Projection and Streaming: Compared, Contrasted, and Synthesized

Michael Kay
Saxonica
<mike@saxonica.com>

Abstract

This paper describes, compares, and contrasts two techniques designed to enable an XML document to be processed without building an entire tree representation of the document in memory. Document projection analyses a query to determine which parts of the document are relevant to the query, and discards everything else during source document parsing. Streaming attempts to execute a stylesheet "on the fly" while the source document is being read.

For both techniques, the paper describes the way that they are implemented in the Saxon XSLT and XQuery engine.

Performance results are given that apply to both techniques, in relation to the queries in the XMark benchmark applied to a 118Mb source document.

The paper concludes with a discussion of ideas for combining the benefits of both techniques and getting more synergy between them.

1. Overview

Document projection is a technique introduced by [1] designed to address the problem that some XML documents are too large to fit as a tree in main memory and therefore cannot be queried by XQuery processors that build a complete tree in memory.[1]The idea is to perform static analysis on the query to determine which parts of the source document are actually needed by the query, and then to build a tree in memory that omits the parts of the document that the query never visits.

Streaming, as we use the term in this paper, is a mechanism defined in the XSLT 3.0 specification [8] that allows a subset of the XSLT language to be processed in streaming mode. It attempts to identify those constructs in the language that can be processed in "constant memory", that is, whose memory requirement

[1]Memories were smaller in those days: Marian and Siméon report a maximum document size of 50Mb for Saxon, and as little as 7Mb for QuiP. But the fact that the limits are higher today does not mean the problem has disappeared. Indeed we are now starting to see the 2Gb boundary imposed by 32-bit addressing becoming a potential problem.

is independent of document size. Provided that the stylesheet conforms to these static constraints on the use of language constructs, a streaming XSLT processor is then expected to be able to handle documents of arbitrary size (with a few caveats, which are well documented).

So projection and streaming are addressing essentially the same problem, and there are similarities (but also significant differences) in the way they tackle it. Both mechanisms are implemented in Saxon [6], essentially independently of each other. Part of the motivation for this paper is to examine whether there are synergies to be obtained by integrating the two mechanisms more closely, for example by using common algorithms for the static analysis, or common data structures for the result of the static analysis, or by using ideas from one of the subsystems to improve the effectiveness of the other.

The paper starts with an overview of the two techniques of projection and streaming, in each case presenting first the published specifications, and then details of the implementation in Saxon. This is followed with an analysis section which attempts to draw out the similarities and differences, and the strengths and weaknesses of the two approaches. The final section presents ideas for how the two mechanisms could be made to work more effectively in tandem, hopefully giving a unified capability with the strengths of both and the weaknesses of neither.

2. Document Projection

2.1. Overview of the Marian/Siméon Technique

The essence of the Marian/Siméon document projection technique is as follows:

1. The query is analyzed (statically) to determine the paths to all elements and attributes that are needed to evaluate the query.
2. This set of paths is used to create a projection filter. The filter is applied to parsing events issued by a (typically) SAX-based parser to that some events are passed on to a tree builder, while others are discarded. This has the effect that only a subset of the elements and attributes in the original source document are retained in the tree (for example, a DOM tree) that is built in memory.
3. If the filter is constructed correctly, then running the query against the reduced (projected) tree produces the same result as running the same query against a full tree containing all elements and attributes in the original document.

Marian and Siméon illustrated this idea using query Q1 from the XMark benchmark [4], specifically:

```
for $b in /site/people/person[@id="person0"] return $b/name
```

For convenience, the XMark queries are listed in an appendix at the end of this paper. Many people would write Q1 using the simpler XPath expression:

```
/site/people/person[@id="person0"]/name
```

But the authors of XMark came from a SQL background and like many users from that tradition, their instinct was to write every query as a FLWOR expression. And document projection tries to handle the query in whatever form the users chose to write it.

We'll make an assumption here which Marian and Siméon never state explicitly, namely that the authors of such a query expect the result to be delivered either in serialized form (as lexical XML), or as a set of parentless element nodes, specifically the `name` element and its descendants (in the XMark data, this will be a single text node, but the analysis does not depend on any schema knowledge). In particular, document projection won't work if the client application expects to be able to do arbitrary navigation from the returned elements (for example, navigating from the `name` up to its containing `person`).

The set of paths needed to evaluate this query is:

```
/site/people/person/@id
/site/people/person/name #
```

where the `#` symbol can be read as meaning "together with the subtree rooted at this node".

In the case where the query was originally expressed as a FLWOR expression using the range variable `$b`, the path analysis needs to associate a set of paths with the variable binding, and then to expand these paths for all references to the variable.

The filter operation essentially takes these paths and retains a node if (a) the node matches one of these paths, or a prefix of one of these paths (for example `/site/people`), or (b) it is a descendant (or an attribute of a descendant) of one of the paths labelled `#`.

In a refinement of the technique, called *optimized projection*, the analysis considers not only which nodes are reachable, but how they are used by the query. For example, given the query `if (@discounted) then price else price - discount`, the query tests for the existence of the attribute `@discounted`; it is capable of returning the `price` element; but on the `else` branch, the `price` and `discount` elements are always atomized (though Marian and Siméon do not use quite this terminology).

Clearly the analysis becomes increasingly complex as additional XQuery features are used. Handling additional axes such as preceding and following sibling axes is one example; use of user-written functions is another. Marian and Siméon address these complications in part by reducing the XQuery language to a small core language which is more amenable to analysis. Despite this, they don't

present the full analysis in their main VLDB paper, but refer to internal technical reports for some of the detail.

With the kind of queries used in the XMark benchmark, document projection will often dramatically reduce the size of the tree that needs to be built. With one exception, all the queries in the benchmark can be executed on a document occupying only 5% of the memory of the full tree. As well as a big saving in memory, the technique gives a (smaller) improvement in execution time, because less time is spent building and searching unnecessary parts of the tree.

The main limitations of document projection, of course, are (a) that it is only effective in cases where the source document is parsed, and a source tree built, for the purpose of executing a single query, and (b) that it is only useful in cases where the query requires a small amount of information from the document. Nevertheless, these conditions apply sufficiently often in real life that the technique should probably be far more widely known and used than it is, especially as it is available in a number of popular XQuery processors.

2.2. Document Projection in Saxon: XMark Performance

Document projection has been implemented in Saxon's XQuery processor for many years, though there is little evidence that it is widely used.[2]When running a query from the command line, it can be activated simply by use of the `-projection` command line option. Statistics on its effectiveness are displayed when the `-t` option is on[3]. From the Java API it is possible to separate the two steps of analyzing a query to create a projection filter, and applying the filter to create a projected document. (This makes it possible, if you really want, to write a query whose sole purpose is to define a document projection, and then to run a variety of queries or transformations on the projected document. In this case, of course, you lose the guarantee that the results will be the same as on the full document. More realistically, you may well want to run the same query repeatedly, against the same projected source document, but with different query parameters.)

The benefits obtained by document projection in Saxon, when running the XMark benchmark queries, are very much in line with the results reported by Marian and Siméon.

For XMark Q1, against a 119Mb source document, the metrics with document projection off are:

[2]This is perhaps an understatement. In preparing examples for this paper, I found and corrected several bugs in the implementation. Since no bugs have been reported by users over several years, this leads one to suspect that the feature is essentially unused.

[3]Some of the metrics given in this paper were obtained using internal instrumentation that is not available via any public API.

```
Analysis time: 375.619 ms
Tree built in 1405.714 ms
Tree size: 4787932 nodes, 79425460 characters, 381878 attributes
Memory used: 368,331,992 [bytes]
Average execution time [across 20 runs]: 0.233 ms
```

Switching projection on changes this to:

```
Analysis time: 373.619 milliseconds
Document projection: Input nodes 5072525; output nodes 102002; reduction ▶
= 98%
Tree built in 1032.552 ms)
Tree size: 78822 nodes, 364998 characters, 25500 attributes
Memory used: 74,333,216 [bytes]
Average execution time: 0.228 ms
```

So we're seeing a reduction in the size of text nodes alone from 79M characters to 364K characters, together with a 98% reduction in the number of element nodes, leading overall to a reduction in the total memory requirement for the query from 368Mb to 74Mb[4]. There's negligible impact on the time for static query analysis, and a useful 40% speed-up in tree building time (useful because this is the dominant cost). The query execution time is negligible compared with the tree building time, so the fact that the query runs a little faster on the projected document is of no practical importance.

For the full range of XMark queries we see the following size reductions. The first figure is the reduction in the number of nodes, the second is the reduction in the number of characters in text nodes:

```
Q1   97.99%   99.95%
Q2   96.24%   99.67%
Q3   96.24%   99.67%
Q4   96.01%   99.96%
Q5   99.42%   99.94%
Q6   99.57%  100.00%
Q7   98.69%  100.00%
Q8   97.91%   99.55%
Q9   96.62%   99.42%
Q10  91.97%   97.40%
Q11  97.28%   99.47%
Q12  97.28%   99.47%
Q13  98.79%   96.40%
Q14  88.27%   65.51%
Q15  98.90%   99.99%
Q16  98.52%   99.99%
```

[4]Figures for memory usage are not very accurate or reliable, because garbage collection happens unpredictably.

```
Q17  97.99%   99.09%
Q18  99.53%   99.96%
Q19  97.86%   99.20%
Q20  98.99%  100.00%
```

I have used slightly different metrics from those used by Marian and Siméon (they reported on the size of the document in memory and on disc, rather than the total number of nodes and the size of the text nodes) so the figures are not directly comparable; however, there is a strong correlation. Marian and Siméon reported:

> *The projected document is less than 5% of the size of the document for most of the queries. On Query 19, Projection only reduces the size of the document by 40%, and it has no effects for Queries 6, 7, and 14. In contrast, Optimized Projection results in projected documents of at most 5% of the document for all queries but Query 14 (33%). The reason for this difference is that Queries 6, 7, 14 and 19 evaluate* `descendant-or-self() (//)` *path expressions for which projection without optimization performs poorly. Query 14 is a special case since it selects a large fragment of the original auction document. Obviously projection cannot perform as well for this kind of query.*

Saxon is therefore achieving very similar results to their *Optimized Projection* results, reducing the number of nodes to 4% or less for all queries except Q14. The problem with Q14 is that the query accesses the string value of `description` elements, which account for 70% of all the text content of the source document.

2.3. Implementation of Document Projection in Saxon

Saxon performs the analysis needed to implement document projection by examining the expression tree after all query parsing, type checking, and optimization is complete.

The first stage of analysis builds a data structure called the path map, which is essentially a graph representation of all the navigation paths performed by the query expression. Some of these navigation paths are explicit in the form of an axis step in the query; others are implicit in the use of constructs like a call to `fn:id` (which effectively searches all elements in the document). The roots (entry points) to this data structure represent the global (externally-supplied) context item, and calls on functions that return new documents such as `fn:doc` and `fn:collection`. Each node in the path map represents a set of XDM nodes that is visited by the expression, and the arcs emanating from this node represent the axis steps used to navigate away from these nodes to other nodes. The initial analysis represents all axes explicitly (including, for example parent and preceding-sibling axes).

Nodes in the path map are labelled at this stage with three boolean properties: *returnable* which indicates whether the relevant nodes can be returned from the

query (rather than being merely visited en route); *atomized* which indicates that the expression atomizes the nodes reached via this path; and *hasUnknownDependencies* which is discussed below.

When an expression is bound to a variable, the set of paths reachable by the initializer of the variable is noted, and when a path expression is used that starts with a reference to this variable, the relevant navigation steps are added to the graph starting at these nodes, effectively concatenating the navigation path used to evaluate the variable with the navigation paths starting at the variable's value.

Saxon does not attempt to analyze calls to user-defined functions (nor, in XQuery 3.0, dynamic function calls): it is assumed that when a node is passed to a function call (other than a known system-defined function) with no atomization, then the function can perform arbitrary navigation starting from that node, and this is indicated by setting the property *hasUnknownDependencies* on the path map node. (In some cases, however, the optimizer will have inlined function calls, in which case path analysis is still possible.[5])

The second stage of analysis is to reduce the path map so that it contains downwards navigation steps only. It's easiest to explain how this is done with some examples:

- If the query contains the path `/books//book/preceding::item`, we replace this with the two paths `/books/descendant::book` and `/descendant::item`. The logic here is that these are the elements we need to retain in the projected document: the presence of the path `/descendant::item` will ensure that all elements named `item` are retained, wherever they appear. Because the relative position of nodes is retained by the document projection process, we can be confident that the axis step `preceding::item` will return the correct `item` elements.
- If the query contains the path `/books/book/title[contains(., ../author)]`, the original analysis will produce the two paths `/books/book/title#` and `/books/book/title/parent::node()/author`. Reduction to downwards selection then changes the second path to `/books/book/author`, because the analysis is able to determine that the pair of steps (`child::title/parent::node()`) is a null navigation and can therefore be eliminated.
- If the query contains the path `/books/book/title/following-sibling::author`, the analysis will produce the two paths `/books/book/title` and `/books/book/author`. Again the analysis is able to determine that navigating down to a `title` element and then across to a sibling `author` element will always select an `author` that is a child of the `book` element.

The third and final step is the actual process of document projection. This is implemented as a filter (an instance of the Saxon class `ProxyReceiver`) on the

[5]The only XMark query to use a user-defined function is Q18, and this function is trivially inlineable.

push-based event pipeline between the SAX XML parser and the tree builder. The filter maintains a stack of currently opened-but-not-yet closed elements; on this stack is a reference to the path in the path map by which the nearest retained ancestor node was reached. An element is retained if (a) its parent is retained, and (b) there is an arc on the path map that selects this element from its parent or ancestor. Elements are also retained (c) if some ancestor node was marked with the `returnable` or `atomized` properties indicating that the entire subtree of the element is required. (Atomization actually only requires the descendant text nodes: descendants of other kinds could be discarded. But when elements are atomized, they almost invariably have a single text node child, so this additional optimization would deliver very little benefit.)

3. Document Streaming

3.1. Overview of Streaming in XSLT 3.0

Many constructs in XPath and XSLT (and XQuery, for that matter) do not lend themselves well to streamed evaluation: the semantics of the language are defined in terms of a tree that can be freely navigated in all directions (including, importantly, upwards). In principle one could identify a subset of stylesheets that only use streamable constructs, and implement these using streaming algorithms that avoid building the entire tree in memory. However, it is likely that very few real-life stylesheets would fall into this category. One particular reason for this is that XSLT, unlike XQuery, relies heavily on dynamic despatch of template rules, which makes it effectively impossible to perform static analysis of the stylesheet as a whole.

In addressing the requirement for streaming to process large documents, the XSL Working Group therefore adopted a different approach:

- Users would be required to indicate an intent that particular parts of the stylesheet (for example, template rules) should be streamable, and the XSLT processor would be required to analyze those parts for streamability.
- It should be possible within a single stylesheet to mix streamed and unstreamed processing; for example, it should be possible during streamed processing of a large document to build an in-memory representation of a small subtree of this document, and then process this subtree using the full power of the language unconstrained by streaming restrictions.
- New constructs should be added to the language to make it easier to write streamable stylesheet code. Examples of such constructs are the `xsl:merge` instruction (which allows multiple documents to be merged in a streamed operation), the `xsl:accumulator` declaration (which allows multiple aggregation functions, such as totalling and averaging, maxima and minima, to be be

computed during a single streaming pass of the document), and the `xsl:on-empty` and `xsl:on-non-empty` which make it possible to specify declaratively what should happen when required input data is absent.

XQuery is designed in the tradition of database query languages, whose developers typically aspire to the principle that optimization is entirely the responsibility of the language processor, not the user. XSLT comes more from the tradition of programming languages, where performance and selection of algorithms are the responsibility of the programmer. It is therefore to be expected that the two languages would take a different approach to streaming.

The static analysis determining whether particular XSLT constructs are streamable is complex. The rules are prescribed in the language specification, so that stylesheet authors can be confident that code designed to be streamable with one implementation will also be streamable with other implementations.

The analysis assumes the existence of an expression tree representing the results of parsing the stylesheet and the XPath expressions embedded within it. Each node in this tree represents an instruction or expression (generically, a *construct*). The analysis computes three properties for every construct:

- A static type (to which the result of evaluating the construct will always conform). The static type analysis is fairly simplistic and does not attempt to be over-precise. It is used mainly to distinguish expressions that return childless nodes (such as text nodes and attributes) from those that can return nodes with children (documents and elements), because this can make a difference to streamability.
- The *sweep* of the construct. Constructs are classified as *motionless*, *consuming* or *free-ranging*. This concept relies on the notion that when processing a document in streaming mode, there is a current position in the document representing the moving cursor that separates tags that have already been read, from tags that have not yet been read. A *motionless* construct is one that can be evaluated without moving this cursor: an example of such a construct might be `exists(@foo)` (because the processing model assumes that when the cursor is positioned on an element start tag, all the attributes of that element are available). A *consuming* construct is one that can be evaluated by moving the cursor from a start tag to the corresponding end tag, that is, by reading the descendants of the current element. An example of such a construct is `. = "foo"` (if the context item is an element, comparing its typed value to a given string requires atomizing the element, which involves reading its descendants). The third category, *free-ranging*, represents everything else: constructs like `following-sibling::x` which require navigation outside the boundaries of the current element. If the context item is a node in a streamed document (an important caveat), then a free-ranging construct leads to the stylesheet being deemed non-streamable.

- The *posture* of the construct. This concept is rather abstract and difficult to explain in simple terms; it characterizes the relationship of nodes selected by the evaluation of the expression to the position of the moving cursor. It also constrains what further navigation is allowed starting from the result of this expression.

  Streamed processing produces three possible postures. A *crawling* posture represents the result of a consuming expression that is processing all the descendants of an element, in the course of moving the cursor from the start tag to the end tag of that element. A *striding* posture is very similar, except that the expression is only processing descendant elements at a fixed depth (for example, children or grandchildren, but not a mixture of both). A *climbing* expression is one that navigates to ancestors of the element at the cursor position, or to attributes of these ancestors (the streaming model assumes that a stack of ancestor nodes and their attributes is available at all times). The key difference between these three postures is what kind of onwards navigation is permitted. When a node was reached in climbing posture (that is, by following the ancestor axis), no downward navigation is permitted, because in general this would need to access parts of the document that have already been read and discarded, or that have not yet been read. In striding posture, further downward navigation is allowed. It crawling posture, downward navigation is not allowed in the general case, because when two nodes A and B are reached in crawling posture, and A comes before B in document order, it is not necessarily the case that all descendants of A appear in document order before all descendants of B: for an example, consider the element:

```
<root>
  <section id="A">
    <section id="B">
      <footer/>
    </section>
    <footer/>
  </section>
</root>
```

  where the `footer` child of section A appears in document order *after* the `footer` child of section B. If downward selection were permitted from the results of a crawling expression such as `descendant::section`, it would not be possible to achieve this by processing one `section` at a time without buffering.

  The other two values for posture are *grounded* and *roaming*. Grounded posture applies to an expression whose result will never contain streamed nodes: for example, expressions whose value is an atomic value or a map, or a node in an unstreamed document. There are no streaming restrictions on the processing that can be applied to the result of a grounded expression. Roaming

posture, by contrast, represents the case where streamability analysis has essentially failed: it applies to an expression such as `preceding::item`, which cannot be evaluated in streamed mode.

Although the XSLT 3.0 specification is very prescriptive about the analysis used to determine whether constructs are deemed streamable or not, it has very little to say about how streamed evaluation of those constructs deemed to be streamable should actually work. That is left entirely to the implementation.

3.2. Streaming in Saxon: XMark Performance

Although streaming is specified by W3C only for XSLT 3.0, Saxon also offers the capability of streamed XQuery execution. If a query is run from the command line using the `-stream` option, Saxon will analyze the query for streamability using rules that are essentially equivalent to those of the `xsl:stream` instruction in XSLT 3.0, and if it is streamable, will execute it in streamed mode. Since we have been studying the set of XMark benchmark queries to examine the impact of document projection, it is instructive to look at the same queries from a streamability perspective.

It turns out that none of the XMark queries is streamable (in Saxon) as originally written, but many of them can easily be rewritten to make them streamable. The main changes required are described below:

- Most of the queries are written as FLWOR expressions, which are typically not streamable (a) because they use non-XPath syntax which is therefore outside the scope of the streamability rules in the XSLT specification, and (b) because they bind variables to streamed nodes. In most cases the rewrite is very simple. For example Q2 is originally written:

```
for $b in /site/open_auctions/open_auction return <increase> { $b/▶
bidder[1]/increase } </increase>
```

  which can be rewritten like this[6] to make it streamable:

```
<xsl:for-each select="/site/open_auctions/open_auction">
   <increase>
      <xsl:sequence select="bidder[1]/increase"/>
   </increase>
</xsl:for-each>
```

- Some of the queries require more extensive rewriting to avoid multiple downward selections. For example Q5 is originally:

[6]This rewrite preserves the bug in the original query whereby the output contains two levels of nested `<increase>` elements.

```
count(for $i in /site/closed_auctions/closed_auction
      where  $i/price >= 40
      return $i/price)
```

which (knowing that a `closed_auction` has only one `price`) can be made streamable by rewriting as:

```
count(/site/closed_auctions/closed_auction/price/number()[. >= 40])
```

Similarly, Q7 reads

```
for $p in /site
return count($p//description) + count($p//annotation) + count($p//▶
email)
```

which can be replaced by the streamable equivalent:

```
count(site//description | site//annotation | site//email)
```

- A few queries require limited buffering of the input. An example is Q4:

```
for    $b in /site/open_auctions/open_auction
where  $b/bidder/personref[@person="person18829"] <<
       $b/bidder/personref[@person="person10487"]
return <history>{ $b/reserve }</history>
```

where the processing can be done by building each `open_auction` in turn as a tree in memory, which can then be discarded to process the next `open_auction` (sometimes called "burst-mode streaming"). This can be achieved with the rewrite:

```
/site/open_auctions/open_auction/copy-of(.)
  [bidder/personref[@person="person18829"] << bidder/▶
personref[@person="person10487"]]
  ! <history>{ reserve }</history>
```

The queries that remain stubbornly unstreamable are those that involve joins (Q8, Q9, Q10, Q11, Q12) or sorting (Q19).

The performance of streamed versus unstreamed versions of the same query (for those queries where streaming is possible) is very consistent:

- For unstreamed execution, there is a one-off cost of building the document tree of around 1400ms, and the cost of evaluating the query is then between 2ms and 50ms.
- For streamed execution, the cost of query execution is in each case between 930ms and 1100ms.
- For comparison, we noted earlier that unstreamed execution against a projected document tree typically reduces the tree-building time from 1400ms to 1000ms, with a very small (and therefore negligible) improvement in query execution time.

We can see that both streaming and document projection (where applicable) give a modest improvement in execution time and a very substantial saving in memory usage, at the cost of having to parse the source document to run a single query: and it is the parsing cost that dominates. It's also worth noting that the query compilation cost, at 375ms, exceeds the document building cost for documents smaller than around 40Mb, even with these very simple queries.

3.3. Streaming in Saxon: Implementation

Saxon, in its latest incarnation (the current public release is 9.7 but this section applies more strictly to the planned 9.8 release), follows the streamability analysis rules in the specification very literally. It has to, because the Working Group has insisted on a clause that says that a processor that wants to claim conformance to the streaming feature must be prepared to report whether a stylesheet is streamable according to the W3C rules or not — no extensions allowed.

A difficulty here is that the streamability properties computed during the analysis are actually required in order to devise a streamed execution strategy, and they must therefore be computed for the final expression tree produced after all optimization rewrites have been completed. But some of the optimization rewrites (for example, function inlining) may convert a non-streamable construct into a streamable one, and would therefore cause Saxon's streamability verdict to differ from the W3C verdict. At user request, there is therefore an option to run the streamability analysis twice: the first run is done before all optimizations to deliver a W3C-conformant streamability assessment, and the second run is done after optimization to produce input to the execution plan.

The formalisms used in the W3C streamability analysis have inspired some changes to the design of Saxon's expression tree. Most notably, every expression now delivers information about its subexpressions (called *operands*) in a consistent way. The relationship between a parent expression and its children is now represented by an `operand` object which contains properties closely based on properties used in the W3C streamability model: for example every operand has a *usage* which explains how the parent expression makes use of the result of evaluating the child expression: this is one of *inspection* (the properties of the returned value are examined, for example using an `instance of` operator), *absorption* (the entire subtree of any nodes in the returned value is used, typically by means of atomization), *transmission* (the parent expression includes the result of the child expression directly in its own result value), or *navigation* (the parent expression performs arbitrary navigation from the nodes in the child expression's result to other nodes in the same tree).

Saxon has started to use these formalisms in other aspects of its static analysis unrelated to streaming (for example, in the analysis done to construct the path map for document projection) but there is much potential for increased reuse in this area.

The most complex aspect of streaming in Saxon, however, has nothing to do with the static analysis of streamability according to W3C rules, but rather with creating a streamable representation of the execution plan for the stylesheet, and with the actual execution of that plan.

As discussed in [3] and [5], streaming in Saxon operates in push mode. *Push* here means that the control flow is in the same direction as the data flow. This is the reverse of the usual interpreter design pattern. Instead of a parent expression requesting (pulling) data obtained from the evaluation of its subexpressions, the arrival of data from the XML parser triggers the evaluation of subexpressions dependent on that data, which in turn triggers the evaluation of parent expressions dependent on the results of those subexpressions. That is to say, the control flow is inverted: and in fact, the process of generating a push-mode representation of the stylesheet code corresponds to the process of program inversion as described many years ago in Jackson Structured Programming [2].[7]

The streaming code in Saxon essentially works in three parts:

1. The first part is the static streamability analysis. As already mentioned, this follows the rules in the XSLT 3.0 specification very closely. Every kind of expression node in the expression tree (representing different kinds of XSLT and XPath construct in the source code) has logic to compute properties such as the posture and sweep of the expression. Once computed, these are retained as cached properties on the expression tree. The logic for many expressions is delegated to the general streamability rules, which are driven by information about the operands of the expression and their usage; all expressions deliver their operands (including properties such as the operand usage, whether the focus changes, and suchlike) using a common interface.

2. The second part is the logic for expression inversion. Where a template or other construct is declared to be streamable, and where analysis reveals that it is actually streamable, the process of inversion constructs a representation of the construct suitable for streamed push-mode evaluation. In general (with some exceptions such as `xsl:choose`, `xsl:fork`, and `xsl:map`) a consuming expression will have exactly one consuming operand. It is therefore possible to identify a route through the expression tree that contains these consuming expressions. For the lowest-level such expression, we allocate a `Watch` which is triggered when the parser encounters a node that matches a particular (motionless) pattern. For each consuming ancestor expression, we allocate a `Feed` which evaluates a particular construct in push (event-driven) mode.

3. The third part comprises the push-mode implementations of every kind of instruction, expression, or system function. For example, the `Feed` for a call on

[7] This is no coincidence, because JSP was greatly concerned with the problems of managing hierarchic data in a streamed representation too large to fit in available memory, in this case using magnetic tape.

the `fn:sum` might look like this (ignoring subtleties in the actual specification of this function):

```
private double sum = 0;

public processItem(Item item) {
    sum += item.toDouble();
}

public void close() {
    getParentFeed().processItem(new DoubleValue(sum));
}
```

In this particular example, the implementation does not pass anything on to the `Feed` for its parent expression until the input sequence is exhausted. In other cases, for example the `Feed` for the `fn:distinct-values` function, values can be passed on as soon as they are known. Here is a simplified version of the `Feed` for `fn:distinct-values`:

```
private Set<AtomicValue> values = new HashSet<AtomicValue>();

public processItem(Item item) {
    if (values.add((AtomicValue)item)) {
        getParentFeed().processItem(item);
    }
}
```

When the processing of an input sequence is initialized, this code creates a new data structure holding the set of distinct values (which is initially empty). Each time a new value is received, it is added to this set (which is a no-op if it was already present in the set); and if it was not already present in the set, it is passed on to the feed for the parent expression.

These push-mode implementations need to be provided for every kind of construct that can take a sequence as input. For expressions that operate on singletons (for example, arithmetic expressions) a generic `Feed` that accepts a single item and invokes the ordinary non-streaming implementation suffices. For a few constructs it is necessary to provide more than one push-mode implementation, because the logic depends on which operand is the streaming operand: a notable example is the `LetExpression` (used not only to implement XPath `let` expressions, but also local variables declared in XSLT: the code for `let $x := distinct-values(//foo) return count($x)` is quite different from the code for `let $x := data(@id) return index-of(.//foo, $x)`, because in the first case the streamed operand is the expression used to initialize the variable, and in the second case the streamed operand is the expression that makes use of the variable.

It's worth remarking in passing that because the control flow is inverted, we can't do dynamic error handling using the normal Java machinery of throwing and catching exceptions. When evaluation of an expression fails, this must cause execution of the parent expression to fail, so the failure is pushed up the pipeline in the same way as success results.

4. Projection and Streaming: a Comparison

The major similarities between the two technologies are:

- They share the objective of using static analysis of a query or stylesheet to establish an execution strategy that reduces the amount of memory needed for the tree representation of large documents.
- There are many similarities in the details of the analysis that is carried out, although it is presented in rather different terms.
- In both cases, the technique is only effective if the source document is being parsed only for the purpose of executing a single query or stylesheet against it. Document projection has a bit more flexibility in that it allows the same query to be executed repeatedly with different parameters (for this use case, streaming requires the document to be re-parsed each time, document projection does not). If the workflow requires multiple queries or stylesheets to be executed against the same source document, neither streaming nor document projection is well suited to the task.

Some significant differences between the two approaches are:

- Document projection is defined for use with XQuery, streaming for use with XSLT. This difference is not entirely superficial, because the nature of the rule-based template processing characteristic of XSLT means that the potential for static analysis is very different from the XQuery case.
- Document projection can be applied to any query. It may be more effective for some queries than others (for some queries the memory requirement might be reduced by 5%, for others by 95%), but the mechanism is designed to work with any query. By contrast the streamability analysis in XSLT makes a binary classification of stylesheets as being either streamable or not streamable, and a stylesheet in the latter category achieves no benefit.
- Document projection typically achieves a linear reduction in the amount of memory needed (for example, to 20% of the original memory requirement), whereas streaming is designed to run in constant memory completely independently of document size. This means that streaming can even be applied to infinite documents, such as a continuous stream of telemetry data.
- Document projection is designed to happen entirely behind-the-scenes, without the knowledge or involvement of the query author, whereas XSLT stream-

ing provides explicit constructs for the invocation of streaming, and alternative ways of writing code when streaming is required, including brand-new mechanisms such as accumulators. This in turn reflects a philosophical difference between XQuery and XSLT, or between query languages and programming languages in general, where the intellectual focus in query language theory has always been automated optimization, while the focus for programming languages has been enabling the programmer to achieve the desired performance metrics as well as correct execution.

5. Seeking Synergy

In this section we'll start looking for ways in which the benefits and of streaming and document projection can be combined, and in which the weaknesses of both can be reduced. We'll start with an example to illustrate the challenges.

5.1. A Simple Example

Consider the following query, which computes the average value of the transactions in a transaction file:

```
[R1] avg(//transaction/(price * quantity))
```

As we've seen, many XQuery users, especially those trained in SQL, will write this by preference as:

```
[R2] avg(for $t in //transaction return $t/price * $t/quantity)
```

Document projection will analyze either of these queries and establish that the only nodes that need to be retained when building the source document tree are the document node, the `transaction` element nodes, and the `price` and `quantity` elements together with their text node descendants (or their typed value, if the processing is schema-aware). This may be a very substantial reduction on the size of the source tree that would otherwise be built, but it is still proportional to the number of transaction elements in the file, so it will run out of memory eventually.

Streamability analysis in XSLT 3.0 will reject both these queries as non-streamable. The second query (R2) is rejected because it binds variables to streamed nodes, and the streamability analysis in the XSLT specification gives up at this point. The first query (R1) is rejected because there is an expression `(price * quantity)` that makes two downward selections in the tree. In the language of the specification, it has two operands whose sweep is consuming. In implementation terms, the difficulty is that in the general case, an expression with multiple downward selections cannot be evaluated without buffering data in memory, and the amount of buffering cannot be predicted, and therefore the query cannot execute in constant memory so it cannot be said to be full streamable. Of course, you

and I can see that in this case the amount of buffering needed is trivial – but that is only because we know instinctively what kind of values we expect to find in elements named `price` and `quantity`.

There's another more subtle difficulty with streaming of this expression: because the `transaction` elements are selected using the shorthand notation `//transaction`, the processor has to be prepared to handle nested transactions: the query has a well-defined outcome for a pathological input such as:

```
<transaction>
  <price>8.25</price>
  <transaction>
     <price>13.50</price>
     <quantity>4</quantity>
  </transaction>
  </quantity>13</quantity>
</transaction>
```

The XSLT 3.0 solution to this is to ask the programmer to rewrite the query in a way that makes the buffering explicit, reflecting the fact that the programmer understands the data much better than any optimizer. Typically it will be rewritten as:

```
[R3] avg(/*/transaction/copy-of(.)/(price * quantity))
```

The injected `copy-of(.)` step explicitly makes an in-memory copy of the transaction element and its subtree (unless, of course, the optimizer finds a smarter way of doing things), and the navigation to the `price` and `quantity` elements is then conventional (unstreamed) navigation within an in-memory tree.

We can see that neither the projection approach nor the streaming approach is perfect here. Document projection uses more memory than it needs, because it fails to recognize that the `avg()` function only needs access to one transaction at a time (the only memory it needs is, at most, a running total and a running count). Streaming requires programmer intervention to copy the transaction nodes; and these subtrees are bigger than they need to be because we copy the whole subtree rather than merely the price and quantity elements. (In some transaction files the transaction element might be very large, for example it might contain an entire history of contract negotiations leading up to an eventual purchase.)

It seems evident that there's room for improvement in how we execute this query. We should be able to get the streaming benefits of only holding one transaction in memory at a time. We should be able (in some mode of operation) to automate the `copy-of()` operation, and we should be able to use a projection-like technique to ensure that this copy contains only the required elements (price and quantity) rather than the full transaction element.

The following sections examine these opportunities in more detail.

5.2. Tolerating Local Variables

The static analysis performed for document projection can handle the existence of local variable bindings within the expression, whereas the streamability analysis in XSLT 3.0 fails as soon as a streamed node is bound to a local variable (with one exception, the case where the variable is a reference to the first argument of a streamable stylesheet function). As we've seen, local variables are used liberally in practical queries, and it would be nice to handle them if we can.

An earlier draft [7] of the XSLT 3.0 specification did in fact attempt this (and indeed, it worked in terms of a path map that was explicitly inspired by Marian and Siméon).

The most obvious way in Saxon to make queries such as

```
for $b in /site/people/person[@id="person0"] return $b/name
```

streamable is to rewrite the query during the optimization phase to eliminate the local variable, relying instead on binding the context item. Saxon already does this for let expressions, in cases where the variable is only used once. Saxon also turns where clauses into predicates when possible, so for example

```
for $e in /*/employee where $e/salary > 10000 return $e/name
```

is rewritten as

```
for $e in /*/employee[salary > 10000] return $e/name
```

Rewriting it as

```
/*/employee[salary > 10000] ! name
```

is no more difficult. (I've used the "!" operator here as the direct equivalent of the original semantics, because there is no de-duplication or sorting into document order. However, the expressions X!Y and X/Y are equivalent in the case where both X and Y have striding posture, as is the case here.)

The basic condition for doing such a replacement is that the focus for evaluating the variable reference is the same as the focus for evaluating its declaration.

5.3. Tolerating Multiple Consuming Operands

We have seen that an expression such as

```
avg(/*/transaction/(price * quantity))
```

fails the streamability rules because it makes two downward selections (or in the language of the spec, it contains two consuming operands). But it can be made streamable by rewriting it as:

```
avg(/*/transaction/copy-of(.)/(price * quantity))
```

Two questions arise: (a) can this rewrite be automated, and (b) can we do better, by only copying the `price` and `quantity` elements, and not the whole `transaction`?

The danger in automating this rewrite is that in the worst case, it could generate a copy of the entire streamed document, thus effectively masking the fact that the query is not really streamable.

Rather than generating an implicit copy, another approach to making this streamable is to generate an implicit `xsl:fork` instruction. The `xsl:fork` instruction allows multiple consuming sub-expressions to be computed during a single pass of the input document, buffering the results of each sub-expression and combining them when the relevant subtree of the input document has been completely consumed.

Map constructors also allow multiple consuming operands, so as an alternative to `copy-of()` or `xsl:fork`, it is possible to make this streamable by writing:

```
avg(/*/transaction/map{"price":data(price), "qty":data(quantity)}!(?
price * ?qty)
```

(Note the need to explicitly atomize the streamed elements by calling `data()`, because streamed nodes cannot be stored in a map.)

Generating a projection of the input subtree can be seen as a further possibility.

So there is a range of possible ways foward that could make it easier to write streamable applications that require multiple downward selections. The main challenge, in fact, is to distinguish those cases that can be done with minimal buffering, like this one, from those where buffering data would essentially mean that the data was not being streamed at all. So it comes down to static analysis. We can see that this case works because the operator with multiple consuming operands (that is, the multiplication operator) requires those operands to be singleton atomic values. Perhaps this is the case to tackle first. This would mean relaxing the general streamability rules in the XSLT 3.0 specification so that the rule starting "If more than one operand is potentially consuming..." has an additional clause:

> *If each of the potentially consuming operands has operand usage absorption and has a required type with item type atomic and cardinality zero or one, then [the expression is] grounded and consuming.*

Implementing streaming for this case would not be difficult; the logic would essentially be the same as for map constructors.

5.4. Document Projection in XSLT

One reason that document projection has been relatively little used is that it only works in XQuery. Apart from the context of XML databases (where streaming

and document projection are not relevant), XSLT is far more widely used than XQuery. So we need to ask the question whether document projection could be implemented in an XSLT context.

The traditional difficulty here has been that XSLT does not permit the kind of static analysis necessary, because of the dynamic nature of template rules. However, this was also a problem for streamability analysis, and in that case the problem has been successfully overcome. The way it was overcome was to allow a mode to be declared as streamable, and to require all template rules in a streamable mode to conform to a number of statically-checkable constraints: chiefly, the match pattern must be motionless, the body of the template rule must be either motionless or consuming, and the body must be grounded (that is, the template must not return streamed nodes).

There's another potential objection to doing document projection in XSLT, which is that many transformations use almost all the data in the source tree when constructing the result tree. (Or to put it another way, XSLT is used for transformation rather than for query.) However, the fact that the technique isn't appropriate to all uses cases doesn't mean that it isn't appropriate to any, and it's easy enough to find examples of stylesheets that extract a small part of the input document.

The obvious place to start document projection in XSLT 3.0 is with the new `xsl:source-document` instruction. This could be given an attribute `projection="yes"` to be used when the stylesheet author thinks that use of document projection would be beneficial.

The main challenge in implementing this is the need to define the analysis rules determining how all XSLT instructions affect the path map. Some of these instructions like `xsl:number` are quite complex. The rules also need revisiting to consider new constructs that have appeared in XPath 3.0, notably dynamic function calls. However, this challenge also presents an opportunity: many of the concepts introduced for the sake of streamability analysis are probably reusable to create general rules for projection analysis. For example:

- The concept of *operand usage* distinguishes four kinds of processing that may be applied to the nodes returned by an expression: *inspection* reads properties of the node and performs no further navigation; *absorption* reads the subtree rooted at a node; *transmission* includes the nodes returned by a sub-expression in the result of the parent expression; and *navigation* permits arbitrary navigation from a node to anywhere else in the tree. These concepts are just as applicable to projection analysis as to streaming analysis, and the fact that all constructs already classify their operands according to these categories means that it should be possible to eliminate many ad-hoc rules that currently appear in the projection analysis.

- The concept of a *focus-setting container* with *controlling* and *controlled* operands is used by the streamability analysis to trace paths where there is a dependency on the context item. Again, Saxon already performs this analysis for streamability purposes, and this could be used to eliminate many ad-hoc rules in the projection analysis.
- In the opposite direction, some of the concepts used for path analysis could be exploited to improve streamability analysis. For example, code that uses the `following-sibling` axis is currently non-streamable; but one can envisage cases where path analysis could be used to identify cases where streaming of this axis is possible.

Performing data-flow analysis across calls of functions and templates remains challenging. The current document projection code in Saxon does not attempt it even for static function calls, let alone dynamic function calls or `xsl:apply-templates`. The new packaging facilities in XSLT 3.0, which allow a function to be overridden in another separately-compiled package, complicate this even further. The approach used for streamability analysis, which relies on modes and functions being manually annotated to describe the scope of their navigation, might be one way forward: realistically, however, only a very small minority of users will understand such features well enough to gain benefit from them. In addition, document projection works best for simple queries, so it might be best to concentrate on doing the best possible job in simple cases.

5.5. Automatic Streaming and Projection

The fact that document projection has been so little used should remind us of an awkward truth: any feature that delivers performance benefits, but needs to be actively enabled by users, will be ignored by the vast majority of the user population. Users are too busy to research all the tools available. If a job is run every day, they will tolerate the fact that it takes two minutes to run rather than investigating ways of reducing the run time to ten seconds.

By contrast, a feature that delivers improved performance "out of the box" will give more users an improved experience and will enhance the reputation of the product in the marketplace.

So we should ask the question whether there are situations where document projection (and indeed streaming) can be enabled automatically, by default.

In some cases we know that a query or stylesheet is only being executed once, and that a source document is being built in memory to be processed once. The main example of this is when the query or transformation is run from the command line. Unfortunately other cases, such as running within Ant, are harder to detect, because products like Ant use a general-purpose compile-build-transform API where the stages are invoked independently of each other. However, providing a "single-shot" API could achieve the twin benefits of (a) providing a much

simpler-to-use interface for applications, and (b) enabling the query or transformation processor to know that there are no hidden complexities.

That then raises the question of whether it is feasible and cost-effective to examine a query or stylesheet to decide whether document projection should be used. Or indeed, whether it would do any harm to use it unconditionally. The worst that can happen is that a slight extra cost is incurred during query analysis, and a slight extra cost during document building, in those cases where there is no benefit.

We could start with the following experiment: when running XQuery from the command line, if the query contains no user-defined functions that take nodes as arguments, then enable document projection by default.

If nothing else, this would at least ensure that any bugs in the projection code are quickly discovered.

References

[1] Amélie Marian Jérôme Siméon Projecting XML Documents Proc VLDB 29, Berlin, Germany, 2009 https://www.cs.rutgers.edu/~amelie/papers/2003/xmlprojection.pdf

[2] M. A. Jackson: Principles of program design. academic press, London, 1975.

[3] Michael Kay 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. Available at https://www.balisage.net/Proceedings/vol3/html/Kay01/BalisageVol3-Kay01.html

[4] XMark http://www.xml-benchmark.org

[5] Michael Kay Streaming in the Saxon XSLT Processor Presented at XML Prague 2014. Available at http://archive.xmlprague.cz/2014/files/xmlprague-2014-proceedings.pdf

[6] Saxon http://www.saxonica.com/

[7] XSL Transformations (XSLT) Version 2.1. W3C Working Draft (superseded), 11 May 2010. Ed. Michael Kay. http://www.w3.org/TR/2010/WD-xslt-21-20100511/

[8] XSL Transformations (XSLT) Version 3.0. W3C Candidate Recommendation, expected to be published 7 February 2017. Ed. Michael Kay. http://www.w3.org/TR/xslt-30

A. Appendix: XMark Queries

This appendix lists the XMark queries in the form they were run for the document projection tests.

They differ from the queries published at `http://www.xml-benchmark.org` in taking input from the context document rather than using the `doc()` function: this was done for convenience, allowing the same query to be used with different-sized input files.

Table A.1. XMark Queries

Name	Query
Q1	`for $b in /site/people/person[@id="person0"]` `return $b/name`
Q2	`for $b in /site/open_auctions/open_auction` `return <increase> {$b/bidder[1]/increase } </increase>`

Q3	`for $b in /site/open_auctions/open_auction`
	`where $b/bidder[1]/increase *2 <= $b/bidder[last()]/increase`
	`return <increase first="{$b/bidder[1]/increase}"`
	`last="{$b/bidder[last()]/increase}"/>`

Q4	`for $b in /site/open_auctions/open_auction`
	`where $b/bidder/personref[@person="person18829"] <<`
	`$b/bidder/personref[@person="person10487"]`
	`return <history>{ $b/reserve }</history>`

Q5	`count(for $i in /site/closed_auctions/closed_auction`
	`where $i/price >= 40`
	`return $i/price)`

Q6	`for $b in /site/regions/*` `return count ($b//item)`
Q7	`for $p in /site` `return count($p//description) + count($p//annotation) + count($p//email)`

Q8	`let $a := for $t in /site/closed_auctions/closed_auction`
	`where $t/buyer/@person = $p/@id`
	`return $t`
	`return <item person="{$p/name}"> {count ($a)} </item>`

Name	Query

Q9	`let $auction := (/) return`
	`let $ca := $auction/site/closed_auctions/closed_auction return`
	`let`
	`  $ei := $auction/site/regions/europe/item`
	`for $p in $auction/site/people/person`
	`let $a :=`
	`  for $t in $ca`
	`  where $p/@id = $t/buyer/@person`
	`  return`
	`  let $n := for $t2 in $ei where $t/itemref/@item = $t2/@id return $t2`
	`  return <item>{$n/name/text()}</item>`
	`return <person name="{$p/name/text()}">{$a}</person>`

Q10	`declare boundary-space strip;`
	`for $i in distinct-values(`
	`  /site/people/person/profile/interest/@category)`
	`let $p := for  $t in /site/people/person`
	`  where  $t/profile/interest/@category = $i`
	`  return <personne>`
	`  <statistiques>`
	`  <sexe>{ $t/profile/gender }</sexe>`
	`  <age>{ $t/profile/age }</age>`
	`  <education>{ $t/profile/education}</education>`
	`  <revenu>{ $t/profile/@income } </revenu>`
	`  </statistiques>`
	`  <coordonnees>`
	`  <nom>{ $t/name }</nom>,`
	`  <rue>{ $t/address/street }</rue>`
	`  <ville>{ $t/address/city }</ville>`
	`  <pays>{ $t/address/country }</pays>`
	`  <reseau>`
	`  <courrier>{ $t/emailaddress }</courrier>`
	`  <pagePerso>{ $t/homepage }</pagePerso>`
	`  </reseau>`
	`  </coordonnees>`
	`  <cartePaiement>{ $t/creditcard }</cartePaiement>`
	`  </personne>`
	`return <categorie>`
	`  <id>{ $i }</id>`
	`  { $p }`
	`  </categorie>`

Q11	`for $p in /site/people/person`
	`let $l := for $i in /site/open_auctions/open_auction/initial`
	`  where $p/profile/@income > (5000 * $i)`
	`  return $i`
	`return <items name="{$p/name}">{ count ($l) }</items>`

Name	**Query**

Q12	`for $p in /site/people/person`
	`let $l := for $i in /site/open_auctions/open_auction/initial`
	`  where $p/profile/@income > (5000 * $i)`
	`  return $i`
	`where  $p/profile/@income > 50000`
	`return <items person="{$p/name}">{ count ($l) }</items>`

Q13	`for $i in /site/regions/australia/item` `return <item name="{$i/name}">{ $i/description }</item>`

Q14	`for $i in /site//item`
	`where contains ($i/description,"gold")`
	`return ($i/name, $i/description)`

Q15	`for $a in /site/closed_auctions/closed_auction/annotation/`
	`  description/parlist/listitem/parlist/listitem/text/emph/keyword`
	`return <text>{ $a }</text>`

Q16	`for $a in /site/closed_auctions/closed_auction`
	`where exists ($a/annotation/description/parlist/listitem/parlist/`
	`  listitem/text/emph/keyword/text())`
	`return <person id="{$a/seller/@person}" />`

Q17	`for  $p in /site/people/person`
	`where  empty($p/homepage/text())`
	`return <person>{$p/name}</person>`

Q18	`declare namespace f="http://f/";`
	`declare function f:convert ($v)`
	`{`
	`  2.20371 * $v (: convert Dfl to Euro :)`
	`};`
	
	`for  $i in /site/open_auctions/open_auction`
	`return f:convert($i/reserve)`

Q19	`for  $b in /site/regions//item`
	`let  $k := $b/name`
	`order by $k`
	`return <item name="{$k}">{ $b/location } </item>`

Name	**Query**

Q20	`<result>`
	` <preferred>`
	`  {count (/site/people/person/profile[@income >= 100000])}`
	` </preferred>`
	` <standard>`
	`  {count (/site/people/person/profile[@income < 100000`
	`  and @income >= 30000])}`
	` </standard>`
	` <challenge>`
	`  {count (/site/people/person/profile[@income < 30000])}`
	` </challenge>`
	` <na>`
	`  {count (for  $p in /site/people/person`
	`  where  empty($p/profile/@income)`
	`  return $p)}`
	` </na>`
	`</result>`