Semantic Query Optimization for Processing XML Streams with Minimized Memory Footprint

Ming Li, Murali Mani and Elke A. Rundensteiner
Department of Computer Science, Worcester Polytechnic Institute
Worcester, Massachusetts 01609, USA
(minglee|mmani|rundenst)@cs.wpi.edu

Abstract

XQuery evaluation over XML streams requires the temporary buffering of XML elements. This paper presents a semantic query optimization solution to minimize memory footprint during XQuery evaluation by exploiting schema knowledge. We focus on one particular class of constraints, namely, the Pattern Non-Occurrence (PNO) constraints for XML streams conforming to pre-defined DTDs. PNO constraints facilitate the early release of buffered data (early buffer release) or possibly avoid to ever store the data (buffer avoidance), thus achieving a minimized memory footprint. We develop an automaton-based technique to detect PNO constraints at runtime. For a given query, optimization opportunities of early buffer release and buffer avoidance which can be triggered by runtime PNO detection are explored and the optimization decision is then encoded into the Raindrop algebraic plan. We implement our optimization technique within the Raindrop XQuery engine. Our experimental studies illustrate that the proposed techniques bring significant performance improvement in both memory and CPU usage with little overhead.

1 Introduction

XML and XQuery [22] have been widely accepted as the standard data representation and query language for web applications. XML streams are passed through network for data exchange in a real-time infrastructure, which has the property of short response time and limited CPU/memory resources.

The in-time evaluation strategy is widely applied in the current XML stream engines for XQuery evaluation [10] [21] [12], where query evaluation is performed while the XML stream input is processed and the query engine produces query result on the fly. Due to the nature of XQuery, as a data-transformation query language, a certain amount of memory footprint (loading some elements to memory from the stream input and keeping them for a certain amount of time) is usually required. When the input consists of a large amount of XML tokens, the main memory buffer requirement can be significant, which might also lead to a significant CPU consumption due to the manipulation cost on the buffered data. To provide real-time responses, serious challenges in CPU and memory utilization are faced by the XQuery evaluation over XML streams.

In many practical applications, XML streams are generated following a pre-defined schema such as DTD and XML schema. For example, in network traffic monitoring, anomalies of network traffic flow may need to be detected from the statistical data sent in XML streams. In such a case, the XML stream, which would be generated by a work-flow engine or simply a customized program, will follow a pre-defined schema.

```
FOR  $a IN doc("source.xml") / root / news_report
RETURN
<Sources>     $a/source     </Sources>
<Dates>       $a/date       </Dates>
<Entries>     $a/entry      </Entries>
<Comments>    $a/comment    </Comments>
                                        Q1
```

```
FOR $c IN doc("source.xml") / root / news_report / entry
WHERE
$c / location = "Boston"
RETURN
<Reporters>  $c/reporter  </Reporters>
<Paragraphs> $c/paragraph </Paragraphs>
                                        Q2
```

Figure 1. XQuery Examples $Q1$ **and** $Q2$

Utilizing such schema constraints on the input data stream enables us to on-the-fly predict the non-occurrence of a given pattern within a bound context. This helps us to avoid data buffering and to release buffered data at an earlier moment, thus achieving a minimized memory footprint. The *Motivating Example* below illustrates such optimization opportunities.

Motivating Example. Suppose that we are evaluating the two example queries *Q1* and *Q2* shown in Figure 1 over the input stream in Figure 2.

For each *news_report* element, *Q1* lists the collection

*This work has been partially supported by the National Science Foundation under Grant No. NSF IIS-0414567.

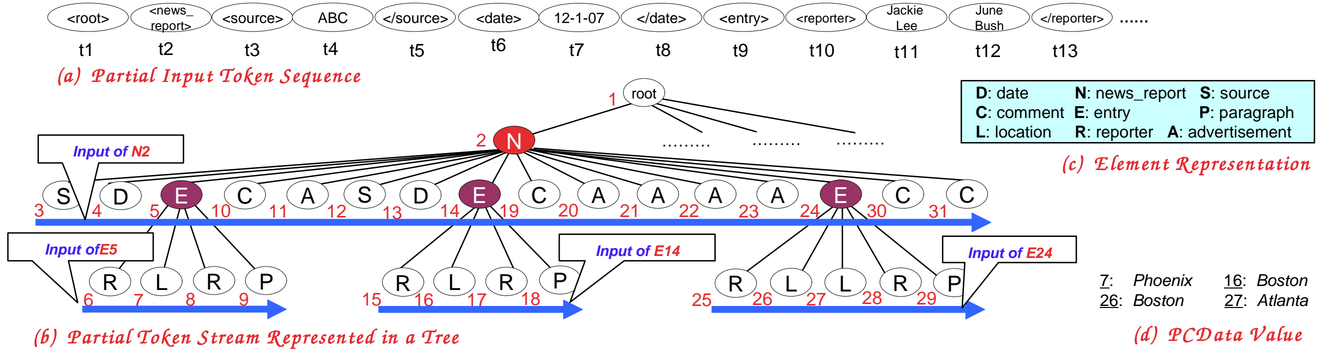

Figure 2. Input XML Token Stream

of its *source*, *date*, *entry* and *comment* subelements and *Q2* returns the *reporter* and *paragraph* under each of its *entry* subelements which contain at least one *location* equal to "Boston". Three types of token input are being considered: the *start tag*, *PCDATA* and the *end tag*. Figure 2(a) shows the first 13 tokens from the input token sequence. Figure 2(b) shows the equivalent XML tree representation. Each element node in the XML tree starts with one start tag token and ends with an end tag token. The nodes in the XML tree are shown by capitalized letters, where Figure 2(c) gives the corresponding description. The PCData values of *locations* are given in Figure 2(d).

Q1 extracts all *news_report* elements (such as the element *N2*). Under a binding, say *N2*, the child patterns that may appear in the return result are called the *expected patterns*. In *Q1*, *source*, *date*, *entry* and *comment* are expected patterns under the binding on *news_report*. Subelements of expected patterns will be located during pattern retrieval on the input stream. Similarly, *Q2* binds to each *entry* (*E5*, *E14* and *E24* in our example). The expected patterns under the *news_report* binding are *reporter* and *paragraph*.

We observe that: **(a).** for *Q1* there is an order requirement on the outputting elements of the expected patterns within each binding, such as that the complete list of *sources* needs to be output before all the *dates* within a *news_report*; **(b).** for *Q2*, the predicate satisfaction is needed before any data output can be performed, such as that a predicate on *location* needs to be satisfied before outputting any *reporter* and *paragraph* within an *entry* binding.

In evaluating *Q1*, due to the requirement in (a), traditional XML stream engines [10] [21] keep the elements of *source*, *date*, *entry* and *comment* until the element being bound has been completely received from the stream (end tag token of *N2* is reached). If a DTD [4] *<!ELEMENT news_report((source, date, entry, comment, advertisement)+, advertisement+, entry+, comment+)>* is given for the *news_report* element type, within the binding on *N2*, when we reach *A21*'s start tag, we can guarantee that in the future no more *source* and *date* elements will be encountered under the current binding (*N2*). Thus, we can output and then release the buffered *dates* and *entries* (*D4*, *D13*, *E5* and *E14*). Furthermore, token sequence of the *entry* element(s) to arrive in the future (*E24*) can be directly output without being buffered. Similarly, while reaching element *C30*, based on the schema we know that no more *entry* element will be seen under this binding. Thus buffered *comment* elements (*C10* and *C19*) can be output and released. *C30* and *C31* can be directly output without buffering.

In *Q2*, whether an *entry* element satisfies the predicate filtering in (b) is only known once the *entry* has been completely met. Thus within each *entry* all the *location* and *reporter* elements require buffering until reaching the end tag of the *entry*. Now suppose a DTD *<!ELEMENT entry(reporter+, location+, reporter+, paragraph+)>* is given for the *entry* element type. For *E5*, when the *reporter* element *R8* is met, we can guarantee that within the current *entry* no more *location* can be seen. Because none of the buffered *location* elements satisfies the filtering requirement (being equal to "Boston"), we are sure this *entry* cannot pass the predicate verification. At this stage, all the buffered *location* and *reporter* elements can be simply discarded and released from the memory and no further buffering is needed on this binding. Thus the token sequence of *R8* and *P9* will be directly dropped. Similarly for *E14*, the arrival of *R17* guarantees no more *location* elements will come under this binding. Because predicate verification gets satisfied by *L16*, the buffered *reporter* element (*R15*) can be output and released. The token sequence of the just-started *reporter* element *R17* can be directly output without buffering. The *paragraph* element(s) (*P18*) coming after can also be directly output because by the schema no *reporter* can come later than any *paragraph*. The same optimization process can be as well applied to *E24*.

Clearly, the memory footprint is reduced by applying such semantic query optimization shown above. We can predict that the CPU performance on query evaluation can also be improved if runtime constraints are captured with reasonable overhead costs. We observe from the above examples that although the semantic knowledge is known statically at the query compilation time, some actual optimization opportunities can only emerge and be detected at *runtime*. For example, buffered *entries* can be released and the handling on future receiving *entries* can be changed from "buffering" to "not buffering" after meeting the start tag token of *A21* in *Q1*. Thus, statically setting buffer avoidance for certain patterns based on the semantic knowledge cannot serve as a generic approach. In this work, we propose a strategy for dynamically detecting constraint knowledge the non-occurrence of a pattern for runtime memory footprint minimization.

State-of-the-Art. Reducing the memory cost is very important for stream applications, as it can enable us to support more application functionalities as well as yield a better memory and CPU performance. Only a limited number of XML stream processing engines [3] [10] [20] [12] [23] have looked at the schema-based optimization opportunity focusing on the memory footprint minimization. Among them, optimization in [3] is not stream specific. FluXQuery [12] only performs static optimizations thus it cannot switch the output mode of a pattern from "buffering" to "not buffering" dynamically at runtime. Besides that, it doesn't support filtering-related optimizations. The main focus of [20] and [10] is not on buffer minimization. They can only statically capture limited constraints from the given schema knowledge. [23] focuses on capturing and maintaining runtime schema change of the input stream, instead of an efficient way to improve buffer performance by applying a given schema.

Contribution. In this work, we study semantic query optimization (*SQO*) with particular focus on minimizing the memory footprint in XML stream processing. Our contributions include:

1. We reason about the pattern non-occurrence (*PNO*) constraint and develop an automaton-based technique to utilize DTD for runtime PNO monitoring.
2. We explore the optimization opportunities for memory footprint minimization that could arise for a given query expressed by our XQuery model. We then propose an efficient execution strategy for realizing embedded runtime PNO constraint detection and runtime plan optimization.
3. We implement our SQO technique within the Raindrop XQuery engine. Our system is efficiently augmented by our optimization module, which uses the Glushkov automaton to extract PNO constraints concurrently with query pattern retrieval.
4. We conduct experimental studies demonstrating that our proposed techniques bring significant performance gains in memory and CPU usage.

In Section 2 we introduce the pattern non-occurrence constraint and propose the mechanism to runtime detect such constraints based on a given DTD. Section 3 proposes the optimization model which utilizes pattern non-occurrence constraints to minimize the memory footprint. System implementation and experiments are discussed in Section 4. Section 5 introduces related works and Section 6 concludes the paper.

2 Pattern Non-Occurrence Constraints

By the previous examples, we can see that within a bound element, the non-occurrence of certain child patterns can runtime trigger the optimization leading to memory footprint minimization. In this section, we study such runtime constraint knowledge, named pattern non-occurrence (*PNO*) constraints. We first give its definition and introduce the corresponding checking algorithm. We then show that the presence of applicable PNO constraints can be monitored at runtime. Thereafter, we introduce the monitoring algorithm for detecting PNO constraint evolution dynamically.

2.1 Element Types and Element Evolution

An element type E is represented as an atomic symbol, *P(E)* represents as the regular expression for type E where $E \rightarrow$ *P(E)*. Under the XML context, the element type is simply denoted by a given *tag name*. *P(E)* is defined by the DTD for type *E*. *SymbSet(P(E))* is the set of all possible subelement types of type *E*. *L(P(E))* denotes the language defined by *P(E)*.

As example let's look at Figure 3(a). A DTD *P(news_report)* is given. A *news_report*'s subelement can be *source*, *date*, *entry*, *comment* or *adver−tisement*, contained by *SymbSet(P(news_report))*.

Element Prefix. A partially received element of type E is called an element prefix of E. The set of possible prefixes of type E is denoted as *Prefix(E)*. Given a finite sequence p, p is in set *Prefix(E)* if there exists an element *ele* in *L(P(E))* where p is *ele*'s prefix.

Element Evolution. Given $p \in$ *Prefix(E)*, an element evolution of p is the process of p evolving into another element prefix p' of the same type by concatenating additional subelements. *Growth(p, E)* is the set of all possible evolved portion: given $p \in$ *Prefix(E)*, for any $p' = pq \in$ *Prefix(E)*, q is in *Growth(p, E)*. An element

evolution of p in *Prefix(E)* is denoted as $\Rightarrow$*(p, q, E)* while the corresponding growth portion is q, which is in *Growth(p, E)*.

Element prefix p of type *news_report* is shown in Figure 3(c). The sequence q = "*advertisement advertisement advertisement*" is in p's *Growth* set. p evolves to the new element prefix p' by $\Rightarrow$*(p, q, E)* (Figure 3(d)). Note that the example is representing the *news_report* element *N2* given in Figure 2.

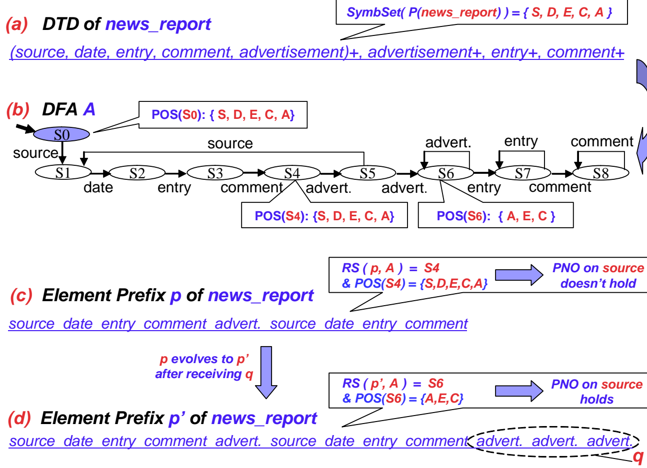

Figure 3. PNO Constraint

2.2 Pattern Non-Occurrence Constraint

2.2.1 Semantic Knowledge on Element Types

We can represent a regular expression *P(E)* using an equivalent *Deterministic Finite Automaton* (*DFA*). For *P(E)*, we let *AutoSet(P(E))* denote the DFAs accepting *L(P(E))*, without redundant states. Given a regular expression [13]. For an element prefix p of type E and a given DFA τ in *AutoSet(P(E))*, *RS(p,* τ*)* denotes the state in τ reached by taking p.

As example, DFA A in Figure 3(b) is an equivalent automaton of the regular expression given for type *news_report*. Element prefix p reaches the state *S4* on A (*RS(p, A)* = *S4*) and p' reaches state *S6* on A. Suppose p' keeps evolving by taking in one *entry*. The state transits from *S6* to *S7*.

2.2.2 PNO Rule

Pattern Non-Occurrence(PNO) Constraint. For p in *Prefix(E)*, the PNO constraint on symbol *symb* holds **iff** *symb* is not contained by any $p' \in$ *Growth(p, E)*, denoted as *PNO(symb, p, E)* = TRUE. Given p in *Prefix(E)*, *PNO(symb, p, E)* guarantees that subelements of type *symb* will not be seen in the remaining portion of the current element.

Possible Occurrence Set(POS). For a DFA state S, the *Possible Occurrence Set*, denoted as *POS(S)*, is the set of symbols which can occur until reaching a final state. *POS(S)* for a DFA without redundant states can be defined as: let *NeighborState(S)* = $\{S' \mid$ there exists an automaton transition from S to $S'\}$, *FutureSet(S)* = $S \cup$ *NeighborState(S)* $\cup_{\forall S' \in NeighborState(S)} FutureSet(S')$, *TransitSymbol(S)* = $\{symb \mid \exists\, S'$ S transits to S' through $symb\}$, then *POS(S)* = $\cup_{\forall S' \in FutureSet(S)} TransitSymbol(S')$.

Datalog [1] or the encoding of some graph reachability algorithm can be applied for calculating POS for the states in a given DFA. The algorithm takes a DFA τ as input and outputs the POS for every automaton state of it. We refer to such algorithm as *POS_Compute(*τ*)*. Some example results for *POS_Compute(A)* is shown in Figure 3(b). Take the start state *S0* as example. Obviously, *POS(S0)* equals to *SymbSet(P(news_report))*.

PNO Rule. Given p in *Prefix(E)*, any τ in *AutoSet(P(E))* and symbol *symb*, *PNO(symb, p, E)* holds **iff** *symb* $\notin$ *POS(RS(p,* τ*))*.

Whether *PNO(symb, p, E)* holds can be determined by a simple application of the above PNO rule. Given p in *Prefix(E)*, DFA τ in *AutoSet(P(E))* and symbol *symb*, determining *PNO(symb, p, E)* is a simple POS check, which returns TRUE if *symb* is contained by *POS(RS(p,* τ*))*.

As example we apply the PNO rule to the element prefix p in Figure 3. By running p on DFA A, state *S4* (*S4* = *RS(p, A)*) is reached. Because *source* is contained by state *S4*'s *POS*, *PNO(source, p, news_report)* does not hold. For *PNO(source, p', news_report)*, we determine that the constraint holds since *source* is not contained by *POS(S6)*.

2.3 PNO Constraint Evolution

2.3.1 Definition

Element evolution $\Rightarrow$*(p, q, E)* is referred to as a *singleton element evolution* if q consists of only one symbol (q = "*symb*", $|q| = 1$). It is denoted as $\mapsto(p, symb, E)$. Given p in *Prefix(E)*, singleton element evolution sg:$\mapsto(p, symb, E)$ and symbol $symb'$, let $p' = p\ symb$, if *PNO(symb', p', E)* holds but *PNO(symb', p, E)* does not, there is a *PNO constraint evolution* on *symb'* at *sg*, denoted as $\xi(symb')$ at *sg*.

Take Figure 3 as example. A PNO constraint evolution on *source* occurs when the second *advertisement* in q is met, because this *advertisement* triggers the state transit from *S5* to *S6* and *source* is contained by *POS(S5)* but not by *POS(S6)*.

2.3.2 Monitoring PNO Constraint Evolutions

Theorem 1. (Monotonicity of PNO Constraints) Given element prefixes $p1$, $p2$ of type E and $p1$ is the prefix of $p2$, for symbol *symb*, if *PNO(symb, p1, E)* holds, then *PNO(symb, p2, E)* also holds.

This theorem can be proven by contradiction. Due to space limitation, the proof is skipped in this paper. Based on *Theorem 1*, *Theorem 2* is straightforward:

Theorem 2. Assume there exists a PNO constraint evolution on *symb* at $sg{:}\mapsto(p,\ symb',\ E)$. Let $p' = p\ symb'$. For any p'' in $Growth(p', E)$, $PNO(symb, p'p'', E)$ holds.

By this theorem we know that after the PNO constraint evolution happens on *symb*, the PNO constraint on *symb* will stay TRUE through the growing of the current element. The earliest we can guarantee the PNO constraint on *symb* being satisfied is the moment when the singleton element evolution *sg* happens.

Algorithm 1 Monitoring Process of PNO Evolution

Procedure: *PNO_Monitoring*
Input:
(1) DFA τ equivalent to $L(P(E))$ (S_0 is the start state and POS for each state in τ is pre-computed)
(2) symbol *symb*
(3) runtime input – a well-formed symbol sequence *SEQ* plus the termination message *End_of_Binding*
Output: notification of $\xi(symb)$

state $S = S_0$
on receiving receiving symbol input $symb_{input}$:
symbol $symb' = symb_{input}$
$S' = tf(S,\ symb')$ (tf as the transit function of τ)
if S != S' (transiting to a new state in τ) **then**
  $S = S'$
  if $symb \notin POS(S)$ **then**
    return notification $\xi(symb)$
  end if
end if
on receiving *End_of_Binding*:
return notification $\xi(symb)$

We propose the monitoring algorithm to keep track of the PNO evolution over a growing input symbol sequence. Given a sequence SEQ of symbols $symb_1$, $symb_2$, $symb_3$, ... if SEQ corresponds to a sequence of singleton element evolution steps sg_1, sg_2, sg_3,... where $sg_1 = \mapsto(\epsilon, symb_1, E)$, $sg_2 = \mapsto(symb_1, symb_2, E)$, $sg_3 = \mapsto(symb_1\ symb_2, symb_3, E)$,... We refer to *SEQ* as a well-formed input sequence of type *E*, where ϵ represents the empty string. SEQ corresponds to the incremental growth of an element of type *E*. Algorithm 1 sequentially reads in a well-formed sequence *SEQ* of type *E* and raises notification if there exist $\xi(symb)$ at receiving an input symbol $symb_{input}$. While the sequence terminates (receiving *End_of_Binding*), PNO on *symb* will be notified if not being raised before.

As example let's again look at Figure 3. The PNO constraint on *source* holds at p' however the PNO evolution happens at $p_{evolution} = p$ *advertisement advertisement*. While the start tag token of the second *advertisement* triggers the automaton transition from *S5* to *S6*, the monitoring algorithm here captures the absence of *source* in *POS(S6)*. Thus the PNO constraint on *source* evolves from FALSE to TRUE and stays TRUE for the remainder of processing the current element.

```
CoreExpr ::= ForClause WhereClause? ReturnClause
           | PathExpr
PathExpr ::= PathExpr "/"|"//" TagName|"*"
           | varName
           | streamName
ForClause ::= "for" "$"varName "in" PathExpr
           ("," "$"varName "in" PathExpr)*
WhereClause :: = "where" BooleanExpr
BooleanExpr ::= PathExpr CompareExpr Constant
           | BooleanExpr and BooleanExpr
           | PathExpr
CompareExpr ::= ">"|"!="|"<"|"<="|">"|">="
ReturnClause = "return" CoreExpr
           |<tagName>CoreExpr ("," CoreExpr)* </tagName>
```

Figure 4. Supported XQuery Subset

3 PNO-Driven Optimization

3.1 Supported Language

In our XQuery engine we focus on a core subset of XQuery described in Figure 4. Basically, we allow "for... where... return" expressions (referred to as FWR) where (1) the "return" clause can further contain FWR expressions and (2) the "where" clause contains conjunctive predicates each of which is a comparison between a variable and a constant.

3.2 PNO-Driven Execution Strategy

For a binding $v, a naive execution strategy, such as the **just-in-time execution strategy** introduced in [21], performs the predicate checking and data output after the bound element has been completely received from the input stream. The buffered subelements of the binding can thus be released from the memory only after the end tag of $v is encountered. Algorithm 2 sketches the just-in-time execution strategy. The strategy used by [10] also falls into this category.

We call the method of handling elements of an expected pattern the **handling mode** of this pattern. If the retrieved elements of a pattern are required to be buffered, the handling mode of this pattern is referred to as *HOLD*. Thus, all expected patterns of a binding are with the *HOLD* handling mode by the above just-in-time execution strategy.

From the motivating example, we can observe that there are two major **optimization opportunities** facilitated by PNO constraints:

1. **Early Buffer Release.** Some buffered elements can be released earlier than the completion of $v.

Algorithm 2 Just-In-Time Execution Strategy

```
Procedure: JustInTime_Strategy
Input: token sequence within a binding, terminated by T
Output: query result of the binding

on receiving a new subelement e:
if e's pattern type E is an expected subelement type then
   buffer the token sequence of e
else
   discard the token sequence of e
end if
on receiving binding termination T:
performing operation on buffered subelements within the binding
and then releasing the subelements
```

For example, in *Q1*, when *A21* is met, the buffered *D4*, *D13*, *E5* and *E14* can be output and the memory can then be released.

2. **Buffer Avoidance.** Elements under expected patterns do not always need to be buffered. They can instead be directly output (such as the *E24* in *Q1*), which is referred to as *on-the-fly token output* or be directly dropped (such as *R8* in *Q2*), which is referred to as *on-the-fly token dropping*. Through this, buffering on some elements are avoided.

We thus propose the execution strategy which utilizing the runtime PNO constraint for query optimization. Our strategy follows the *Event Condition Action* (*ECA*) rule-based framework. Through the ECA framework, expected PNO constraints (**Condition**) and the corresponding optimization steps (**Action**) are associated as a *condition-action pair*. Based on the stream input (**Event**), the PNO monitor reports PNO evolution at runtime, which triggers the satisfaction of expected PNO conditions and then the corresponding action will be taken. There are two types of actions can be driven by runtime PNO monitoring:

1. **Operation on the current buffered data**, which checks the buffered predicates (checking *L7* after reaching *R8* while evaluating *E5* in *Q2*), outputs the buffered data (outputting *D4*, *D13*, *E5* and *E14* after reaching *A21* while evaluating *N2* in *Q1*) and then releases the buffered data. *Early buffer release* is thus achieved by such operation.

2. **Runtime switch of handling mode,** which changes the handling mode for an expected pattern from HOLD to *TOKEN_OUTPUT* or *TOKEN_DROP*. Hence, future receiving elements of the pattern can be handled in the way of on-the-fly token output or on-the-fly token dropping. *Buffer avoidance* is thus achieved. Take the *entry* pattern in *Q1* as example. At the beginning the pattern is with the *HOLD* mode and requires to be buffered. After reaching *A21*, its mode is switched from *HOLD* to *TOKEN_OUTPUT*. The future receiving *entry* element(s) (*E24*) can thus be directly output in tokens without any buffering.

Algorithm 3 PNO-Driven Execution Strategy

```
Procedure: PNO-Driven_Strategy
Input:
(1) token sequence within a binding, terminated as T
(2) PNO Monitor M running the PNO_Monitoring procedure
(3) expected condition-action set
Output: query result of the binding

on receiving a new subelement e:
pass E (e's pattern type) to M
if new PNO evolution is detected by M then
   check the condition-action pairs
   if new condition is satisfied then
      perform its corresponding action
   end if
end if
if E is an expected subelement type then
   perform operations defined by E's handling mode
else
   discard the token sequence of e
end if
on receiving binding termination T:
pass End_of_Binding message to M
```

Algorithm 3 depicts the procedure of the proposed PNO-driven execution strategy. Based on such optimization framework, the rest of this section we will describe our mechanism to determine the expected PNO conditions and their corresponding actions (condition-action pairs) for a given XQuery.

3.3 Optimization on Sequence Output

Let's first consider an XQuery of the form as "*FOR* $\$v$ *IN* .../v *RETURN* $\$v/r_1$, $\$v/r_2$, ... $\$v/r_n$" ($Q_{seq}$), where for every element binding on $\$v$, returning the list of pattern r_1 to r_n with the binding. The required order among the output patterns is referred to as *output sequence order*, where the list of r_i elements must be output earlier than the list of r_k elements, if $i < k$. Straightforwardly, for elements of type r_1, they can be output directly without any buffering. For $1 < k \leq$ n, before any output on the elements of type r_k, all the r_1, r_2, ..., r_{k-1} elements must be already output. Hence the elements of type r_1 to r_{k-1} need to be completely met, which can be captured by the satisfaction of PNO constraints on these elements. Thus, before the current element evolves to a state satisfying all these PNO constraints, elements of type r_k (1 < k $\leq$ n) have to be buffered. After such PNO condition is satisfied, we can perform the following actions: (1) Output the buffered r_k elements; (2) Release the buffer on r_k elements; (3) Change the handling mode on r_k from *HOLD* to *TOKEN_OUTPUT*. Thus, for Q_{seq}, the event-condition pairs encoded with binding $\$v$ includes:

Condition 1 (C1): $\emptyset$
Action 1 (A1): change the *handling mode* for r_1 from *HOLD* to *TOKEN_OUTPUT*.

Condition 2 (C2): PNO holds on r_1.
Action 2 (A2): output and then release the buffered r_2 elements, change the *handling mode* for r_2 from *HOLD* to *TOKEN_OUTPUT*.
......

Condition n (Cn): PNO holds on r_1,...,r_{n-1}.
Action n (An): output and then release the buffered r_n elements, change the *handling mode* for r_n from *HOLD* to *TOKEN_OUTPUT*.

Following algorithm 3, we monitor the PNO evolution on pattern r_1 to r_n and undertake actions upon the satisfaction of the corresponding conditions defined above. If two conditions are satisfied together, the one with corresponding action associating with an earlier pattern in the output sequence will be fired earlier for guaranteeing the result correctness.

Let's look at the evaluation of *Q1* as example. At the beginning, the *source*'s mode is set as *TOKEN_OUTPUT* and others returned patters are set as *HOLD*. The start tag of *A21* triggers the PNO evolution on *date* and *entry*. Thus, condition C_2 and C_3 get satisfied at the same time. Action A_2 is taken, followed by action A_3: *D4*, *D13*, *E5* and *E14* are thus output and released, and the handling mode of *entry* is set to *TOKEN_OUTPUT*. The *E24* is thus directly output without buffering. The PNO on *entry* evolves when *C30* is met, which triggers action A_4: buffer on *C10* and *C19* is output and then released, the handling mode on *comment* is set to *TOKEN_OUTPUT*, which leads to the direct output of *C30* and *C31*.

3.4 Optimization for Correlated Output

We then consider XQueries with correlated binding, such as the form "*FOR* $v *IN* .../v, $u *IN* $v/u *RETURN* $u v/r_1, v/r_2, ... v/r_n" ($Q_{nest-seq}$), where the RETURN clause combines $u and patterns within its correlated outer binding $v. Straightforwardly, any output on the u elements requires that all the elements of pattern r_1, r_2, ..., r_n have been completely met. Thus, before the current element satisfies the PNO constraint on r_1 to r_n, elements of type u have to be buffered. With the satisfaction of the PNO constraints on $r_1, r_2, ..., r_n$ (**Condition 1**), we can perform the following action: (1) Supposing the buffered u elements are $u.1$, $u.2$, ..., $u.m$ in their arrival order, for each $u.i$ from $u.1$ to $u.m$, output the buffer r_1 elements, r_2 elements, ..., r_n elements, then output $u.i$; (2) Release all the buffer on elements of r_k ($1 \leq k \leq n$); (3) Change the *handling mode* for u from *HOLD* to *TOKEN_OUTPUT*. (**Action 1**)

Note that the *TOKEN_OUTPUT* handling mode is slightly more complex than the examples shown before. Additional action for appending the buffered collection of r_1 to r_n are needed besides directly outputting each newly arriving u element, following on the output requirement. For example, if the RETURN clause of $Q_{nest-seq}$ is changed to "v/r_1, v/r_2, ... v/r_n $u", such output appending will be performed before the token output on the newly arrived u element: while the start tag token of the u element is met, we output all the buffered elements from each r_k pattern ($1 \leq k \leq n$), then output the start tag token as well as the following input tokens of this on-progress u element.

After the PNO of pattern r_1 to r_n as well as the PNO of pattern u all have been satisfied with in the binding $v (**Condition 2**), the buffered elements of pattern r_1 to r_n can be released (**Action 2**) because no more u element can come within the binding.

3.5 Optimization on Conjunctive Filtering

Let's now consider XQueries with conjunctive predicate filtering in the form as "*FOR* $v *IN* .../v *WHERE* v/p_1 ="val_1", v/p_2 ="val_2", ... v/p_m ="val_m" *RETURN* ..." (Q_{filter}). For a query under such form, no result can be returned for every element binding on $v if the filtering on any of the pattern p_1 to p_m fails.

A predicate pattern $v/p may fail if its p may not occur within $v, or it is involved in a selection. The failure of a required $v/p filters out $v. If the PNO of pattern p is satisfied within $v, we can test whether p fails. This test is an *early filtering* because otherwise we could have only concluded whether p fails when the end tag of $v is encountered. If p fails, all the buffered data can be released and all the potential buffering can be avoided within this $v. On the other hand, once the predicate checking is determined to be satisfied, the query can be handled in the way as the WHERE clause is removed. Refer to the query which takes away the WHERE clause of Q_{filter} as Q'_{filter}. The condition-action encoding on $v for Q_{filter} is thus as following: for every condition-action pair of Q'_{filter}, extending its condition to include the PNO of all the predicate patterns p_1 to p_m, in order to trigger the optimization in case $v satisfies the filtering. Besides that, a new condition-action pair will be added corresponding to each single predicate pattern p_i ($1 \leq i \leq m$), for informing the early filtering once any filter is determined to be failing.

Take *Q2* as an example. For *E5*, when the start tag token of *R8* is met, the PNO condition on *location* is met thus the buffered *location(s)* (*L7*) will be checked. *E5* is determined as failing because *L7* is not equal to "Boston". Early filtering can then be performed: the buffered *R6*, *L7* are released and the handling mode of *location*, *reporter* and *paragraph* are turned to *TOKEN_DROP*. So *R8* and *P9* will be directly dropped without buffering.

3.6 Optimization for Multi-Level Bindings

There are two different scenarios of PNO-driven optimization for querie with multi-level bindings:

(1) **The inner binding is with the handling mode of DIRECT_OUPUT or DIRECT_DISCARD**: this inner binding can be treated in the same fashion as the top most binding for buffer optimization. For

example, for evaluating XQuery *Q3* given in Figure 5 over the input XML stream in Figure 2, when the inner binding on *E24* is processed, *E24* is with the mode of *DIRECT_OUPUT*. Thus, the binding on *N2* does not affect the process of the subelements of *E24*: due to *L27* equal to "Boston", when the start tag token of *R28* is reached, *R25* can be output/released and further arriving tokens of *R28* and *P29* can be output directly without buffering.

(2) **The inner binding is with the handling mode of HOLD**: if the binding does not contain any predicate checking or it is with a satisfied predicate checking, no buffer optimization can be applied on this inner binding for decreasing memory consumption; on the other hand, if the binding is with a predicate determined to be failing, the buffer for the inner binding can be released and no buffering on this binding is further needed. Again we evaluate *Q3* over the input in Figure 2. when the inner binding on *E14* is processed, it is with the mode of *HOLD*. Due to *E14* satisfying the predicate filtering, no optimization can be performed memory-wise. The process of *E5* (the mode as *HOLD*) shows the opposite case. The buffered *R6* can be released and no further buffer is needed when *R8* is reached, because *E5* fails the predicate checking.

```
FOR $a IN / root / news_report                          Q3
RETURN
<Sources>   $a/source    </Sources>
<Dates>     $a/date      </Dates>
<Entries>
   FOR $b IN $a/entry
   WHERE $b / location = "Boston"
   RETURN
   <Reporters>    $b/reporter     </Reporters>
   <Paragraphs>   $b/paragraph    </Paragraphs>
</Entries>
<Comments>  $a/comment   </Comments>
```

Figure 5. XQuery Example *Q3*

3.7 Condition-Action Encoding for XQuery

Above we have described three different query templates and the corresponding condition-action encoding mechanisms. The general encoding mechanism for a given XQuery is performed by traversing the tree structure of a query and compute the optimization decisions for certain destination binding using the encoding method of the query templates. We apply the *query tree* in [21] to represent the structural pattern in an XQuery. Due to space limitation, in this paper we just describe the basic idea for the encoding algorithm. The encoding algorithm has two main components: the *tree traverser* and the *template applier*. The *traverser* traverses the query tree and directs the *applier* to certain destination nodes. The *applier* outputs a set of event-condition pairs, attached to the corresponding destination node. Initially, the traverser is called on the root node of the query tree and recursively operates on each destination node.

On each destination binding, the condition-action encoding is represented by a data structure called *CAG* (Condition-Action Graph). CAGs efficiently keep track of the conditions and ensure that an action is taken when its corresponding condition has been satisfied. A CAG is a state machine where each state (*condition state*) represents a set of PNO constraints. Each state is associated with its corresponding action set which will be fired after the constraints get satisfied. As examples, the CAG of Q_{seq} and the CAG of $Q_{nest-seq}$ are shown in Figure 6 and Figure 7 respectively.

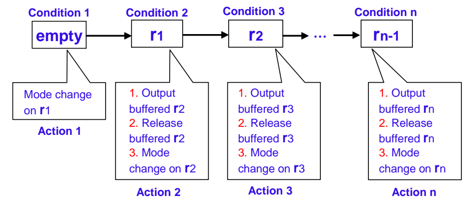

Figure 6. CAG of Q_{seq}

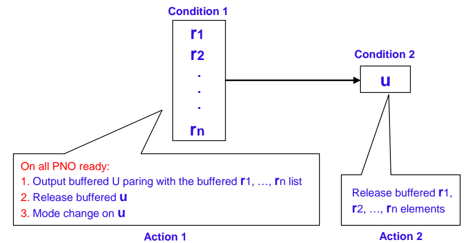

Figure 7. CAG of $Q_{nest-seq}$

4 Performance Evaluation

4.1 System Implementation

We have incorporated the proposed optimization strategy into *Raindrop* system [21] using Java 1.4. Figure 8 shows the system framework.

In our *Constraint Engine*, the *Glushkov Automaton* (*GA*) [5] is used for PNO constraint monitoring. Referred to [5], for an one-unambiguous regular expression, an equivalent GA can be constructed in quadratic time. A GA has the properties that: (1) every state in a GA corresponds to a symbol in the regular expression, and (2) every transition has one and only one destination state. In a GA, there is an one-to-one mapping from its automaton state to the symbols in the corresponding regular expression. Such mapping leads to a convenient automaton construction and simplified automaton states.

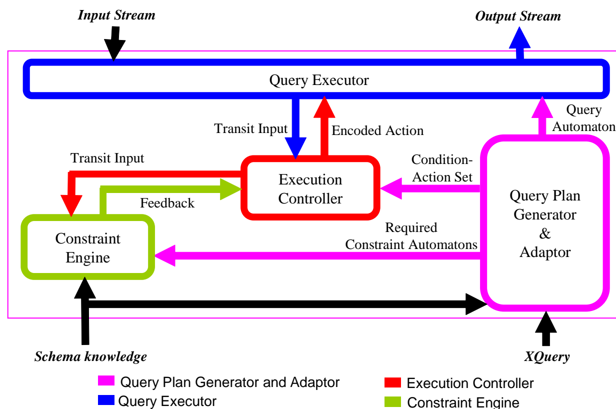

Figure 8. System Architecture

4.2 Experimental Setting

Experiments are run on two Pentium 4 3.0GHz machines, both with 768MB of RAM. One machine sends the XML stream to the second machine, i.e., the query engine. We assume the incoming data is well-formed and do not check for the well-formedness. The parsing time in the overall execution time thus is negligible.

In section 4.3 we will report the performance of our SQO techniques on a 5G data input from the Protein Sequence Database (PSD) [9]. From its DTD, we can see that the data can be highly irregular. The dataset contains a sequence of *ProteinEntry* elements. A *ProteinEntry* element has 13 subelements: 8 of them can be optional. The experiment tests queries two varying factors: filter position and selectivity. 30 different queries with filter position varying from 1 to 12 and selectivity varying from zero to 100% are evaluated. As our future work, we plan to perform experiments on an on-line auction data generated by ToXGene [2], which conforms to the schema used in XMark [18].

4.3 Experimental Results

Memory Consumption. By changing the filter position and selectivity, our SQO technique should be able to minimize the amount of data that is buffered: with a smaller selectivity (less results being produced) or an earlier filter (position being smaller), less data needs to be buffered. The results shown in Figure 9 provide the verification. X axis shows the combination of different filter position and selectivity, which includes all 30 queries. Y axis shows the accumulative memory consumption for each query.

CPU Performance. Figure 10 shows the chart of query execution time. We can see that more avoidance on data buffering generally leads to a bigger enhancement in CPU performance. Y axis here shows the execution time for each query. In the best case (i.e., the query for which selectivity is 0% and the position is zero), plans optimized with SQO reduce the execution time of the original plan by 64%.

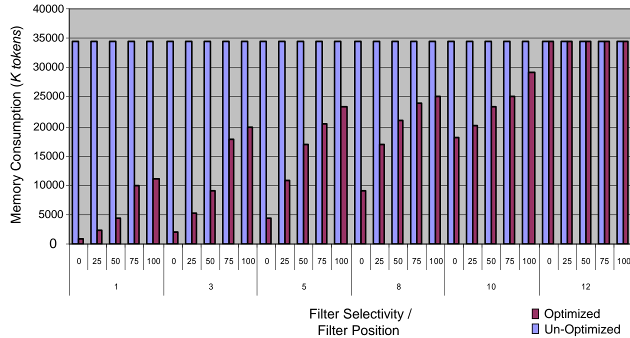

Figure 9. Memory Consumption

Technique Overhead. By fixing the selectivity at 100% and the filter position at the right-most end, the overhead of our proposed SQO techniques is shown in Figure 11. In such scenario none of the monitoring checking will lead to any buffer avoidance or early buffer release. The performance difference between the optimized and un-optimized plan is the overhead in the worst case, introduced by the cost of PNO monitoring.

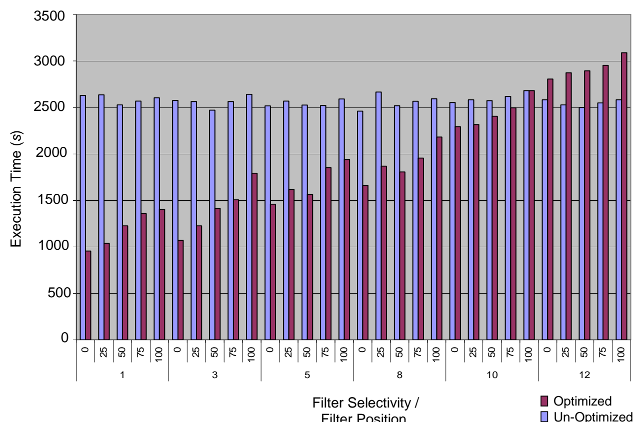

Figure 10. CPU Performance

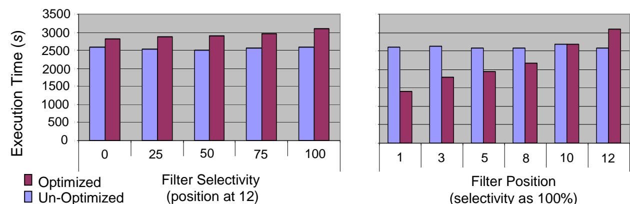

Figure 11. Overhead of the SQO Technique

Conclusion. Above experimental results reveal that the proposed SQO is practical in two senses: (1) the technique can surely reduce the memory consumption;

(2) the savings brought by the techniques on CPU performance can be significant in most cases.

5 Related Work

Projecting XML [16] [3] [19] aimed to address the problem of reducing memory by pre-filtering the data from the data input stream based on the paths from the query. [6] utilized a pre-computed index to reduce the memory and CPU costs. However, these approaches do not fit in the requirement of streaming scenario.

Streaming query evaluation for XPath queries has been studied in [11] [8] [7]. and streaming for XQuery has been studied in [10] [17] [14] [15] [21] [23]. Commonly these XQuery engines try to address XQuery on streams using automaton / transducer-networks for pattern retrieval and introducing stream-specific operations for data filtering and result re-construction.

[10] [20] [12] [23] are the closest to this work. [10] mainly focuses on the state sharing for multiple query evaluation. The goal of [12] is to minimize the buffer size by directly outputting tokens of some extracted patterns. It only performs static optimizations thus it cannot be switch the output mode of a pattern from "buffering" to "outputting" / "dropping" at runtime (*HOLD* to *TOKEN_OUTPUT* / *TOKEN_DROP* in our previous description). They also do not support filtering-related computations thus no early filter can be performed. [20] also uses schema constraints to detect the failure of predicate patterns earlier and hence can purge the data earlier when an element fails on its predicate(s) and will thus not be returned. However its focus is on avoiding unnecessary pattern retrievals. It cannot perform join-related computations incrementally nor other aspects of the filtering-related optimization. It only captures limited cases of XML constraints, instead of completely considering the given input schema. The main focus of [23] is capturing and maintaining runtime schema change of the input stream. It can be combined with the SQO techniques proposed by this paper.

6 Conclusion

The memory footprint in XML stream processing can be decreased by applying schema knowledge of the input data. In this work we develop an automaton-based technique to utilize schema knowledge for runtime PNO constraint detection. We identify possible optimization opportunities triggered by runtime PNO constraints and encode them into the Raindrop algebraic plan. We implement our optimization technique within the Raindrop XQuery engine. Our experimental studies illustrate that our techniques bring significant performance improvement in both memory and CPU usage with little overhead.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. Foundations of databases. 1995.

[2] D. Barbosa and A.Mendelzon. ToXgene: a template-based data generator for XML. In *WebDB*, pages 49–54, 2002.

[3] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen. Type-based xml projection. In *VLDB*, pages 271–282, 2006.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language 1.0 (fourth edition). In *http://www.w3.org/TR/REC-xml/*, 2006.

[5] A. Bruggemann. One-unambiguous regular languages. In *Infor. and Comp., 142(2)*, pages 182–206, 1998.

[6] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation- vs. index-based xml multi-query processing. In *ICDE*, pages 139–150, 2003.

[7] Y. Chen, S. B. Davidson, and Y. Zheng. Vitex: A streaming xpath processing system. In *ICDE*, pages 1118–1119, 2005.

[8] Y. Chen, S. B. Davidson, and Y. Zheng. An efficient xpath query processor for xml streams. In *ICDE*, page 79, 2006.

[9] P. S. Database. http://pir.georgetown.edu.

[10] Y. Diao and M. J. Franklin. Query processing for high-volume xml message brokering. In *VLDB*, pages 261–272, 2003.

[11] T. J. Green, A. Gupta, G. Miklau, and M. Onizuka. Processing xml streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[12] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *VLDB*, pages 228–239, 2004.

[13] D. Kozen. Automata and computability. In *W.H.Freeman and Company, New York*, 2003.

[14] X. Li and G. Agrawal. Efficient evaluation of xquery over streaming data. In *VLDB*, pages 265–276, 2005.

[15] B. Ludascher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based xml query processor. In *VLDB*, pages 227–238, 2002.

[16] A. Marian and J. Siméon. Projecting xml documents. In *VLDB*, pages 213–224, 2003.

[17] F. Peng and S. S. Chawathe. Xpath queries on streaming data. In *SIGMOD Conference*, pages 431–442, 2003.

[18] A. Schmidt, F. Wass, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: a benchmark for xml data management. In *VLDB*, pages 974–985, 2002.

[19] M. Schmidt, S. Scherzinger, and C. Koch. Combined static and dynamic analysis for effective buffer minimization. In *ICDE*, pages 236–245, 2007.

[20] H. Su, E. A. Rundensteiner, and M. Mani. Semantic query optimization for XQuery over xml streams. In *VLDB*, pages 1293–1296, 2005.

[21] H. Su, E. A. Rundensteiner, and M. Mani. Automaton meets algebra: a hybrid paradigm for xml stream processings. *DKE Journal*, pages 576–602, 2006.

[22] W3C. XQuery 1.0 and Xpath 2.0 formal semantics. http://www.w3.org/TR/query-semantics, 2004.

[23] S. Wang, H. Su, M. Li, M. Wei, S. Yang, E. A. Rundensteiner, and M. Mani. R-sox: runtime semantic query optimization over xml streams. In *VLDB*, pages 1207–1210, 2006.