

The template invokes another template with @name and (in parallel) it tells the processor to copy the <books> element; the latter results in the processor moving forward in the input, to the </books> end tag. Now consider the template that processes @name:

```
  <xsl:template match="@*">
    <xsl:apply-templates select="ancestor::*" />
  </xsl:template>
```

It references the ancestors of @name. But wait! The processor has already moved to </books>, so accessing the ancestors of @name will require backing up, which is not allowed.

Notice in the first template it invoked another template, passing a *reference* to @name:

```
<xsl:apply-templates select="@name" />
```

You cannot pass node references.

Why? Because by the time the invoked template is executed the processor may have already moved away from @name (e.g., to the </books> end tag) and any processing of @name would then require backing up.

A template can be invoked with the name of the attribute. This is legal:

```
<xsl:template match="author">
   <xsl:apply-templates select="name(@name)" />
   <xsl:copy-of select="books"/>
</xsl:template>
```

Now it is not passing a reference to the attribute, it is merely passing a string.

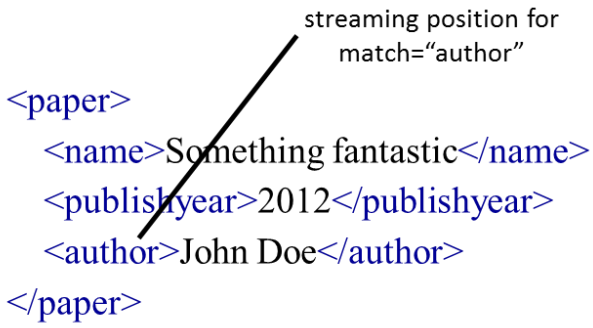
Rule #4 When moving up, you cannot move down again

Recall that it is okay to move up the tree to ancestor nodes. But you cannot then move down.

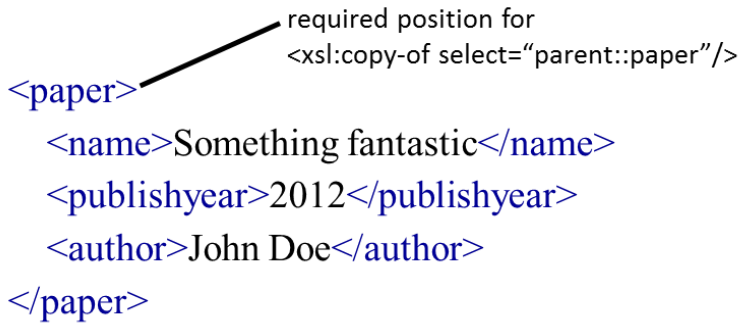
This template instructs the processor to move up to the parent element and then make a copy of it and everything within it:

```
<xsl:template match="author">
   <xsl:copy-of select="parent::paper"/>
</xsl:template>
```

Moving up to the parent element is okay. Making a copy of everything within it requires moving down and that's not allowed. Let's see why:



Making a copy of author's parent will involve moving up to <paper> and then down to <name>, <publishyear> and <author>:



Selecting <name> and <publishyear> requires backing up, which is not allowed.

Rule #5 No node references

Variables are not allowed to have node references.

The value of the variable, foo, is a node reference: foo has a reference to the author node:

```
<xsl:variable name="foo" select="author" />
```

That is not allowed because when the node (variable) is used by code somewhere else, the XSLT processor doesn't what is happening to it, the code may move backward or forward. While it might be possible to do some kind of static analysis, the rules become too complex, so the XSLT Working Group decided to simply *not allow node references*.

So these are not legal:

```
<xsl:variable name="age" select="@age" />
```

```
<xsl:param name="bar" select="current()" />
```

This is legal:

```
<xsl:variable name="foo" select="string(author)" />
```

Now foo just has a literal value, it does not have a reference to a node.

This is also legal:

```
<xsl:variable name="bar" select=" 'Name: ' || @name" />
```

The value of bar will be a literal value: the concatenation of 'Name: ' and the value of @name.

Interestingly, an xsl:apply-templates that is inside a variable:

```
<xsl:variable name="authors">
  <xsl:apply-templates select="author" />
</xsl:variable>
```

is guaranteed to return a *copy* of the nodes, not a *reference* to the nodes, so that is legal.

So you are allowed to use variables, just not variables that have node references.

Rule #6 Be a chameleon, hide!

You can hide a portion of the input from streamability rules, thus allowing your XSLT program to do anything it wants with that portion. This is accomplished using copy-of(). Using copy-of() you are telling the processor, "I know that this portion is not too big, it fits into memory, it's all right."

This violates the one downward selection rule:

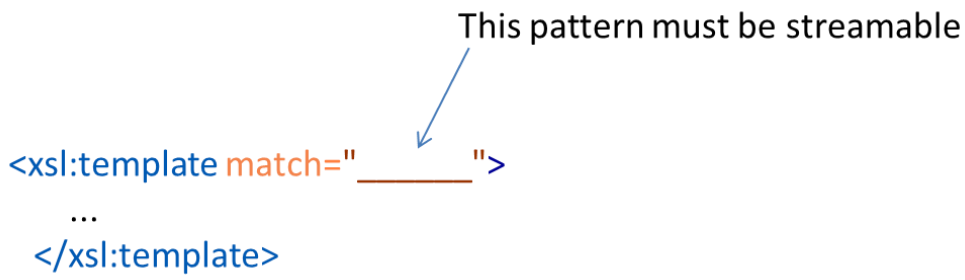
```
<xsl:template match="book">
   <xsl:variable name="book" select="."/>
   <xsl:value-of select="$book/title" />
   <xsl:value-of select="$book/author" />
</xsl:template>
```

By copying book we can perform any operations we desire:

```
<xsl:template match="book">
   <xsl:variable name="book" select="copy-of(.)"/>
   <xsl:value-of select="$book/title" />
   <xsl:value-of select="$book/author" />
</xsl:template>
```

Rule #7 Streamable Patterns

The patterns that you use in a template match must be streamable.



The following match pattern is not allowed. It is making two downward selects: selecting album and then selecting song.

```
<xsl:template match="album[song]">
```

Rule: you cannot have another downward select inside a predicate.

You cannot have a numeric predicate:

```
<xsl:template match="album[3]">
```

You cannot have a predicate involving a numeric expression:

```
<xsl:template match="album[position() > 3]">
```

You cannot have a predicate that accesses preceding nodes:

```
<xsl:template match="album[preceding::album/@name = 'it']">
```

You cannot use keys:

```
<xsl:template match="key('song-name')/album">
```

You cannot have a match pattern that starts with a variable reference:

```
<xsl:template match="$test/help">
```

The following match patterns are okay.

You can have a motionless expression in a predicate:

```
<xsl:template match="album[@song]">
```

You can have a predicate that walks the ancestor axis. As we discussed earlier, the ancestor axis is motionless. Example:

```
<xsl:template match="album[ancestor::*[3]/@name = 'michael']">
```

You can use the doc() function in the predicate:

```
<xsl:template match="album[@singer = doc('cfg.xml')/setting/name]">
```

You can have a path expression:

```
<xsl:template match="album/song/name">
```

A text node cannot have any children, so referencing a text node is motionless. You can search a text node using a pattern predicate:

```
<xsl:template match="para/text()[matches(., '\d\d\d\d')]">
```

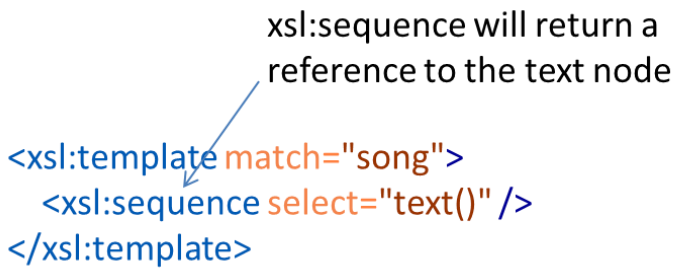
You can walk the ancestor axis and get the name of the parent node:

```
<xsl:template match="@age[name(parent::*)]">
```

Rule #8 Ground all templates

Templates must be *grounded*. Grounded means the template does not return references to nodes: the result of an xsl:apply-templates must not be a reference to nodes.

This is not allowed:



This also is not allowed since xsl:sequence returns a reference to the attribute:

```
<xsl:template match="song">
   <xsl:sequence select="@singer" />
</xsl:template>
```

xsl:sequence creates a reference to nodes. References do not "sit" on the ground (i.e., are not grounded).

It is allowed for a template rule to return a copy of nodes:

```
<xsl:template match="song">
   <xsl:copy-of select="text()" />
</xsl:template>
```

It is allowed for a template rule to return a string, using xsl:value-of:

```
<xsl:template match="song">
   <xsl:value-of select="@singer" />
</xsl:template>
```

"Grounded" means you don't return a reference to a streamed node.

Rule #9 Use motionless filters

Any expression that is placed in a predicate must be *motionless*.

The predicate in this expression is motionless:

```
*[self::para or self::p]
```

The attribute axis is motionless:

```
price[@currency[starts-with(., 'EUR')]]
```

The instance of operation asks for a property of a node, which is motionless:

```
node()[. instance of element()]
```

The ancestor axis is motionless:

```
cfg[ancestor::root/@version = $glob-version]
```

Multiple ancestors is motionless:

```
*[ancestor::foo >> ancestor::bar]
```

A text node is motionless:

```
buildno/text()[. = $builds/buildno[last()]]
```

Rule #10 Master xsl:fork

Recall the first example, where we had two downward select expressions:

```
<xsl:template match="album">
  <xsl:value-of select="song" />
  <xsl:value-of select="writer" />
</xsl:template>
```

xsl:fork helps us deal with that.

xsl:fork tells the XSLT processor: "From this point on I want you to make multiple starting points in the input stream (as in multi-threading) and process each of them in parallel."

The following shows how to use xsl:fork

```
<xsl:template match="author">
  <xsl:fork>
    <xsl:sequence>
      <xsl:value-of select="song" />
    </xsl:sequence>
    <xsl:sequence>
      <xsl:value-of select="writer" />
    </xsl:sequence>
  </xsl:fork>
</xsl:template>
```

xsl:fork may only have xsl:sequence as its child. Each xsl:sequence must return a grounded expression, which is why xsl:value-of is used (xsl:value-of returns a string, which is grounded).

So, this template is not legal:

```
<xsl:template match="album">
  <xsl:value-of select="song" />
  <xsl:value-of select="writer" />
</xsl:template>
```

But this template is legal (streamable):

```
<xsl:template match="author">
  <xsl:fork>
    <xsl:sequence>
      <xsl:value-of select="song" />
```

```
        </xsl:sequence>
        <xsl:sequence>
          <xsl:value-of select="writer" />
        </xsl:sequence>
      </xsl:fork>
    </xsl:template>
```

Sorting is not possible in a streaming program. You can, however, kind-of do a sort as shown here:

```
    <xsl:template match="author">
      <xsl:fork>
        <xsl:sequence>
          <xsl:copy-of select="entry[@type eq 'error']" />
        </xsl:sequence>
        <xsl:sequence>
          <xsl:copy-of select=" entry[@type eq 'warning']" />
        </xsl:sequence>
      </xsl:fork>
    </xsl:template>
```