A Transducer-Based XML Query Processor*

Bertram Ludäscher† Pratik Mukhopadhyay‡ Yannis Papakonstantinou‡

†San Diego Supercomputer Center
University of California, San Diego
ludaesch@sdsc.edu

‡Department of Computer Science & Engineering
University of California, San Diego
{pmukhopa,yannis}@cs.ucsd.edu

Abstract

The XML Stream Machine (XSM) system is a novel XQuery processing paradigm that is tuned to the efficient processing of sequentially accessed XML data (streams). The system compiles a given XQuery into an XSM, which is an XML stream transducer, *i.e.*, an abstract device that takes as input one or more XML data streams and produces one or more output streams, potentially using internal buffers. We present a systematic way to translate XQueries into efficient XSMs: First the XQuery is translated into a network of XSMs that correspond to the basic operators of the XQuery language and exchange streams. The network is reduced to a single XSM by repeated application of an XSM composition operation that is optimized to reduce the number of tests and actions that the XSM performs as well as the number of intermediate buffers that it uses. Finally, the optimized XSM is compiled into a C program. First empirical results illustrate the performance benefits of the XSM-based processor.

1 Introduction

XML is the standard for information exchange between applications and information sources. For example, Web service implementations and mediators exchange XML messages and data, and often interact with information systems and databases via XML import/export mechanisms. Web front-ends receive XML from the underlying information sources and transform them into XHTML. Migration of XML archives [MBR+00] to new, "refreshed" schemas requires XML tranformations. Continuous data streams from sensor networks [CFGR02] are transformed into formats that can be readily consumed, *e.g.*, triggering some procedures after detecting certain events [ROA02]. The common denominator of the examples above is that some sequentially accessed XML data needs to be efficiently accessed and transformed into some output XML data that fits the format required by the consuming application.

A large number of important applications require extremely efficient processing and transformation of sequentially accessed streams. For example, there are many ongoing works on efficient XSLT processors [Res02, SAB, XAL, Kay], which convert XML files or strings into XHTML. Other novel streaming applications, *e.g.*, efficient XML-based information filtering do not require the expressive power of general XML queries and transformation in the style of XQuery, but are based on a limited subset, *i.e.*, XPath queries [AF00, CFGR02].

The importance of efficient (XML) stream processing will keep growing as communication and computing systems evolve. This is driven by two trends. The first (and widely recognized trend) is the proliferation of data stream sources whose number and bandwidth increases as communication systems provide rapidly increasing bandwidths and connectivity that allows remote sources, such as sensors, to produce and emit data streams. The second, long-term, driver of the importance of efficient (XML) stream processing will be the growth of stream bandwidths at rates that surpass the growth of CPU-memory and CPU-disk access and transfer rates. It is already inefficient for many stream applications to buffer the data in disk-based buffers and analyze it later. At some point it may become impossible for high-bandwidth streams to be pushed even to the RAM memory – the CPU will have to quickly react to pieces of incoming data using just its cache memory.

A qualitatively different architecture is needed

* work partially supported by NSF/ITR OCE-0121726 RoadNet and NSF/NPACI ACI-9619020 Nara (Ludäscher), NSF/DGov-9983510 I2T (Ludäscher, Mukhopadhyay, Papakonstantinou), NSF/IDM-9734548 Career (Papakonstantinou)

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

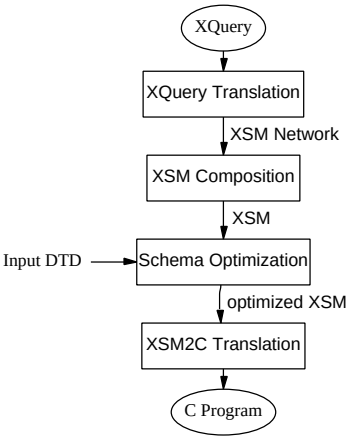


Figure 1: XQuery to XSM compilation

where stream processing is performed "on the fly" and with minimal memory: Computations are performed without storing data in secondary storage and by utilizing the limited size CPU cache memory as opposed to the RAM.[1] Each piece of the incoming stream will have to be consumed with just a few actions by the CPU. The simple, yet high bandwidth, sequential access that XML streams provide requires novel query processor architectures.

We propose the XSM-based architecture and algorithms for the construction of XML query (in particular, XQuery) processors that efficiently process XML streams on-the-fly. An *XSM* (*XML Stream Machine*) can be viewed abstractly as a state machine that can execute a few simple operations on the incoming data and its buffers. XQueries are compiled into XSMs, which are consequently translated into C programs, as shown in Figure 1.

The key challenge is in the XQuery2XSM Compiler, which has to construct an XSM for a given XQuery and optimize the produced XSM with respect to time and buffer usage, taking into account both the query and the schemas of the input streams. The various powerful XQuery constructs require that a systematic approach is taken towards building efficient XSMs. The following steps are followed: Each primitive subexpression e of the given XQuery q is translated into an XSM M_e. Then an XQuery expression is reduced to an XSM network where the output buffer of M_e is an input buffer for $M_{e'}$ if e is a subexpression of e'. In addition, the XSMs corresponding to **for** expressions of XQuery output into buffers that correspond to the variables defined by the **for** and those buffers are subsequently read by expressions that use the variables. An XSM network-based implementation corresponds to a conventional evaluation of XQuery. It requires multiple intermediate buffers, many of which correspond to subexpression results and variable bindings that may not have to be computed and stored. It also requires multiple XSMs to operate. Overall, it is inefficient from both time and memory perspectives. It is usually preferable to *compose* the XSM network into a single XSM which is optimized along the following dimensions:

- Minimize the computation performed for each incoming piece of stream data, *i.e.*, reduce the number of tests, read and write actions.[2]
- Minimize the number and size of buffers.
- Pipeline the computation and write tokens in the output stream as soon as possible.

We present an architecture and algorithms that achieve the above goals. The outline and main contributions of the paper are as follows: Section 2 presents the XML Stream Machine model. Section 3 describes XSM networks and the translation of XQueries into XSM networks. Notice that the essential non-recursive aspects of XQuery are captured by our translation. Section 4 describes XSM composition, which enables the reduction of XSM networks into a single XSM. The resulting XSM is optimized in a number of ways by a smart composition algorithm that essentially optimizes the connection between producer and consumer operators. By doing so, it manages to remove unnecessary tests and actions as well as redundant intermediate buffers. Finally, in Section 6, we present some experimental results illustrating the efficiency of the XSM approach.

2 XML Stream Machine Framework

In this section, after recalling some preliminaries on XML, we present the mechanics of XML Stream Machines (XSMs), an extension of conventional transducers designed to efficiently query and transform streams of XML data.

2.1 XML and XML Streams

In our XML model, we consider sets of *element names* $\mathcal{E}$ and *character data* (strings) $\mathcal{D}$.[3] Below, we will also consider XQuery expressions containing variables drawn from a set of *variable names* $\mathbb{V}$.

An *XML stream* is a sequence of tokens, where the set of *tokens* $\mathcal{T}$ is defined as:

$$\begin{aligned}\mathcal{T} &= \{\ \langle e\rangle \mid e \in \mathcal{E}\} \cup \{\ \mathsf{d}(x) \mid x \in \mathcal{D}\} \\ &\cup\ \{\ \langle /e\rangle \mid e \in \mathcal{E}\} \cup \{\ s_v, e_v \mid v \in \mathbb{V}\} \cup \{eol\}\ .\end{aligned}$$

[1] In many systems areas, such as networks, utilization of the limited CPU cache memory becomes an important goal.

[2] Constrast this approach with conventional query processors that build algebra-based evaluation plans that consist of multiple operators producing intermediate results [LPV00, FSW00, MP02, GVD+01]. Often these are constructed by one operator only to be discarded or reformatted by the next.

[3] We omit XML attributes to simplify the presentation. One can either add attributes to our approach directly, or assume they have been represented as subelements.

The *open*, *data*, and *close tokens*, $\langle e \rangle$, $\mathsf{d}(x)$, and $\langle /e \rangle$, respectively, correspond to the events that a parser like SAX encounters when processing an XML document. If an XML stream only comprises such tokens, we call it a *pure XML stream*. The *control tokens* s_v (e_v) indicate the start (end) of the binding of a variable v and can appear in internal buffers of an XSM or in shared buffers between XSMs and are used for coordinating actions. Similarly, the control token *eol* is used for grouping variable bindings; it marks the *end of list* of variable bindings belonging to the same group.

A *well-formed XML stream* is one in which every pure XML (sub-)stream which is delimited by control tokens is a well-formed XML fragment (defined as usual). In the sequel we only consider well-formed streams and fragments. Well-formed XML fragments correspond to *labeled ordered trees*, with labels of inner nodes indicating the element name of a node and leaf nodes representing character content.

XML Document Type Definitions (DTDs) are used to specify sets of valid documents via extended context-free grammars. DTD grammar rules are of the form *lhs* $\rightarrow$ *rhs*, where *lhs* is an element name in $\mathcal{E}$, and *rhs* is a regular expression over $\mathcal{E}$ and `#PCDATA`.[4] For example, the DTD

```
<!ELEMENT  root  (a)+ >
<!ELEMENT  a     (b)* >
<!ELEMENT  b     #PCDATA >
```

corresponds to the grammar $G =$

$$root \rightarrow a^+ \ , \quad a \rightarrow b^* \ , \quad b \rightarrow \texttt{\#PCDATA}$$

and validates XML documents where *root* elements contain one or more a elements, a elements contain zero or more b elements, and b elements contain character data.

For querying streams, it is reasonable to assume *acyclic* DTDs, *i.e.*, in which no element may contain a subelement of the same type. This implies that all valid XML streams have bounded depth and hence no stack is required to check well-formedness. Conversely, with cyclic DTDs, elements can be nested to any depth which on streams of arbitrary size is impractical and rules out efficient stream processing.[5] In the sequel, we only consider valid streams over acyclic DTDs. Such DTDs are equivalent to regular expressions: *e.g.*, the language $L(G)$ of documents valid with respect to G is the same as $L(R)$ where

$R = \langle root \rangle \ (\langle a \rangle \ (\langle b \rangle \ \texttt{\#PCDATA} \ \langle /b \rangle \ )^* \ \langle /a \rangle \ )^+ \ \langle /root \rangle$ is a regular expression. Observe that the XML trees defined through $L(R)$ have maximal depth four.

[4] The terminal symbol `#PCDATA` represents XML leaves holding character data ($=\mathcal{D}$).

[5] Alternatively, we could add stacks to our XSM framework to keep track of well-formedness and validity over cyclic DTDs. However, in our experience, real world DTDs (even when not confined to streaming applications) rarely require recursive element definitions.

$$Q\left\{\begin{array}{l}\overbrace{\textsf{for \$X in \$R/a}}^{E}\ \textsf{return}\\ \quad L\left\{\begin{array}{l}\overbrace{\textsf{for \$Y in \$X/b}}^{F}\ \textsf{return}\\ \quad \underbrace{\langle\textsf{res}\rangle\ \overbrace{\textsf{\$Y,\$X}}^{G}\ \langle/\textsf{res}\rangle}_{H}\end{array}\right.\end{array}\right.$$

Figure 2: Running example

2.2 XQuery and Running Example

We consider the following subset of XQuery expressions XQE, which includes the essential non-recursive aspects of the language [XQu01]:

$$\begin{array}{lr}
XQE ::= XQE_1 \ / \ Constant & \%\ Path\\
\quad | \ \ XQE_1 \ , \ XQE_2 & \%\ Concatenation\\
\quad | \ \ \langle Tag \rangle \ XQE_1 \ \langle / Tag \rangle & \%\ Element\ Creation\\
\quad | \ \ \textbf{for}\ Var\ \textbf{in}\ XQE_1 & \%\ Generator\\
\qquad [\textbf{where}\ Cond] & \%\ optional\ Condition\\
\qquad \textbf{return}\ XQE_2 & \%\ Body\\
\quad | \ \ Var &
\end{array}$$

$$\begin{array}{l}
Cond ::= \ \textbf{not}\ Cond_1\\
\qquad | \ \ Cond_1\ \textbf{and}\ Cond_2 \ \ | \ \ Cond_1\ \textbf{or}\ Cond_2\\
\qquad | \ \ Var_1 = Var_2 \ \ | \ \ Var = Constant
\end{array}$$

Example 1 (XQuery) Consider the XQuery in Figure 2. Subexpressions correspond to subqueries and have been marked with unique names ($E, F, \ldots, Q$). Given a binding for the variable $R, the overall query Q binds the outermost loop variable $X to each ⟨a⟩ child of $R in turn by means of the *path expression* E="$R/a". For each binding of $X, the inner loop L successively binds $Y to the ⟨b⟩ children of $X. For each such $Y binding, H outputs a new ⟨res⟩ element containing the $Y binding (a ⟨b⟩ element) followed by the $X binding (the enclosing ⟨a⟩ element). Thus, this XQuery *unnests* the ⟨b⟩ subelements from within the enclosing ⟨a⟩ elements. Note that each $X binding is output once for each contained $Y binding; the required *copying* of $X is achieved by pointer reset operations on the $X buffer (cf. the use of x0 in $M(L)$, Figure 5). □

We say that the variable V is *free* in an XQuery expression if it is not within the scope of a "**for** V **in** ...". For example, in Figure 2, $R is free in Q, $X is free in L, and $Y and $X are free in H.

We call *input variables* the variables that are free within the outermost XQuery Q, as they correspond to the *input streams* of the Q. In our example, the input variable $R corresponds to the only input stream. In a conventional, non-streaming application, the input variable $R has to be bound to a single root XML element. In general, a variable can be bound not only

to a single element but to a *sequence* of elements. To delimit the individual bindings of a variable V on a stream, we use the control tokens s_v and e_v.

2.3 XML Stream Machines

XML stream machines resemble traditional *transducers* [HU79] and translate one or more XML input streams into (usually) one output XML stream. Like transducers and finite state machines, they have finite sets of states Q and state transitions T. The latter are defined based on the content of the input, the current state, and some internal memory. In the case of XSMs, the memory is a finite set of buffers $B = B_1, \ldots, B_n$. Some buffers are distinguished *input buffers* and *output buffers*, which are associated with the *input streams* and *output streams* of the XSM, respectively.

For example, the XSM in Example 2 (Figure 3 and Figure 4) has two input buffers and thus input streams Y' and X', and produces an output stream Z. The individual components of a stream machine $M = (Q, q_0, B, T)$ are explained next.

XSM Buffers and Buffer Actions

Buffers are used to store parts of streams, *i.e.*, sequences of tokens. The size of a buffer depends on the query that the XSM evaluates, and the input on which it operates. Each buffer B_i has a set of associated *pointers* $ptr(B_i)$. A pointer p is either a *read pointer* or a *write pointer*. An input buffer only has read pointers, while an output buffer only has a single write pointer. All other buffers of an XSM are *working buffers*, each of which has has read pointers and a single write pointer.

In state transitions, XSMs can access and query buffer contents (via read operations such as "$*p$", see below), and execute sequences of *actions* $A = A_1, \ldots, A_k$. An *atomic action* A_i can have the form:

1. "p++": advance pointer p (*advance*)
2. "w(p, c)": at p, write c, then advance p (*write*)
3. "w$(p, *r)$": at p write $*r$, then advance p (*write*)
4. "$p' := p$": set p' to the position of p (*reset*)

In (2) and (3), p is a write pointer, r is a read pointer, and c is a token.

We assume that all action sequences are in *action normal form*, *i.e.*, no pointer increment action p++ is trailed by any non-increment action, so all increment actions occur in one block at the end of the sequence.[6]

[6] This normal form has to be used to guarantee the correctness of the optimized composition in Section 5.1.

XSM Control

An XSM has a finite number of states Q, one of which is the distinguished *initial state* q_0. When the XSM is in q_0, all pointers are on their leftmost (*i.e.*, empty) buffer positions. An XSM moves from the current state q to the next state q', provided there is a *transition* $t \in T$

$$t: \quad q \xrightarrow{\varphi|A} q'$$

whose *condition* φ is satisfied. Before entering q', the action sequence A is executed.

The *transition condition* φ is a boolean combination over the following atomic expressions:

- $p{=}p'$, $p{\neq}p'$, $p{<}p'$ (*pointer comparison*)
- $*r{=}c$, $*r{\neq}c$, $*r{=}*r'$, $*r{\neq}*r'$ (*token comparison*)

We require that XSMs are *deterministic* and *complete*, *i.e.*, conditions of outgoing transitions are *pairwise disjoint* and the disjunction of all such conditions is *valid*. The latter is achieved via a special sink state s_{err} into which an XSM moves if no explicitly mentioned transition is satisfied. To avoid clutter, we omit this state in our figures and all transitions into it. For example, Figure 3 hides the implicit error state and all its associated transitions such as $0 \xrightarrow{*y' \neq s_{y'}} s_{err}$.

Example 2 (Concat XSM) Consider the XSM $M(G)$ in Figure 3 with its input buffers Y' and X' (and associated read pointers y', x') and output buffer Z (with write pointer z). The XSM assumes Y' bindings to be delimited by $s_{y'}$ and $e_{y'}$ tokens, similar for X' bindings. In the initial state 0, $M(G)$ reads the current token $*y'$ from the Y' buffer and moves to state 1 if the expected start token $s_{y'}$ is found ($*y'{=}s_{y'}$). Otherwise, the machine moves to the implicit sink state s_{err}, indicating an error. The transition 1→2 is executed if $s_{x'}$ is read on the X' buffer ($*x'{=}s_{x'}$). At this point, s_z is written to the output buffer Z (w$(z, s_z,)$). In state 2, while the end of the Y' binding has not been reached ($*y'{\neq}e_{y'}$, denoted "*y'!=ey'" in the figures), Y' tokens are copied to Z. Once the Y' binding has been processed ($*y'{=}e_{y'}$) the machine continues in state 3 with copying X' content to Z until the end of the X binding is encountered. $M(G)$ returns to its initial state 0 after recognizing $*x'{=}e_{x'}$, *i.e.*, the end of the X' binding, and outputting e_z, which indicates that the concatenated result has been output to Z. Summarizing, whenever reaching 0, $M(G)$ has consumed exactly one Y' and one X' binding and has produced one Z binding. □

Observe that the XSM $M(G)$ not only works for single Y' and X' bindings, but for streams of such bindings, effectively computing the operation

$$zip: \ [x_1, x_2, ...], [y_1, y_2, ...] \mapsto [(y_1, x_1), (y_2, x_2), ...]$$

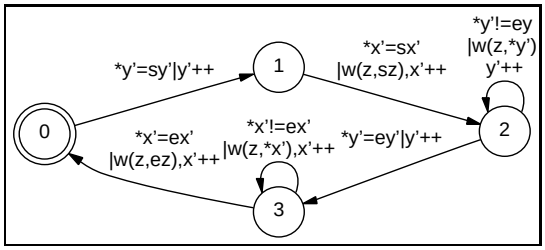


Figure 3: XSM $M(G) = \mathit{Concat}(\mathsf{Y}', \mathsf{X}', \mathsf{Z})$

For any finite input, this requires an equal number of X and Y bindings. Recall that XSMs are deterministic and thus compute *functions* from input streams to output streams. In particular, as illustrated with the previous example, XSMs map *prefixes* of input streams to *prefixes* of the output stream(s) and do *not* require the complete input to be available before producing output. Indeed, XSMs are designed to output as soon as possible, *i.e.*, in an *eager* mode and thus are well-suited for pipelining queries.

Comparison of XSMs with Transducer Models

There are some subtle but important points distinguishing the XSM framework from standard transducer models: Classical transducers work on finite alphabets, and their only memory is the state in which they are in. In contrast, XSMs process whole XML fragments built from the finite domain of element names $\mathcal{E}$, and the infinite domain of data values $\mathcal{D}$ (`#PCDATA`). Therefore, transitions cannot be specified in a finite manner by explicitly mentioning the input symbols. Instead, depending on the query being processed, XSMs store relevant "chunks" (the variable bindings) of input into buffers and use conditions such as above to specify transitions and actions. A simple yet powerful mechanism of accessing buffers of XSMs is through *pointers*, which are absent from conventional transducers. Another major difference is that XSMs can work on *multiple input streams*. For example, the *concat* XSM in Figure 3 processes and "zips" two input streams by concatenating pairs of bindings for Y and X, producing the the output stream Z.

3 Translation to XSM Networks

The XSM Compiler translates XQueries into optimized XSMs. Since XQuery is a non-trivial query language with numerous constructs, the translation is accomplished in a series of steps as depicted in Figure 1. The first step, presented in this section, is the translation of the input XQuery into an XSM *network*, which consists of individual XSMs corresponding to the subexpressions of the XQuery.

We first present XSM networks, followed by the translation of XQueries into such networks. The process is based on building buffers for subexpression results and variables, a "basic" XSM for each kind of XQuery subexpression, and appropriately connecting the buffers and XSMs. Notice that the "for" statement of XQuery and its ability to generate variables that are visible from its direct and indirect subexpressions complicates the XSM network structure.

XSM Networks

An *XSM network* is a directed acyclic graph (DAG), whose nodes are XSMs and whose labeled edges are of the form $M_1 \xrightarrow{B} M_2$ indicating that the output buffer of M_1 is the input buffer of M_2, *i.e.*, the buffer B is shared. M_1 is called the *producer*, M_2 the *consumer* XSM. An *input stream* I of a network is denoted by an edge of the form $\mathsf{in} \xrightarrow{I} M$ (so I is an input buffer of M). Similarly, an *output stream* of a network is denoted in the form $M \xrightarrow{O} \mathsf{out}$, where O is an output buffer of M.

Example 3 (XSM Network) Figure 4 illustrates the XSM network for our running example of Figure 2. It has a single input stream corresponding to the input buffer R of $M(E)$ and a single output stream, corresponding to the output buffer O of $M(H)$. □

XSMs compute functions on streams by mapping prefixes of the input streams (partial inputs) to prefixes of the output streams (partial outputs). Based on this, computation in a network with input streams $I_1, \ldots, I_k$ proceeds as follows. There is a list $\mathcal{B}$ of "computed buffers", which is initially set to $\{I_1, \ldots, I_n\}$. At each step an XSM M is chosen whose input buffers are in $\mathcal{B}$. We run M to compute a (partial) output for its output buffer B and we add B to $\mathcal{B}$. We continue with all M's until a (partial) output for O is computed. The DAG structure of the network guarantees that eventually O is computed.

Translation Algorithm

Let us first explain the relationship between the network's buffers and the variables and subexpressions of the given XQuery. We associate each input variable I with a corresponding input buffer named I. Then, for every *path*, *concatenation*, and *element creation* (sub)expression Q (*i.e.*, every subexpression other than a for expression or a single variable) we create a buffer named $out(Q)$, which will store the output results of the subexpression.

Which buffers are created for a for expression

$$F = \textbf{for } V \textbf{ in } Q_1 \textbf{ return } Q_2$$

depends on the *free variables* in the body Q_2 of F.

In the first case, the body Q_2 contains no free variables other than the loop variable V. For example, the outermost for expression Q in Figure 2 meets this condition, since the only free variable in its body L is the loop variable $X itself. In such cases, we only

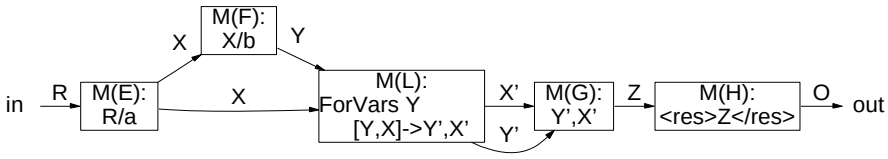


Figure 4: XSM network of running example XQuery

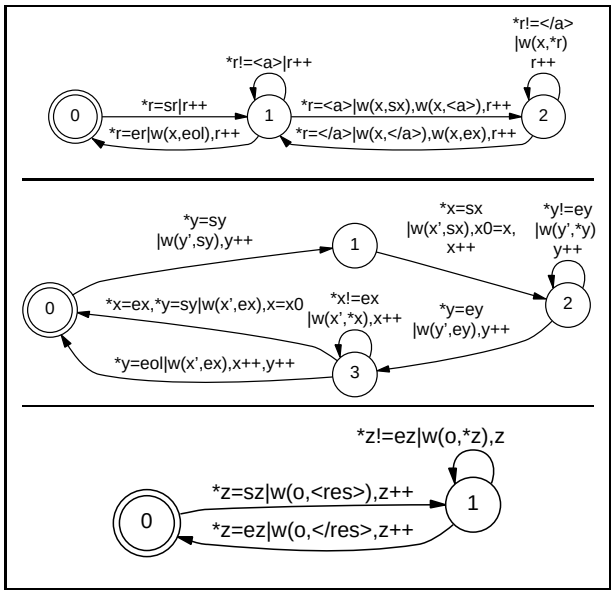


Figure 5: XSM template instances (top to bottom): $M(E)$=*Path*(R,a,X), $M(L)$=*ForVars*(Y,[Y,X],[Y',X']), $M(H)$=*CreateEl*(Z,res,O)

need to create two buffer name *aliases* (without actually creating new buffers), corresponding to the fact that those buffers are shared: First, we have that $out(Q_1) \equiv V$, *i.e.*, the output buffer $out(Q_1)$ of the generator Q_1 is named after the loop variable V. In the running example, $out(E) \equiv \$X$ (see Figures 2,4). Second, $out(F) \equiv out(Q_2)$, *i.e.*, the output buffer of the for is the output buffer of the body Q_2. In the running example, this means that the result buffer $out(Q)$ of the overall XQuery Q is the same as $out(L)$, the output buffer of Q's body L.

In the more complex second case, the body of the for expression F contains free variables *other* than the loop variable. Such is the case with the for expression L, whose body H contains the free variable $X, which is *not* the loop variable of L. As in the previous case, we declare that $out(Q_1) \equiv V$ and $out(F) \equiv out(Q_2)$. In addition, for every *free variable* U in the body of F we produce a new buffer named U'. Consequently, any reference to the variable U in the body of F will be meant to refer to the buffer U', unless it is within the scope of a nested for subexpression in the body or the generator of F.

For translating an XQuery into an XSM network, we use the following XSM *templates*:

- *Path*(*InBuf*, *ChildTag*, *OutBuf*)
- *Concat*(*InBuf1*, *InBuf2*, *OutBuf*)
- *CreateEl*(*InBuf*, *ElemTag*, *OutBuf*)
- *ForVars*(*InVar*, [*InVars*], [*OutVars*])

The *ChildTag* and *ElemTag* parameters have to be instantiated with constants, *InBuf*, *OutBuf*, *InVar* with buffer names, and *InVars*, *OutVars* with lists of buffer names.

We are now ready to produce XSM networks that use the buffers described above. For every subexpression Q of the given XQuery:

---

if $Q = Var$ **then**
  `/* skip for variables */`
else if $Q =$ "Q_1/c" **then**
  produce *Path*$(out(Q_1), c, out(Q))$
  `/* e.g. Fig.5:` *Path*(R,a,X) `*/`
else if $Q =$ "Q_1 , Q_2" **then**
  Concat$(out(Q_1), out(Q_2), out(Q))$
  `/* e.g. Fig.3:` *Concat*(Y',X',Z) `*/`
else if $Q =$ "$\langle e \rangle Q_1 \langle /e \rangle$" **then**
  CreateEl$(out(Q_1), e, out(Q))$
  `/* e.g. Fig.5:` *CreateEl*(Z,res,O) `*/`
else if $Q =$ "**for** Var **in** Q_1 **return** Q_2" and $free(Q_2) \setminus \{V\} \neq \emptyset$ **then**
  $InVars := free(Q_2)$;
  $OutVars' := \{V' \mid V \in InVars\}$;
  produce *ForVars*$(V, [InVars], [OutVars'])$
  `/* e.g. Fig.5:` *ForVars*(Y, [Y,X], [Y',X']) `*/`

---

4 XSM Composition

In XSM networks, consecutive XSMs are linked via buffers. For example, consider two XSMs that are linked via a shared buffer B^s: $M_1 \xrightarrow{B^s} M_2$. XSM composition allows us to replace M_1 and M_2 with a single XSM $M_3 = (M_2 \circ M_1)$. The composition creates opportunities for eliminating the need for the shared buffer B^s and for optimizing the composed XSM (see below).

For a state q, let $readPtr(q)$ denote the set of read pointers on which any outgoing transition $t : q \xrightarrow{\varphi|A} q'$ depends, *i.e.*, $readPtr(q)$ contains all read pointers occurring in the condition φ or action A. Then $scPtr(q)$ is the subset of $readPtr(q)$ which point into the shared connection buffers B^s:

$$scPtr(q) := ptr(B^s) \cap readPtr(q)$$

Input
• producer XSM $M_1 = (Q^1, q_0^1, B^1, T^1)$,
• consumer XSM $M_2 = (Q^2, q_0^2, B^2, T^2)$,
• shared connection buffers $B^s = B^1 \cap B^2$
Output
• composed XSM $M_3 = (Q^3, q_0^3, B^3, T^3)$

begin
$Q^3 := Q^1 \times Q^2$; $q_0^3 := (q_0^1, q_0^2)$; (0)
$B^3 := B^1 \cup B^2$; $T^3 := \emptyset$;

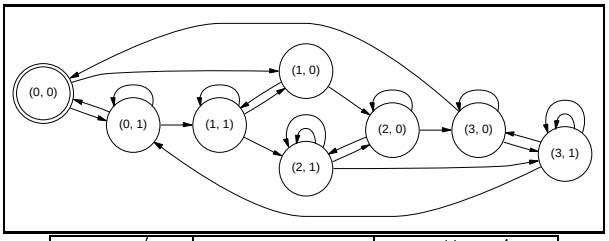
for $(q_1 \xrightarrow{\varphi_1|A_1} q_1') \in T^1$, $(q_2 \xrightarrow{\varphi_2|A_2} q_2') \in T^2$ **do**
  if $scPtr(q_2) = \emptyset$ **then**
    $add(T^3, \{(q_1, q_2) \xrightarrow{\varphi_2|A_2} (q_1, q_2')\})$ (1)
  else
    $\psi := \bigwedge_{r \in scPtr(q_2)} \neg\mathsf{AE}(r)$;
    $add(T^3, \{(q_1, q_2) \xrightarrow{\psi \wedge \varphi_2|A_2} (q_1, q_2')\})$ (2)
    $add(T^3, \{(q_1, q_2) \xrightarrow{\neg\psi \wedge \varphi_1|A_1} (q_1', q_2)\})$ (3)
end

Figure 6: Basic XSM composition algorithm

Note that we suppress the parameter B^s of $scPtr()$ since it will be clear from the context.

In the sequel, we present two composition algorithms that allow us to replace any network of XSMs by a single XSM. Consider first the algorithm of Figure 6. It takes a *producer* XSM M_1 and a *consumer* XSM M_2 linked through shared buffers B^s. The states of the composed XSM M_3 are given as the product of the states of M_1 and M_2 (line 0). Intuitively, if M_1 is in state q_1 and M_2 in q_2, then M_3 is in (q_1, q_2). In a sense, M_3 executes both, transitions from M_1 and M_2. Since M_2 produces the output of the composition, we execute its write actions as soon as possible, thereby implementing an *eager* output strategy.

(1–3) determine the transitions of M_3: For each pair of transitions $(t_1, t_2) \in T^1 \times T^2$, following the eager strategy, we do a transition t_2 in the consumer M_2 if possible. Determining whether the transition is possible depends on $scPtr(q_2)$, *i.e.*, the set of read pointers into the shared buffers B^s on which the transition t_2 of M_2 depends. If $scPtr(q_2)$ is empty, M_2 can move to the subsequent state q_2' directly (provided φ_2 is satisfied and the actions A_2 are executed), since t_2 is independent of the shared buffer and, hence, of t_1 (line 1). However, if there are read pointers $r \in scPtr(q_2)$, *i.e.*, pointers required by t_2 and which point to buffers written by M_1, we can execute t_2 only if those pointers are not at the end of their associated buffer. For this we use the pointer comparison $\mathsf{AE}(r)$ ("At-End r"), which is a runtime check "r=wp" comparing the position of r in B^s with the position of its associated write pointer wp (=$writePtr(buffer(r))$ from M_1. Note that each buffer has exactly one write pointer. If $\mathsf{AE}(r)$ evaluates to *false*, then no relevant read pointer of M_2 has reached its end of the buffer and, similar to (1), we can make a step in M_2 (2). Otherwise, the M_2 step t_2 cannot be executed in q_2 and we proceed with a t_1 step (3).



$q \to q'$	φ	*actions* A
$(0,0)\to(0,1)$	$\neg\mathsf{AE}(z')\wedge *z'=s_z$	$\mathsf{w}(o,\langle res\rangle), z'$++
$(0,0)\to(1,0)$	$\mathsf{AE}(z')\wedge *y'=s_y$	y'++
$(2,1)\to(2,1)$	$\neg\mathsf{AE}(z')\wedge *z'\neq e_z$	$\mathsf{w}(o,*z'), z'$++
$(2,1)\to(2,1)$	$\mathsf{AE}(z')\wedge *y'\neq e_y$	$\mathsf{w}(z,*y'), y'$++
$(2,1)\to(2,0)$	$\neg\mathsf{AE}(z')\wedge *z'=e_z$	$\mathsf{w}(o,\langle/res\rangle), z'$++
$(2,1)\to(3,1)$	$\mathsf{AE}(z')\wedge *y'=e_y$	y++
$(2,0)\to(2,1)$	$\neg\mathsf{AE}(z')\wedge *z'=s_z$	$\mathsf{w}(o,\langle res\rangle), z'$++
$(2,0)\to(2,0)$	$\mathsf{AE}(z')\wedge *y'\neq e_y$	$\mathsf{w}(z,*y'), y'$++
$(2,0)\to(3,0)$	$\mathsf{AE}(z')\wedge *y'=e_y$	y++
$(1,1)\to(2,1)$	$\mathsf{AE}(z')\wedge *x'=s_x$	$\mathsf{w}(s_z,z)$
$(1,1)\to(1,1)$	$\neg\mathsf{AE}(z')\wedge *z'\neq e_z$	$\mathsf{w}(o,*z'), z'$++
$(1,1)\to(1,0)$	$\neg\mathsf{AE}(z')\wedge *z'=e_z$	$\mathsf{w}(o,\langle/res\rangle), z'$++
$(3,0)\to(0,0)$	$\mathsf{AE}(z')\wedge *x'=e_x$	$\mathsf{w}(z,e_z), x'$++
$(3,0)\to(3,0)$	$\mathsf{AE}(z')\wedge *x'\neq e_x$	$\mathsf{w}(z,*x'), x'$++
$(3,0)\to(3,1)$	$\neg\mathsf{AE}(z')\wedge *z'=s_z$	$\mathsf{w}(o,\langle res\rangle), z'$++
$(0,1)\to(0,0)$	$\neg\mathsf{AE}(z')\wedge *z'=e_z$	$\mathsf{w}(o,\langle/res\rangle), z'$++
$(0,1)\to(1,1)$	$\mathsf{AE}(z')\wedge *y'=s_y$	y'++
$(0,1)\to(0,1)$	$\neg\mathsf{AE}(z')\wedge *z'\neq e_z$	$\mathsf{w}(o,*z'), z'$++
$(1,0)\to(2,0)$	$\mathsf{AE}(z')\wedge *x'=s_x$	$\mathsf{w}(s_z,z)$
$(1,0)\to(1,1)$	$\neg\mathsf{AE}(z')\wedge *z'=s_z$	$\mathsf{w}(o,\langle res\rangle), z'$++
$(3,1)\to(3,0)$	$\neg\mathsf{AE}(z')\wedge *z'=e_z$	$\mathsf{w}(o,\langle/res\rangle), z'$++
$(3,1)\to(0,1)$	$\mathsf{AE}(z')\wedge *x'=e_x$	$\mathsf{w}(z,e_z), x'$++
$(3,1)\to(3,1)$	$\neg\mathsf{AE}(z')\wedge *z'\neq e_z$	$\mathsf{w}(o,*z'), z'$++
$(3,1)\to(3,1)$	$\mathsf{AE}(z')\wedge *x'\neq e_x$	$\mathsf{w}(z,*x'), x'$++

Figure 7: Result of basic composition $M(H) \circ M(G)$

One can show that each sequence of state transitions of the composed XSM M_3 on any input stream I corresponds to valid sequences of state transitions of the network $(M_2 \circ M_1)(I)$ and produces the same output, so the above composition algorithm is correct. Since XSM networks are acyclic, we can repeatedly compose adjacent XSMs until there is a single XSM left.

Example 4 (Basic Composition)
Figure 7 depicts the machine resulting from applying the basic composition algorithm on the XSMs $M(H)$ and $M(G)$ of Figure 5. □

5 Optimizations

While the composition algorithm of Figure 6 is simple and elegant, the efficiency of the resulting XSM can be improved in several ways.

5.1 Lockstep Optimization

First, we exploit the fact that the basic algorithm introduces runtime checks $\mathsf{AE}(p)$ which can be shown to be valid or unsatisfiable using a static analysis technique. The basic idea is to statically analyze when the

producer M_1 and the consumer M_2 operate in "lockstep" on the shared connection buffer, *i.e.*, when a read pointer r is trailing its associated write pointer wp by at most one position.[7] In such cases, the optimized composition can eliminate AE checks. However, note that some checks are inherently runtime and data dependent and thus cannot be decided or "optimized away" by any compile-time analysis. When r and wp are in lockstep, we do not need the buffer as an intermediate storage. Consequently, we are able to eliminate tests, actions, and often the whole buffer.

The input and output of the optimized composition algorithm in Figure 8 are as before. However, unlike the basic algorithm which can always be applied, we can use the improved version only if no state $q_2 \in Q^2$ depends on two different read pointers into a shared connection buffer. Thus, the optimized composition algorithm in Figure 8 has the following input/output, precondition, and initialization steps:

Input
- producer XSM $M_1 = (Q^1, q_0^1, B^1, T^1)$
- consumer XSM $M_2 = (Q^2, q_0^2, B^2, T^2)$
- shared connection buffers $B^s = B^1 \cap B^2$

Precondition
for all $q_2 \in Q^2 : |scPtr(q_2)| \leq 1$

Output
- optimized composed XSM $M_3 = (Q^3, q_0^3, B^3, T^3)$

Initialization
$Q^3 := Q^1 \times Q^2 \times \{go, no, ae\}$;
$q_0^3 := (q_0^1, q_0^2, no)$; $B^3 := B^1 \cup B^2$; $T^3 := \emptyset$;

The states Q^3 of the optimized composed XSM M_3 have Q^1 and Q^2 components of M_1 and M_2, respectively, and can be in one of three modes: go, no, ae.

In a state $(q_1, q_2, go) \in Q^3$, the consumer M_2 can go forward eagerly and execute a transition t_2 since any read pointer r on which t_2 may depend is known to be trailing its associated write pointer wp by exactly one step. Thus AE(r) will be *false* and can be eliminated; cf. (GO) in Figure 8: if q_2 has exactly one read pointer r into the shared buffer and if r is advanced by M_2, then we execute t_2 but go into *no* mode. Otherwise (note the precondition of the algorithm), we know that $scPtr(q_2) = \emptyset$ and we can remain in *go* mode after executing t_2.

In states of the form (q_1, q_2, no) (*e.g.*, the initial state q_0^3 of M_3 is of this form), we know that the read pointer r has reached its write pointer wp (*i.e.*, AE(r) is true) and we usually cannot go forward with M_2 but have to execute a transition t_1 of M_1 first (NO). The only exception is when $scPtr(q_2) \neq \{r\}$ hence q_2 does not depend on any read pointer into B^s. In this case, we can stay in *no* mode and just execute a M_2 step. In the main case, $scPtr(q_2) = \{r\}$ and several simplifications can be applied, since we know that what

[7] Recall that we are interested in moving M_2 as soon as possible to achieve eagerness.

for $(q_1 \xrightarrow{\varphi_1|A_1} q_1') \in T^1, \quad (q_2 \xrightarrow{\varphi_2|A_2} q_2') \in T^2$ **do**
  `/* handle` $(q_1, q_2, go) \in Q^3$ `*/` (GO)
  if $scPtr(q_2) = \{r\}$ and r`++` $\in A_2$ **then**
    $add(T^3, \{(q_1, q_2, go) \xrightarrow{\varphi_2|A_2} (q_1, q_2', no)\})$
  else
    $add(T^3, \{(q_1, q_2, go) \xrightarrow{\varphi_2|A_2} (q_1, q_2', go)\})$
  `/* handle` $(q_1, q_2, no) \in Q^3$ `*/` (NO)
  if $scPtr(q_2) = \{r\}$ **then**
    $wp := writePtr(buffer(r))$;
    if $\{\mathsf{w}(wp, \_) \in A_1\} = \{\mathsf{w}(wp, X)\}$ **then**
      $(\varphi_{12}, A_{12}, A_{12}') := \mathsf{simplify}((\varphi_1 \wedge \varphi_2 | A_1; A_2), *r{=}X)$;
      if r`++` $\in A_2$ **then**
        $add(T^3, \{(q_1, q_2, no) \xrightarrow{\varphi_{12}|A_{12}'} (q_1', q_2', no)\})$
      else
        $add(T^3, \{(q_1, q_2, no) \xrightarrow{\varphi_{12}|A_{12}} (q_1', q_2', go)\})$
    else if $\{\mathsf{w}(wp, \_) \in A_1\} = \emptyset$ **then**
      $add(T^3, \{(q_1, q_2, no) \xrightarrow{\varphi_1|A_1} (q_1', q_2, no)\}$
    else
      $add(T^3, \{(q_1, q_2, no) \xrightarrow{\varphi_1|A_1} (q_1', q_2, ae)\})$
  else
    $add(T^3, \{(q_1, q_2, no) \xrightarrow{\varphi_2|A_2} (q_1, q_2', no)\})$
  `/* handle` $(q_1, q_2, ae) \in Q^3$ `*/` (AE)
  if $scPtr(q_2) = \{r\}$ **then**
    $add(T^3, \{(q_1, q_2, ae) \xrightarrow{\neg \mathsf{AE}(r) \wedge \varphi_2|A_2} (q_1, q_2', ae)\})$;
    $wp := writePtr(buffer(r))$;
    if $|\{\mathsf{w}(wp, \_) \in A_1\}| = 1$ **then**
      $add(T^3, \{(q_1, q_2, ae) \xrightarrow{\mathsf{AE}(r) \wedge \varphi_1|A_1} (q_1', q_2, go)\})$
    else
      if $\{\mathsf{w}(wp, \_) \in A_1\} = \emptyset$ **then**
        $add(T^3, \{(q_1, q_2, ae) \xrightarrow{\mathsf{AE}(r) \wedge \varphi_1|A_1} (q_1', q_2, no)\})$
      else
        $add(T^3, \{(q_1, q_2, ae) \xrightarrow{\mathsf{AE}(r) \wedge \varphi_1|A_1} (q_1', q_2, ae)\})$
  else
    $add(T^3, \{(q_1, q_2, ae) \xrightarrow{\varphi_2|A_2} (q_1, q_2', ae)\})$
$Q^3 := Q^3 \setminus \{unreachable\ states\}$;
$B^3 := B^3 \setminus \{unused\ buffers\}$

Figure 8: Optimized composition algorithm

is written by M_1 is consumed by M_2 immediately: If there is exactly one M_1 action A_1 that writes some X to the write pointer wp that M_2 is reading from via r, then we can do both a M_1 step and a M_2 step. Moreover, we can simplify the conjunction $\varphi_1 \wedge \varphi_2$ of the composed condition as well as the sequence of actions $A_1; A_2$ using the equality $*r = X$:

The simplified condition φ_{12} is obtained by applying the substitution $*r \mapsto X$ and reducing the resulting expression as much as possible using well-known logic simplifications. In particular, if φ_{12} can be reduced to *false*, then the transition can be removed. The simplified A_{12} is similarly obtained by substituting X for $*r$ and simplifying if possible. In addition, all pointer increment actions of A_1 are moved after any non-increment action of A_2 and just before the increment actions of A_2, in order to obtain the *action normal form* (Section 2.3) which guarantees the correctness of the composition and simplication. A_{12}' is

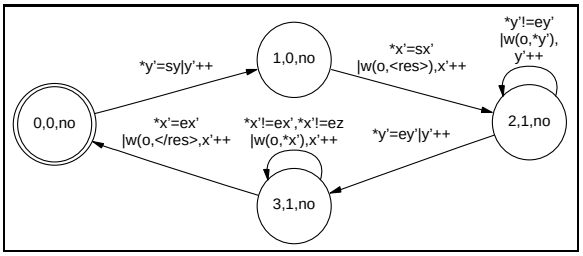


Figure 9: Optimized composed XSM $M(H) \circ M(G)$

like A_{12} but with the actions $\mathsf{w}(wp, X)$ and r++ removed. If A_1 does not contain a write action, then we only move M_1 and stay in *no* mode. If instead A_1 contains more than one write via wp then we have "lost lockstep" and go into ae mode.

Finally, the (AE) block in Figure 8 handles transitions emanating from $(q_1, q_2, ae) \in Q^3$, *i.e.*, which may require runtime checks $\mathsf{AE}(r)$. Note that there are situations where we "fall back" into lockstep, *i.e.*, either *no* or *go* mode, depending on whether we have zero or one write actions on wp, and under the proviso that $\mathsf{AE}(r)$ is detected at runtime.

Example 5 (Optimized Composition)
Figure 9 depicts the XSM resulting from applying the optimized composition on $M(H)$ and $M(G)$.

Figure 11 shows the final result of successive application of the optimized composition on the XSM network in Figure 4, *i.e.*, the XSM corresponding to our running example XQuery Q. □

5.2 Schema-Based Optimization

If the XML schema of the input stream is known, further optimizations are possible. For example, consider the XSM $M(E)$=*Path*(R,a,X) in Figure 5. If we knew that on the input stream R only $\langle \mathsf{a} \rangle$ elements can appear, we could simplify the XSM further. In particular, the transition corresponding to the self-loop in state 1 can be eliminated since it is known that $*r \neq \langle \mathsf{a} \rangle$ will always be *false*. Similarly, test for control events s_v, e_v marking the beginning and end of a variable binding of v can be eliminated using schema information, appropriately augmented with information regarding start/end variable tag symbols.

XML schema information is incorporated into our XSM framework as follows. Given the input schema as an XML DTD, we first use a type inference algorithm [XQu02, PV00] to infer the types of variables used in the input XQuery Q. These variable types directly yield the *buffer types* of the XSM network equivalent to Q. We can denote the type of a variable v (and thus of its associated buffer v) by an *augmented DTD*, *i.e.*, a DTD whose root element is a variable name. We then interpret the open and close tags $\langle v \rangle$ and $\langle /v \rangle$ as the control events s_v and e_v, respectively.

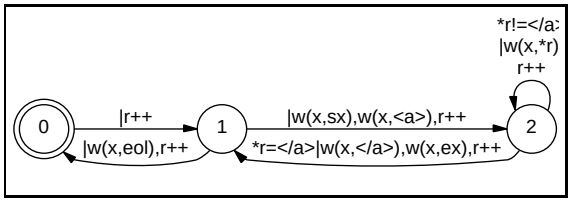


Figure 10: $M(E)$=*Path*(R,a,X) after schema-based optimization with $\mathrm{DTD}(R) = \{R \to a^*, \ldots\}$.

Example 6 (Augmented DTD)
Consider the XQuery in Figure 2 and its XSM network. Given the input DTD:

$$\{root \to a^*,\ a \to b^*,\ b \to \mathtt{PCDATA}\}\ ,$$

and using a standard type inference for variables, we obtain the following augmented DTDs:
$\mathrm{DTD}(R) = \{R \to a^*,\ a \to b,\ \ldots\}$,
$\mathrm{DTD}(X) = \{X \to a,\ a \to b,\ \ldots\}$,
$\mathrm{DTD}(Y) = \{Y \to b,\ \ldots\}$, etc.
By interpreting tags of variables as control events, we obtain the desired schema information on buffer contents. For example, for the R buffer we obtain $s_r \langle \mathsf{a} \rangle \ldots \langle /\mathsf{a} \rangle \cdots \langle \mathsf{a} \rangle \ldots \langle /\mathsf{a} \rangle\, e_r$, similarly, for the X buffer we get $s_x \langle \mathsf{a} \rangle \ldots \langle /\mathsf{a} \rangle\, e_x$, etc. □

Using standard type inference of XQuery variables and augmented DTDs for XSM buffers, we can further simplify and thus optimize XSMs.

Example 7 (Schema-Based Optimization)
Consider the XQuery Q and the augmented DTDs from the previous example. Figure 10 shows an optimized version of $M(E)$ from Figure 5. Note that the loop transition t: $1 \to 1$ has been eliminated since its condition $*r \neq \langle \mathsf{a} \rangle$ is equivalent to *false* for the given $\mathrm{DTD}(R)$ having only $\langle \mathsf{a} \rangle$ elements. Similarly, (parts of) conditions that are always *true* have been eliminated. □

Schema-based optimization not only applies to basic XSMs such as instances of the Path template, but also to XSMs resulting from composition or (lockstep) optimized composition.

Example 8 (Schema-Based Optimization)
The transition $(3, 1, no) \to (3, 1, no)$ of the optimized XSM in Figure 9 has the condition $*x' \neq e'_x \wedge *x' \neq e'_z$. Using $\mathrm{DTD}(X)$ we can simplify this to $*x' \neq e'_x$ since e_z cannot occur in an X buffer. □

6 Experimental Results

The output of the XSM compiler is a C program which uses a SAX parser on the incoming XML stream. We measured the performance of our XSM-based query processing engine and compared it to several publicly available XSLT engines by running the following query

on the DBLP XML database. We report below the results for the query Q_{trans} that transforms an incoming XML stream into an (X)HTML stream: For each conference paper, an HTML table row is generated, listing the title of the paper, followed by a *nested table* listing all authors of that paper. Q_{trans} is:

```
<html>
 <table>
  for $X in doc(root)/dblp/inproceedings
  return
    <tr><td> $X/title </td>
        <td>
          <table>
            for $Y in $X/author return
              <tr><td> $Y </td></tr>
          </table>
        </td>
    </tr>
 </table>
</html>
```

Our experiments were carried out on a workstation with an Intel Xeon 2.2GHz processor with 1GB of RAM, running MS Windows 2000 Professional, after warming up the filesystem cache to minimize interference from the disk and the OS virtual memory manager. Sun JDK version 1.3 was used as the Java compiler/JVM. The C compiler was gcc run under the cygwin tool suite. For parsing the XML documents the expat C library and the Apache tools for Java were used. We used varying sizes of XML input streams from the well-known DBLP database (80MB). The results of the experiments are summarized in Table 1.

Discussion

In general, we observed that the XSM-based implementations are significantly faster for simple transformations, and the time spent in the process grows linearly with the document size for queries with linear data complexity. In contrast, there is a size beyond which the time taken by the XSLT-based tools begins to grow superlinearly, *i.e.*, the tested XSLT engines do not scale to very large files. The results offer some insight into the problems of currently available transformation engines based on XSLT. A key reason that their performance degrades and does not attain linear complexity is memory management. It seems that they do not detect that Q_{trans} can process one `inproceedings` record at a time. Instead the memory management issue is delegated to Java's garbage collection, which is strained.

In contrast, our approach of applying optimizations locally and minimizing the number of buffers in a transducer framework leads to programs which scale very well on certain classes of "streamable queries", *i.e.*, queries that exhibit a local transformation behavior. Our experiments show that for such queries our translation algorithm coupled with the optimizing composition and the schema-based optimizations leads to highly efficient throughput. Note that even the limited class of XQueries we consider is sufficiently more complex than XPath queries, which are the typical topic of many XML streaming and filtering applications [AF00, Oni01, ILW00, DFFT02].

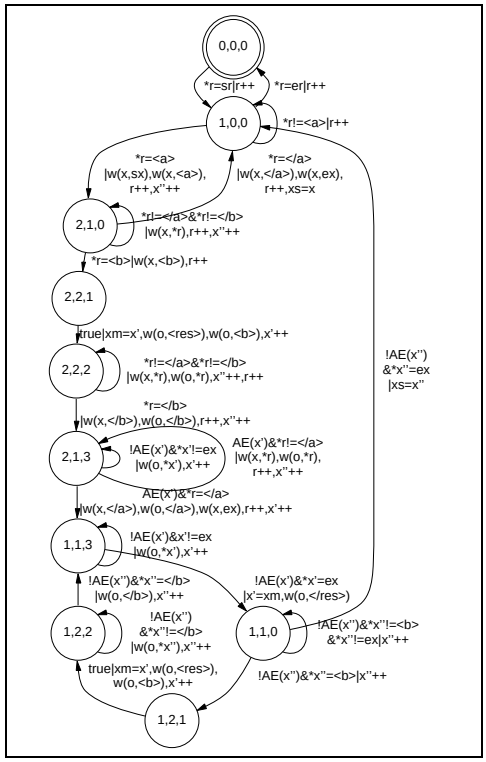


Figure 11: Optimized XSM for XQuery Q

7 Related Work

The availability of large amounts of data that are produced by sensors, coupled with a need to analyze this data in real-time, has led to research on the various ways in which the streaming nature of the data sources affects the traditional query evaluation approaches. Most of the work has focused on simple relational data streams and has addressed issues in efficient query processing and memory consumption. Many of the query processing works have focused on join queries. At the plan execution level, pipelined operators have been developed for join operations [WA91, HH99, UF00, LHEN02]. At the optimization level, new cost models for relational optimizers have been developed that account for the fact that the data comes from streams rather than from disks [VN02]. In addition, techniques have been developed to account for changes in data arrival rates, as well as the fact that these costs may change during the evaluation of a plan. This has led to the development of adaptive query evaluation plans [AH00, MSH02, AFTU96]. The

Query	Stream Size	Xalan time (in ms)	SAXON time (in ms)	XSLTC time (in ms)	XSM Java time (in ms)	XSM C time (in ms)
Q_{trans}	4KB	663	703	708	266	30
	5MB	7031	7070	7081	2360	312
	10MB	23200	22900	22950	4375	594
	20MB	102710	102100	102400	8266	1156
	80MB	∞	∞	∞	32078	4640

Table 1: Experimental Results: experiments marked ∞ timed out or failed (memory violation)

issue of join/selection order in continuous queries over streams is discussed in [CDN02].

The optimization of aggregation queries has also attracted significant attention. Efficient algorithms and data structures for maintaining aggregates have been developed, both for the case of the entire stream and the case of a sliding window over the stream [MRL98, DGIM02]. The notion of correlated aggregate queries (*i.e.*, queries computing the aggregated value of an attribute where there is a selection condition involving the aggregate value of another attribute) is introduced in [DGGR02], and techniques for the computation of approximate results to correlated aggregate queries over a single relational data stream are discussed. Techniques for computing approximate answers to aggregate queries involving joins are presented in [GKS01]. The ideas of efficient stream processing for joins and aggregates can be adopted in subsequent versions of the XSM.

The issue of memory requirements of relational stream queries has been studied in various contexts. The class of conjunctive select-project-join (SPJ) queries with arithmetic comparisons that can be evaluated in constant memory is identified in [ABB+02]. The widespread use of XML as data model has led to a large body of work on XML-based information integration and query processing, *e.g.*, [FMS01, SSB+00].

On the issue of XML stream processing, the fast evaluation of regular expressions using a pipelined (SAX based) approach is considered in [ILW00] and [GMOS02]. In [OMFB02] equivalences of XPath location paths are used to rewrite queries with reverse axes into reverse-axis-free ones, thereby enabling a more efficient SAX based stream processing. [VS02] considers the problem of efficiently validating XML documents against DTDs.

The key contribution of our work is the introduction of an extended state transducer model for processing XML streams. Transducers have been used earlier by the database community for type checking and type inference [PV00]. However, it is the first time that transducers become effectively the execution plan of the query. In previous work on XML query processing, the plan has been typically based on algebras having operators that extend the relational algebra with operators specifically suited to XML query processing [VGD+02, JLST01, LPV00]. Other work in compiling XML transformations directly to programs exists in the context of XSLT [XSL02]. However, to the best of our knowledge, the XSLT community has not developed ways to minimize the memory and number of operations of an XSLT program.

References

[ABB+02] A. Arvind, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing Memory Requirements for Queries over Continuous Data Streams. In *ACM PODS*, 2002.

[AF00] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *VLDB*, 2000.

[AFTU96] L. Amsaleg, M. Franklin, A. Tomasic, and T. Urhan. Scrambling Query Plans to Cope with Unexpected Delays. In *PDIS*, 1996.

[AH00] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *ACM SIGMOD*, 2000.

[CDN02] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In *ICDE*, 2002.

[CFGR02] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *ICDE*, 2002.

[DFFT02] Y. Diao, P. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and Scalable Filtering of XML Documents (demo description). In *ICDE*, 2002.

[DGGR02] A. Dobra, M. Garofalakis, J. E. Gehrke, and R. Rastogi. Processing Complex Aggregate Queries over Data Streams. In *ACM SIGMOD*, 2002.

[DGIM02] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining Stream Statistics over Sliding Windows. In *Annual ACM-SIAM Symp. on Discrete Algorithms (SODA 2002)*, 2002.

[FMS01] M. F. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middle-ware queries. In *ACM SIGMOD*, May 2001.

[FSW00] M. Fernandez, J. Simeon, and P. Wadler. An Algebra for XML Query. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Delhi, India, 2000.

[GKS01] J. Gehrke, F. Korn, and D. Srivastava. On Computing Correlated Aggregates Over Continual Data Streams. In *ACM SIGMOD*, 2001.

[GMOS02] T. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata and Stream Indexes, 2002. `http://www.cs.washington.edu/homes/suciu`.

[GVD+01] L. Galanis, E. Viglas, D. J. DeWitt, J. F. Naughton, and D. Maier. Following the Paths of XML Data: An Algebraic Framework for XML Query Evaluation. `http://www.cs.wisc.edu/niagara/papers/algebra.pdf`, 2001.

[HH99] J. Hellerstein and P. J. Haas. Ripple Joins for Online Aggregation. In *ACM SIGMOD*, 1999.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[ILW00] Z. G. Ives, A. Y. Levy, and D. S. Weld. Efficient Evaluation of Regular Path Expressions on Streaming XML Data. TR UW-CSE-2000-05-02, University of Washington, 2000. `http://data.cs.washington.edu/papers/xscan.pdf`.

[JLST01] H. Jagadish, L. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *Proc. DBLP Conference*, 2001.

[Kay] M. Kay. Saxon XSLT Processor. `http://sourceforge.net/projects/saxon`.

[LHEN02] G. Luo, P. Haas, C. Ellmann, and J. Naughton. A Scalable Hash Ripple Join Algorithm. In *ACM SIGMOD*, 2002.

[LPV00] B. Ludäscher, Y. Papakonstantinou, and P. Velikhov. Navigation-Driven Evaluation of Virtual Mediated Views. In *EDBT*, Konstanz, 2000.

[MBR+00] R. Moore, C. Baru, A. Rajasekar, B. Ludäscher, R. Marciano, M. Wan, W. Schroeder, and A. Gupta. Collection-Based Persistent Digital Archives (Part I&II). *D-Lib Magazine*, 6(3&4), March 2000.

[MP02] P. Mukhopadhyay and Y. Papakonstantinou. Mixing Querying and Navigation in MIX. In *ICDE*, 2002.

[MRL98] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *ACM SIGMOD*, 1998.

[MSH02] S. Madden, M. Shah, and J. Hellerstein. Continuously Adaptive Continuous Queries over Streams. In *ACM SIGMOD*, 2002.

[OMFB02] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Workshop on XML-Based Data Management (XMLDM)*, Prague, March 2002.

[Oni01] M. Onizuka. XML Toolkit and Tutorial. `http://www.cs.washington.edu/homes/suciu/XMLTK/`, 2001.

[PV00] Y. Papakonstantinou and V. Vianu. DTD Inference for Views of XML Data. In *ACM PODS*, 2000.

[Res02] Resin XSLT to Java compiler. `http://www.caucho.com/`, 2002.

[ROA02] Real-time Observatories, Applications, and Data management Network (ROADNet). `http://roadnet.ucsd.edu/`, 2002.

[SAB] Sablotron XSLT processor. `http://www.gingerall.com/`.

[SSB+00] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *VLDB Journal*, pp. 65–76, 2000.

[UF00] T. Urhan and M. J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

[VGD+02] S. D. Viglas, L. Galanis, D. J. DeWitt, D. Maier, and J. F. Naughton. Putting XML Query Algebras into Context, 2002. `http://www.cs.wisc.edu/niagara/Publications.html`.

[VN02] S. D. Viglas and J. F. Naughton. Rate-Based Query Optimization for Streaming Information Sources. In *ACM SIGMOD*, 2002.

[VS02] V. Vianu and L. Segoufin. Validating Streaming XML Documents. In *ACM PODS*, 2002.

[WA91] A. Wilschut and P. Apers. Dataflow Query Execution in a Parallel Main Memory Environment. In *Proc. PDIS*, 1991.

[XAL] Xalan XSLT processor. `http://xml.apache.org/xalan-j/`.

[XQu01] XQuery 1.0: An XML Query Language. W3C Working Draft, `www.w3.org/TR/xquery/`, 2001.

[XQu02] XQuery 1.0 Formal Semantics. W3C Working Draft, March 2002. `http://www.w3.org/TR/query-semantics/`.

[XSL02] XSLTC Compiler Documentation. `http://xml.apache.org/xalan-j/xsltc/`, 2002.