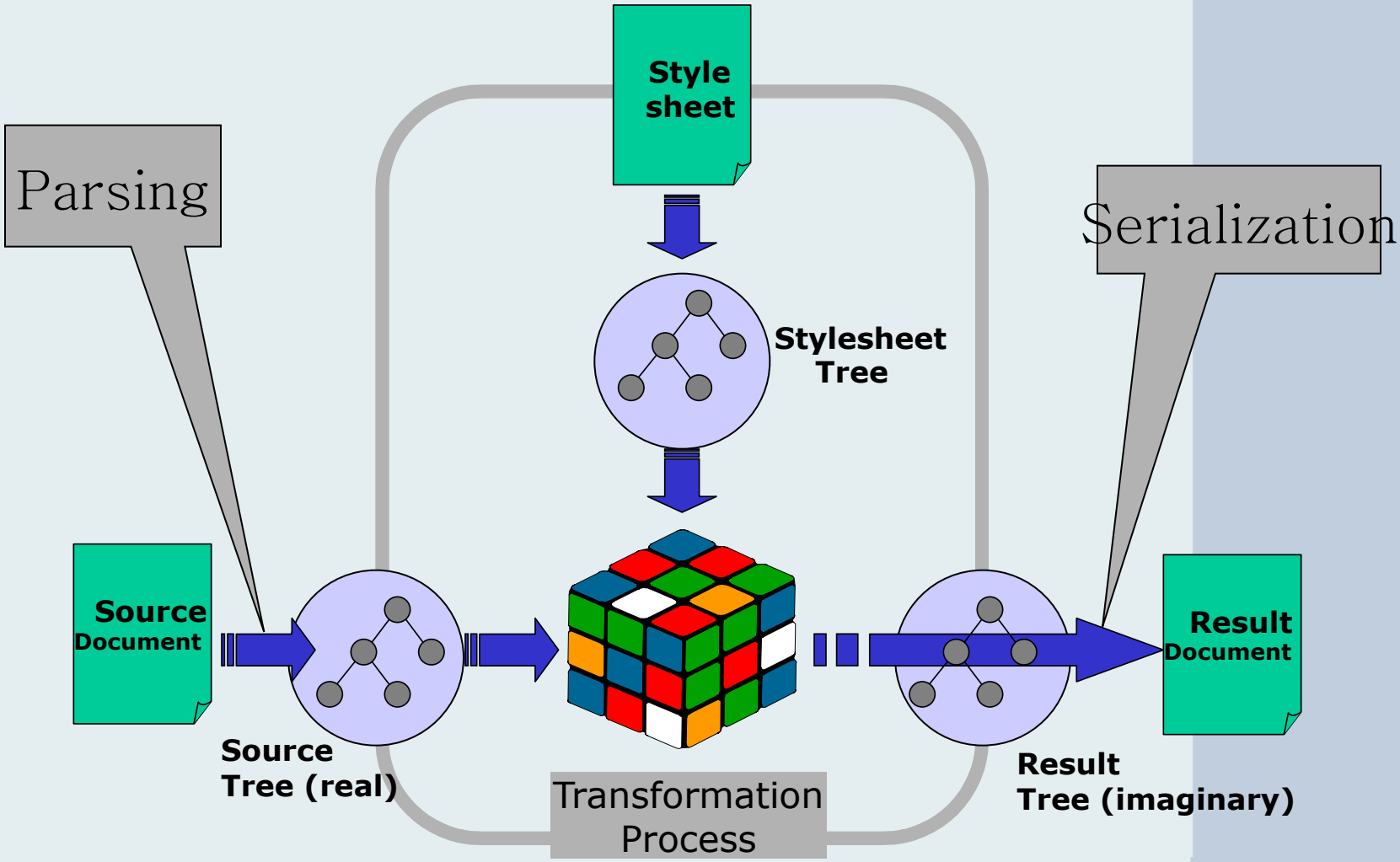
SAXONICA.COM
XSLT AND XQUERY PROCESSING

Streaming in XSLT 2.1

Michael H. Kay

Traditional XSLT processing model



The Goal

- Where possible, avoid building the source tree in memory
- Allow processing of indefinitely large source documents in constant memory
- Reduce latency: start producing output before all the input is available

Previous attempts at streaming transformation

- Alternative languages to XSLT, for example STX, XStream
- Research on XPath streaming
  - often a subset (e.g. no predicates)
  - generally single XPath expressions only
- Research on XSLT streaming
  - always a subset, usually rather small
- Press releases
  - e.g. Datapower, Intel
  - no technical information available
- Ad-hoc XSLT language extensions
  - Saxon since 8.9

The XSL WG approach

- Driven by use cases
  - describe transformations that “ought” to be streamable
  - (order of output events corresponds to order of input events)
- Build on existing XSLT language
  - new compatible language features
  - retain foundations as a functional language
  - mix streaming and non-streaming code
- Don’t rely too heavily on optimizers
  - exploit the fact that users know their data
  - users should say when they want streaming
  - this constrains them to follow certain rules
  - translating from XSLT code to a streamed execution strategy should involve no magic

Some streaming design patterns

- Aggregation
  - avg(//employee/salary)
- Event stream isomorphism
  - rename all elements
  - delete selected elements
  - order-retaining grouping
- Accumulation
  - add current balance to every transaction
- Burst-mode or windowing
  - output contains N employee records, each created by transforming the corresponding input record

Aggregation example

```
<xsl:stream href="employees.xml">
   <xsl:value-of select="avg(*/employee/salary)"/>
</xsl:stream>
```

- Body of <xsl:stream> is analyzed for streamability
- Analyzes navigation paths starting from the context node
- Allows only one downward selection path

Windowing

- To get around the restriction of one downward selection, copy subtrees when necessary

```
<xsl:stream href="employees.xml">
  <xsl:value-of
    select="avg(*/employee/copy-of()/(salary + bonus))"/>
</xsl:stream>
```

- The path analysis recognizes that child::salary and child::bonus are not on navigation paths from the original context node

Streaming Templates

- Example: renaming and selective deletion

```
<xsl:template match="*">
   <xsl:element name="{lower-case(name())}">
      <xsl:apply-templates/>
   </xsl:element>
</xsl:template>

<xsl:template match="note"/>
```

- Should be streamable because output events are isomorphic with input events

Streaming modes

- Declare streamability as a property of a mode

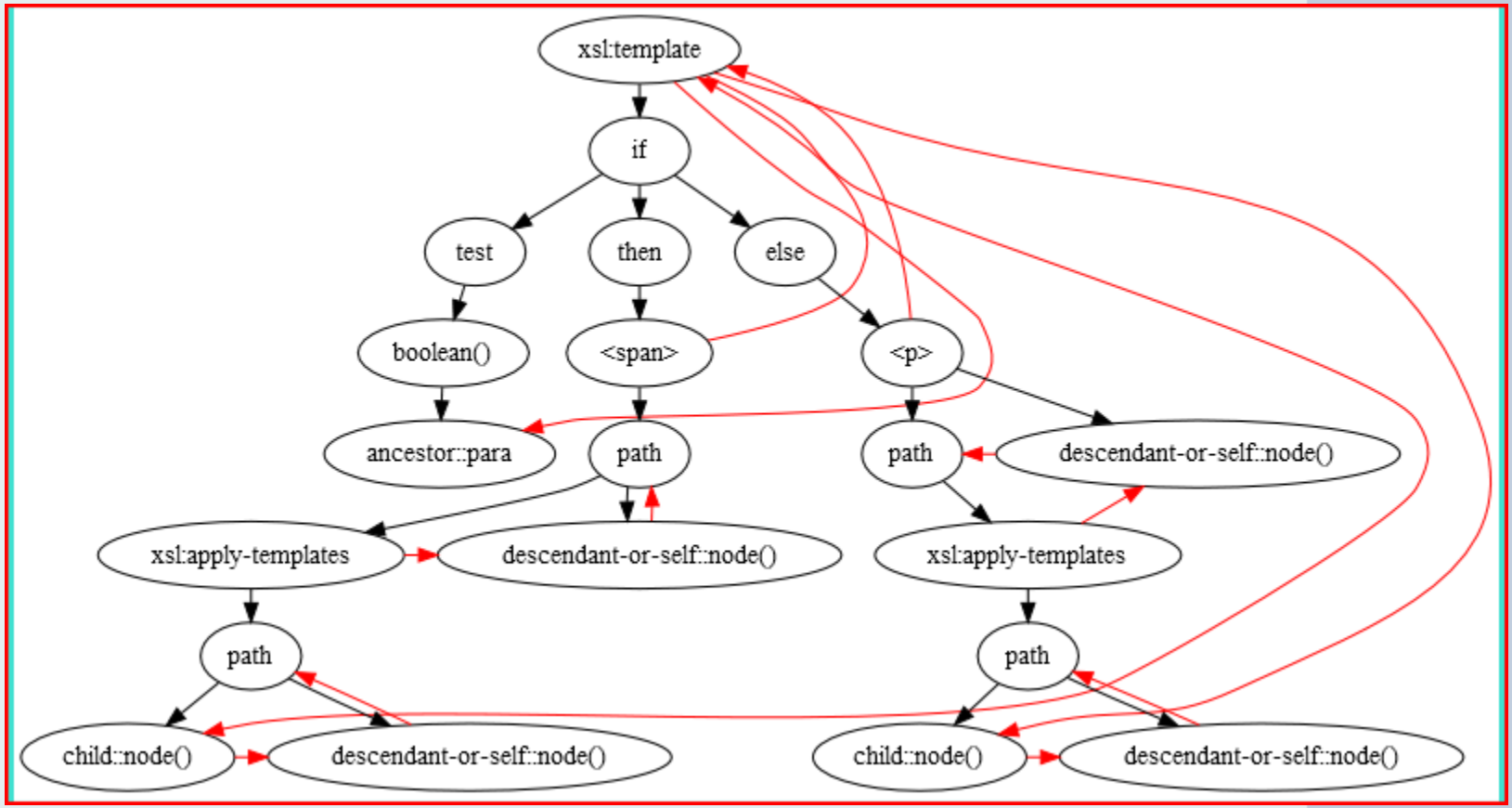
```
<xsl:mode name="M" streamable="yes"/>
```

- Apply path analysis to every template rule in the mode (independently)
- If all the templates in the mode are streamable, then streamed execution is feasible

Streamability rules in more detail

- Trace all navigation paths from the context node
  - including calls to functions and named templates
- The construct is streamable if:
  - no path has a sideways step (e.g. preceding/following)
  - no path goes up and then down (../xxxx)
  - no path goes to descendants and then down (//xxx/yyy)
  - no two paths both go down unless they are in mutually exclusive branches of a conditional
  - no path jumps into a looping construct such as xsl:for-each
  - no path contains a reordering construct such as xsl:sort
  - no path cycles back to the original construct (i.e. the construct doesn't return a streamed node as its result)

Expression trees and data flow graphs



What data is available while streaming?

- Immediate properties of the context node
  - name, node-kind, type annotation, base URI
- Attributes of the context element
- Ancestors of the context node and their attributes
- For each ancestor:
  - a count of preceding siblings, broken down by (name, node-kind, type)
  - sufficient to evaluate match="p[N]" and simple cases of xsl:number

xsl:iterate

- Allows applications to “remember what they have seen” while processing a stream of nodes
- Similar to xsl:for-each, but:
  - explicitly sequential
  - allows breaking out of the loop
  - parameters can be set during one iteration for use during the next iteration
- Syntactic sugar for a head-tail recursion (or a fold-left higher-order function)

xsl:iterate example

```
<account>
  <xsl:stream href="transactions.xml">
    <xsl:iterate select="transactions/transaction">
      <xsl:param name="balance" select="0.00" as="xs:decimal"/>
      <xsl:variable name="newBalance"
                    select="$balance + xs:decimal(@value)"/>
      <balance date="{@date}" value="{$newBalance}"/>
      <xsl:next-iteration>
         <xsl:with-param name="balance" select="$newBalance"/>
      </xsl:next-iteration>
    </xsl:iterate>
  </xsl:stream>
</account>
```

Benefits of xsl:iterate

- For the user:
  - many users find implementing such logic using head-tail recursion is HARD
  - many users write $O(n^2)$ code because it's simpler than doing recursion
- For the system:
  - easier to implement without blowing the stack
  - easier to implement with a streaming pass over the input sequence

Merging and splitting

- Example use cases:
  - Merge log files from several web servers, all of which are sorted by date/time
  - Given a file of employee data, create one output file containing the employees in each location
- Also need to calculate multiple values during a single pass of the input
  - Get the five highest and lowest paid employees
  - Remove all comments and report how many there were

xsl:merge

- Multiple input files
  - homogenous or heterogenous
  - each pre-sorted using compatible sort keys
  - system checks that the files are correctly sorted (fatal error if not)
- Defines action taken with each group of items having common merge keys
- Works with both streamed and unstreamed inputs
  - but optimized for streaming

xsl:merge example
Merging a collection of log files

```
<xsl:merge>
  <xsl:merge-source select="uri-collection('log-collection')">
    <xsl:merge-input>
      <xsl:stream href="{.}">
        <xsl:copy-of select="events/event"/>
      </xsl:stream>
      <xsl:merge-key select="@timestamp" order="ascending"/>
    </xsl:merge-input>
  </xsl:merge-source>
  <xsl:merge-action>
      <xsl:sequence select="current-group()"/>
  </xsl:merge-action>
</xsl:merge>
```

xsl:fork

- Allows two or more “downward” expressions on the same input stream
- Evaluates multiple results for the same input in a single pass
- Implementation may use multiple threads but this is not required

xsl:fork example
Multiple output files

```
<xsl:stream href="employees.xml">
  <xsl:fork>
    <xsl:result-document href="male.xml">
      <xsl:copy-of select="*/employee[@gender='male']"
    </xsl:result-document>
    <xsl:result-document href="female.xml">
      <xsl:copy-of select="*/employee[@gender='female']"
    </xsl:result-document>
  </xsl:fork>
</xsl:stream>
```

xsl:fork example
Streamed input, buffered output

```
<xsl:stream href="employees.xml">
  <xsl:fork>
    <minSalary value="min(*/employee/salary)"/>
    <maxSalary value="max(*/employee/salary)"/>
  </xsl:fork>
</xsl:stream>
```

Implementation in Saxon

- Saxon 8.9
  - burst-mode streaming using saxon:read-once
- Saxon 9.1
  - burst mode streaming
    - new syntax saxon:stream()
    - more flexible path expressions
- Saxon 9.2
  - adds streaming templates
    - strict rules about what's allowed
  - non-streaming xsl:iterate implementation
- Saxon 9.+
  - much more!
  - hope to describe at Balisage 2010