

# Semantics based Buffer Reduction for Queries over XML Data Streams

Chi Yang<sup>†</sup> Chengfei Liu<sup>†</sup> Jianxin Li<sup>†</sup> Jeffrey Xu Yu<sup>‡</sup> and Junhu Wang<sup>§</sup>

<sup>†</sup>Faculty of Information and Communication Technologies  
Swinburne University of Technology  
Melbourne, VIC 3122, Australia  
{cyang, cliu, jili}@ict.swin.edu.au

<sup>‡</sup>Department of System Engineering and Engineering Management  
The Chinese University of Hong Kong  
Hong Kong, China  
yu@se.cuhk.edu.hk

<sup>§</sup>School of Information and Communication Technology  
Griffith University  
Gold Coast, QLD 4222, Australia  
j.wang@griffith.edu.au

## Abstract

With respect to current methods for query evaluation over XML data streams, adoption of certain types of buffering techniques is unavoidable. Under lots of circumstances, the buffer scale may increase exponentially, which can cause memory bottleneck. Some optimization techniques have been proposed to solve the problem. However, the limit of these techniques has been defined by a concurrency lower bound and been theoretically proved. In this paper, we show through an empirical study that this lower bound can be broken by taking semantic information into account for buffer reduction. To demonstrate this, we build a SAX-based XML stream query evaluation system and design an algorithm that consumes buffers in line with the concurrency lower bound. After a further analysis of the lower bound, we design several semantic rules for the purpose of breaking the lower bound and incorporate these rules in the lower bound algorithm. Experiments are conducted to show that the algorithms deploying semantic rules individually and collectively all significantly outperform the lower bound algorithm that does not consider semantic information.

*Keywords:* XML Data Stream, Query Optimization, and Buffer Management.

## 1 Introduction

With the development and deployment of XML technologies, processing of XML format streaming data has become critical for data dissemination in cases where the data throughput or size make it infeasible to rely on the conventional approach which stores the data before

processing it (Altinel and Franklin, 2000, Gray, 2004, Jian et al., 2003). The streaming XML data is generated naturally by message-based Web services such as purchase orders, retail transactions, personal content delivery, etc. In all those services, loosely coupled systems interact by exchanging high volumes of business data tagged in XML as a token sequence forming continuous data streams (Fegaras et al., 2002, Bose and Fegaras, 2004, Peng and Chawathe, 2003, Ludasher et al., 2002).

Some methods (Fegaras et al., 2002, Peng and Chawathe, 2003, Ludasher et al., 2002, Su et al., 2004, Su et al., 2005, Altinel and Franklin, 2000) have been proposed for evaluating XPath or XQuery queries over XML data streams. When there are large bunches of simple queries evaluated on a document, automaton-based (Altinel and Franklin, 2000, Ludasher et al., 2002) methods are attractive due to their efficiency and clean design. The weakness of automaton-based algorithm is that it becomes difficult when supporting XPath queries with predicates. For any algorithm developed considering queries over XML data streams with predicates, the buffer scale may increase exponentially under lots of circumstances. This can cause memory bottleneck. Bar-Yosseff et al. (Bar-Yosseff et al., 2004, Bar-Yosseff et al., 2005) investigated the space complexity of XPath evaluation on streams and proved that for any algorithm  $A$  that evaluates a star free XPath query  $Q$  on an XML streaming document  $D$ , the minimum bits of space that  $A$  needs to use can be specified as the *concurrency lower bound*, denoted as  $\Omega(\text{CONCUR}(D, Q))$ . This lower bound is defined on the concept of a *concurrency*. As shown in Figure 1, document  $D$  is represented as a stream of 16 events called *time steps*. The concurrency of the document  $D$  with respect to query  $Q$  at step  $t \in [1, m]$  is the number of content-distinct nodes in  $D$  that are *alive* at step  $t$ . As shown in Figure 1, let  $Q = a[p]/b[c]/e$ . At step 14, two  $e$  elements are alive. The first is at step 3 because whether it will be selected depends on whether its  $a$  grandparent will have a  $p$  child. The second is at step 13, because whether it will be selected depends on whether its  $b$  parent will have a  $c$  child and its  $a$  grandparent will have a  $p$  child. So the concurrency at step 14 is 2. The

---

Copyright © 2008, Australian Computer Society, Inc. This paper appeared at the Nineteenth Australasian Database Conference (ADC2008), Wollongong, Australia, January 2008. Conferences in Research and Practice in Information Technology, Vol. 75. Alan Fekete and Xuemin Lin, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

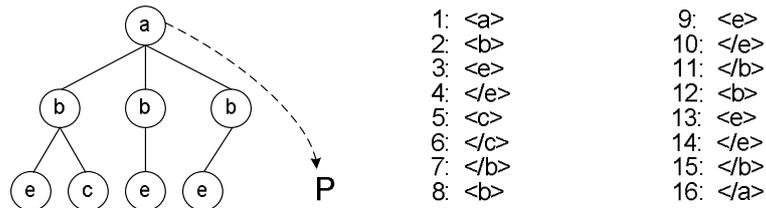


Figure 1 Concurrency of  $D$  w.r.t.  $Q=a[p]/b[c]/e$

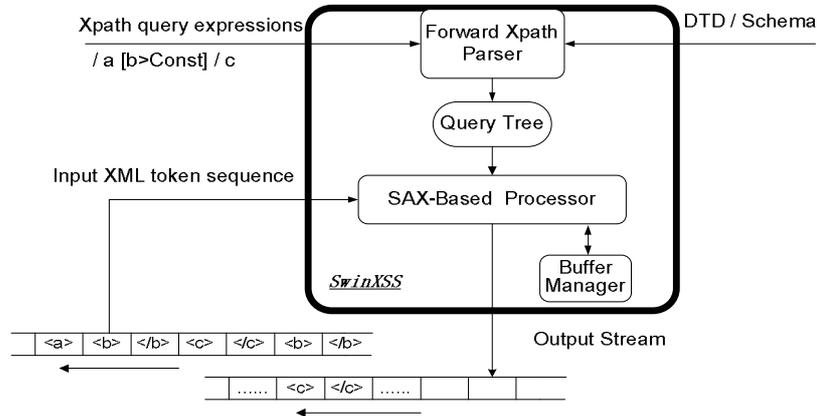


Figure 2 Architecture of SwinXSS

document concurrency of  $D$  w.r.t.  $Q$ , denoted as  $CONCUR(D,Q)$ , is the maximum concurrency over all steps  $t \in [1,m]$ . For example,  $CONCUR(D,Q)$  in Figure 1 is 2. The concurrency lower bound is suitable for single variable predicate queries. For queries with a multi-variable predicate, the dominance lower bound is defined in (Bar-Yossef et al., 2005). It is simple to verify that if  $Q$  is a non-predicate query,  $CONCUR(D,Q)$  is 1. However, for queries with single predicate, it is easy to construct documents with arbitrarily large concurrency.

Recently, utilizing semantic information to optimize query evaluation known as semantic query optimization (SQO) has generated promising results in XML query processing (Su et al., 2004, Su et al., 2005). The goal of SQO is to trim a query tree such that some evaluation effort can be saved if it would not return results. In this paper, we focus on utilizing semantic information for buffer reduction. Bear in mind the concurrency lower bound that limits the performance of any algorithm, this paper is motivated to answer the question whether this lower bound can be broken when semantic information is taken into account.

The main contributions of this paper are two fold: Firstly, aiming to break the concurrency lower bound and to reduce buffer space consumption, we design several semantic optimization rules and algorithms based on these rules. Secondly, we conduct an empirical study showing that our semantic buffer reduction algorithms can break the concurrency lower bound and outperform significantly over the lower bound algorithm that does not consider semantic information.

The rest of the paper is organized as follows. In Section 2, we briefly introduce a SAX-based query evaluation system that is built for this empirical study. The query evaluation algorithm that consumes buffer

space in line with the concurrency lower bound is designed. In Section 3, we further analyze the concurrency lower bound for obtaining guidelines for designing semantic rules. It follows with the algorithms that incorporate the semantic rules into the lower bound algorithm. In Section 4, we show experiment results for comparing the algorithms using semantic rules with the concurrency lower bound algorithm. Section 5 concludes the paper.

## 2 SAX based Query Evaluation System

To enable the empirical study, we build a SAX-based query evaluation system over XML document streams called Swinburne XML Stream System (SwinXSS). The architecture of SwinXSS is shown in Figure 2. SwinXSS implements a subset of XPath 2.0 called *Forward XPath* similar to (Bar-Yossef et al., 2005). The Forward XPath parser takes a query as well as semantic information as inputs and generates a query tree. The SAX-based stream processor then evaluates the generated query tree on an input XML data stream and generates an output stream. It communicates with the buffer manager for effective buffer management. The SAX-based processor relies on a SAX parser that pushes out events when it encounters *start-tags*, *end-tags*, etc. and activates the corresponding event handlers. Two event handlers that are most relevant to this study are *startElement* and *endElement* that are activated to handle start-tag and end-tag events respectively.

### 2.1 Query Evaluation Algorithm with Concurrency Lower Bound

In SwinXSS, a query is first transformed into a query tree. The leftmost path of a query tree is called the *main path* of the query and the lowest node in this path is

**Function startElement(e)**


---

```

1. if (!stackFlag) { // stackFlag = false
2.   if (qNode.leftChild = e) {
3.     qNode = qNode.getLeftChild();
4.     if (qNode = outputNode) output(e);
5.     if (qNode.leftChild.rightChild != null) {
6.       stackFlag = true; entryNode = qNode;
7.       stackNode = qNode.getLeftChild();
8.     }
9.   }
10. }
11. else { // stackFlag = true
12.   if (!predicateFlag) { // predicateFlag = false
13.     if (qNode.leftChild = e) {
14.       Stack.push(e);
15.       qNode = qNode.getLeftChild();
16.       if (qNode = outputNode) {concur++;
17.         if (concur > concurLB) concurLB++;}
18.     }
19.     if (qNode.rightChild = e) {
20.       qNode = qNode.getRightChild();
21.       predicateFlag = true; predicateNode = e;
22.     }
23.   } else { // predicateFlag = true
24.     if(qNode.leftChild = e)
25.       qNode = qNode.getLeftChild();
26.     if(qNode.rightChild = e)
27.       qNode = qNode.getRightChild();
28.   }
29. }

```

---

called the *query output node*. Other paths of the query tree are called the *predicate paths*. To compute the concurrency lower bound, we develop a best effort algorithm that delivers or discards query output nodes whenever all the predicates defined in the query are evaluated such that the number of *live* elements of the output node is the lowest. The algorithm works by processing the stream of SAX startElement and endElement events for each node of the query tree. It is mainly implemented as two functions startElement(e) and endElement(e) where *e* is the element tag event. We simply treat it as an element. In the real implementation, we also have content event handler that deals with the buffering and outputting of the content of the output node elements.

In the algorithm, we use a *stack* to buffer the elements for the query output node and other main path nodes that are necessary for the query evaluation. *stackNode* is used to record the query node that starts to use the stack and *entryNode* the parent node of *stackNode*, *predicateNode* is used to record the root node of any predicate subtree. The initial values of these variables are all set to “null”. *outputNode* is used to record the query output node. *stackFlag* marks that the stack is being used and *PredicateFlag* marks that predicates are being evaluated, both having “false” as the initial value. Each query node *qNode* may have a *parent*, *leftChild*, and *rightChild* and

**Function endElement(e)**


---

```

1. if (qNode = e) {
2.   if (!stackFlag) {qNode = qNode.getParent();
3.     qNode.leftCondition = true;}
4.   else { // stackFlag = true
5.     if (!predicateFlag) { // predicateFlag = false
6.       checkedResult = checkLeftRight(qNode);
7.       reset(qNode);
8.     }
9.     if (checkedResult) {
10.      // both leftCondition and rightCondition are true
11.      if (qNode = stackNode) {
12.        while (Stack.size != 0) {
13.          t = Stack.pop();
14.          if (t = outputNode) {output(t); concur -- ;}
15.        }
16.      }
17.      if (qNode = entryNode) {
18.        stackNode = null; stackFlag = false;}
19.      qNode = qNode.getParent();
20.      qNode.leftCondition = true;
21.      cNode = qNode;
22.      while (checkRight(cNode)) {
23.        if (cNode = stackNode) {
24.          do {t = Stack.pop();
25.            if (t = outputNode) {output(t); concur -- ;}
26.          } until (t = e);
27.        }
28.        else cNode = cNode.getParent();
29.      }
30.    }
31.  }
32.  else { // either leftCondition or rightCondition is false
33.    if (qNode = entryNode) {
34.      entryNode=null; stackFlag = false;}
35.    else {
36.      do {t = Stack.pop(); if (t = outputNode) concur --;}
37.      }until (t = e);
38.      qNode = qNode.getParent();
39.    }
40.  }
41. } else { // predicateFlag = true
42.   checkedResult = checkLeftRight(qNode);
43.   reset(qNode);
44.   if (checkedResult ^ (predicateNode = e)) {
45.     predicateFlag = false; predicateNode = null;
46.   }
47.   qNode = qNode.getParent();
48.   if (qNode.leftChild = e) qNode.leftCondition = true;
49.   if (qNode.rightChild = e)
50.     qNode.rightCondition = true;
51. }

```

---

use *leftCondition* and *rightCondition* to record the predicate evaluation results from its leftChild and rightChild, respectively. The initial value of *qNode* takes the root node of the query tree as *leftChild* and null as

*rightChild*.

In *startElement()*, we first process start-tags of those query nodes in the main path up to *stackNode* which has predicates to be evaluated (Lines 1-10). In case there is no predicate at all in the query tree, output the elements for *outputNode* immediately (Line 4). If the start-tag matches a node in the main path below *entryNode*, it is pushed in the stack for predicate evaluation later (Lines 13-17). To calculate the concurrency lower bound defined in (Bar-Yossef et al., 2005), we only count the query output node. We use *concur* to record the concurrency of current time step and *concurLB* to record the maximum concurrency up to now. After the stream is processed, *concurLB* yields the document concurrency from which the lower bound can be obtained (Line 16). If the start-tag matches a predicate node, preparations will be made (Lines 18-21) and predicate evaluation will follow (Lines 24-25).

In *endElement()*, Line 2 is used to process an end-tag that matches a node above *stackNode*. Lines 4-31 are used to process an end-tag that matches a node below *entryNode* in the *main path* while Lines 32-40 are used to process an end-tag that matches a node in a *predicate path*. Given a matching node *qNode*, the function *checkLeftRight(qNode)* checks whether both its *rightCondition* and *leftCondition* are true or not (Line 5). For the case of true, we check if *qNode* is the *stackNode*, and if so, we empty the stack and output elements if it matches *outputNode* and adjust current concurrency accordingly (Lines 7-12). We take an ***eager predicate evaluation approach*** that measures the concurrency lower bound exactly. Whenever *qNode* is evaluated to be true from both *leftChild* and *rightChild* in Line 5, we check if all its ancestors up to *stackNode* are also evaluated to be true from its *rightChild* by the function *checkRight()*. If so, we immediately pop up the stack to the element that matches *qNode*, *output* all elements that match *outputNode* and adjust the current concurrency as well (Lines 16-22). Similarly, whenever *qNode* is evaluated to be false from either *leftChild* or *rightChild*, we also immediately pop up the stack to the element that matches *qNode*, *discard* all elements in the stack including those match *outputNode*, and adjust the current concurrency (Line 27). In other words, we keep *concur* and hence *concurLB* as low as possible and this algorithm reflects the concurrency lower bound calculation.

### 3 Semantic Buffer Reduction

Given that buffering may constitute a major memory bottleneck on one hand and space complexity measured as concurrency lower bound has been theoretically proved as the limit of any algorithm can achieve, can we by any means break this bound? In this section, we aim to give a positive answer to the question by exploring semantic information and use it for buffer reduction.

#### 3.1 Analysis of Concurrency Lower Bound

From the algorithm presented in Section 2, we can define three states for a query output element being processed: *live*, *selected*, and *discarded* where *selected* and *discarded* states are certain for the element to be outputted or ignored, respectively while a *live* state is

uncertain and buffer is required for the element with a live state. From the algorithm, we keep the number of *live* output elements as low as possible so the concurrency lower bound is achieved. We denote the document concurrency  $\text{CONCUR}(D,Q)=f(l)$  where  $l$  is the deepest layer an output element may appear in the stream  $D$ . Let  $\text{MAX}(k)$  be the maximum cardinality for all elements at layer  $k$ , we have the following.

$$\sum_{k=1}^l \prod_{i=1}^k 1 \leq f(l) \leq \sum_{k=1}^l \prod_{i=1}^k \text{MAX}(k)$$

The left side occurs when there is no predicate at all or the states of all output elements can be immediately determined as either selected or discarded upon their arrivals. In such case, no buffer is needed at all.  $\text{CONCUR}(D,Q)$  is calculated in a way that whenever all related predicates for an output element have been evaluated, action is taken immediately to either *output* or *discard* the element so that  $f(l)$  is as close to the left side as possible. However, the output element has to be buffered if those predicates are not yet evaluated so the *live* state of the element can not be converted to either *selected* or *discarded* at the time. If we can take the advantage of semantic information and make the evaluation of some predicates early, we can change the live state of the output element early. In other words, we can break the lower bound!

#### 3.2 Semantic Rules for Buffer Reductions

With the above analysis as guidelines, we explore useful constraints from schema in a DTD or XML Schema and design semantic rules to use them for buffer reduction.

##### Rule 1: Predicate After Rule

It is easy to find in a schema the appearing order between those nodes in the main path including the output node and those in predicates. Actually this information is especially important because we want to evaluate the predicates early such that the elements for the output node can go through early. Given a node  $v$  in the main path of a query tree, if from schema we know that the elements of its right child  $p$  always arrive after the elements of its left child  $a$ , we may apply for the *Predicate After Rule* denoted as  $\text{PREDAFTER}(\text{Child}(v)=a, \text{Child}(v)=p)$  for buffer reduction. This rule states that each  $a$  element arrives before any  $p$  element. If there exists a constraint  $f(a,p)$  which becomes true after the arrival of certain number of  $a$  elements and this change triggers that the predicate on  $p$  also becomes true, then the previously buffered  $a$  elements under  $v$  and subsequently arriving  $a$  elements can be outputted immediately. For example, in a stock market, ordinary users can open as many as 5 windows to observe the market, but a VIP user can open as many windows as he or she wants. If  $\text{PREDAFTER}(\text{Child}(user)=window, \text{Child}(user)=VIP)$  and the query is  $/market/user[VIP]/window$ , then once the 6<sup>th</sup> window arrives for a user, we can immediately output the buffered 5 windows and the current window and the subsequent windows for the user before the start-tag of *VIP* arrives. However, the lower bound algorithm will

have to wait until the either start-tag of *VIP* or *user* arrives.

### Rule 2: Predicate Ahead Rule

Similarly for a node  $v$  in the main path of a query tree, if from schema we know that its right child  $p$  is before its left child  $a$ , we may apply for the *Predicate Ahead Rule* denoted as  $\text{PREDAHEAD}(\text{Child}(v)=a, \text{Child}(v)=p)$ . This rule is especially important to immediately dump *live* output elements which will eventually be discarded. For the previous example, if the query is the same but the rule is changed from  $\text{PREDAFTER}$  to  $\text{PREDAHEAD}$ , then we do not need to buffer window elements at all. If *VIP* does appear for a user, all *window* elements arriving later will be immediately outputted; otherwise, they will be discarded. For the latter, the lower bound algorithm has to buffer all the window elements until the end-tag of *user* element arrives.

### Rule 3: Maximum Cardinality Rule

If knowing that one  $v$  element has at most  $n$  elements for child node  $e$ , we may apply for the *Maximum Cardinality Rule* denoted as  $\text{MAXI}(\text{Child}(v)=e, n)$ . XML Schema provides *maxOccurs* to specify this information. If we have  $\text{MAXI}(\text{Child}(\text{user})=\text{interest}, 4)$  and the query is */market/user[interest='golf']/stock*. The 4<sup>th</sup> end-tag of *interest* will enable us to discard all buffered window elements and future arriving window elements of the user immediately. However, this cannot be done by the lower bound algorithm.

### Rule 4: Co-exist Rule

From cardinality constraints defined in a schema, we may infer the coexistence of a pair of elements  $a$  and  $b$  under  $v$ , denoted as  $\text{COEX}(\text{Child}(v)=a, \text{Child}(v)=b)$ . For example, if we have  $\text{COEX}(\text{child}(\text{user})=\text{VIP}, \text{child}(\text{user})=\text{vroom})$ , and the query */market/user[VIP]/vroom*, we can immediately output *vroom* elements with no need to wait and check *VIP*. Similarly we may have the exclusive rule  $\text{EXC}(\text{Child}(v)=a, \text{Child}(v)=b)$ , which means that either  $a$  or  $b$  is a child element of  $v$  but not both. The query in the form of */v[a]/b*, will not output any  $b$ .

## 3.3 Incorporation of Semantic Rules into Lower Bound Algorithm

If  $\text{PREDAFTER}(\text{Child}(v)=a, \text{Child}(v)=p)$  where  $v$ ,  $a$ , and  $p$  correspond to  $qNode$ ,  $qNode.leftChild$ , and  $qNode.rightChild$ , respectively, and there exists constraint  $f(a,p)$  between  $a$  and  $p$ , we add Lines  $a-n$  between Line 13 and Line 14 in the *startElement()*. If  $f(a,p)$  becomes true after current  $a$  element arrives (Line a), we infer that the predicate will be evaluated to be true and thus no need to be evaluated (Line b). We then check all *rightChild* of those nodes up to *stackNode* and see if they are all true (Lines c-g). If so, we pop up the stack up to  $qNode$  and output elements that match *outputNode* and adjust the current concurrency (Lines h-m). After that we continue with the processing of the arrived  $a$  element.

---

```

a) If (PREDAFTER(qNode.leftChild, qNode.rightChild) ^
    f(qNode.leftChild, qNode.rightChild)) {
b)   qNode.rightCondition = true;
c)   cNode = qNode;
d)   while !(cNode=stackNode) {
e)     cNode=cNode.getParent();
f)     if !checkRight(cNode) skip;
g)   }
h)   if (checkRight(cNode) {
i)     while (!Stack.top()==qNode) {
j)       t=Stack.pop();
k)       if (t = outputNode) {output(t); concur--};
l)     }
m)   }
n) }

```

---

If  $\text{PREDAHEAD}(\text{Child}(v)=p, \text{Child}(v)=a)$  where  $v$ ,  $a$  and  $p$  correspond to  $qNode$ ,  $qNode.leftChild$  and  $qNode.rightChild$ , respectively, we add Lines  $o-r$  also between Line 13 and Line 14 in *startElement()*. The arrival of the first start-tag of  $a$  symbolizes the end of all  $p$  elements. If we know all  $p$  elements are evaluated to be false by *checkRight()*, we pop up the current top node in the stack which is  $qNode$ . Then we set  $qNode$  to its parent node to bypass the processing of the subtree rooted with  $qNode$ . If the checking in Line o fails, we continue with Lines 14-16 in the original algorithm (Line s).

---

```

o) if (PREDAHEAD(qNode.leftChild, qNode.rightChild) ^
    !checkRight(qNode)) {
p)   Stack.pop();
q)   qNode=qNode.getParent();
r) }
s) else {Lines 14 – 16}

```

---

The treatment of the Maximum Cardinality rule is similar to that of the Predicate After Rule. Instead of checking  $f(a,p)$  in Line a, we count the number of arriving  $a$  elements to see if it reaches the maximum cardinality.

The treatment of the Co-exist rule is also similar, unlike that of the Maximum Cardinality rule, we do not need to check extra condition.

## 3.4 Example Run

We demonstrate the optimized algorithm by an example. Experiment results will be shown in Section 4. As shown in Figure 3, we have  $Q = a[p]/b[m[x > \text{const1}]/n < \text{const2}]/c$ , which could be normalized into query expression  $Q = a[p]/b[m/x > \text{Const1} \ \&\& \ m/n < \text{Const2}]/c$ , and our algorithm treats above two expressions as equal queries. In brief, a recursive nested predicate in a query can be computed as non-recursive one. The tree structure representation of the query can be seen in Figure 3(a). The light nodes are elements on the main path, and dark nodes are predicate nodes. The incoming instance document sequence  $D$  is noted as SAX events in Figure 3(b), the query will select qualified  $c$  elements. The semantic information belonging to query node  $b$  is  $\langle !ELEMENT \ b(m^*, \ c+) \rangle$ , of which the structure and predicate sequence can determine the optimization. From Figure 3(a), we know node  $b$  carries a predicate and this predicate will always be evaluated before the arrival of the first  $c$  element under current  $b$  element. Another semantic information for query node  $a$  is

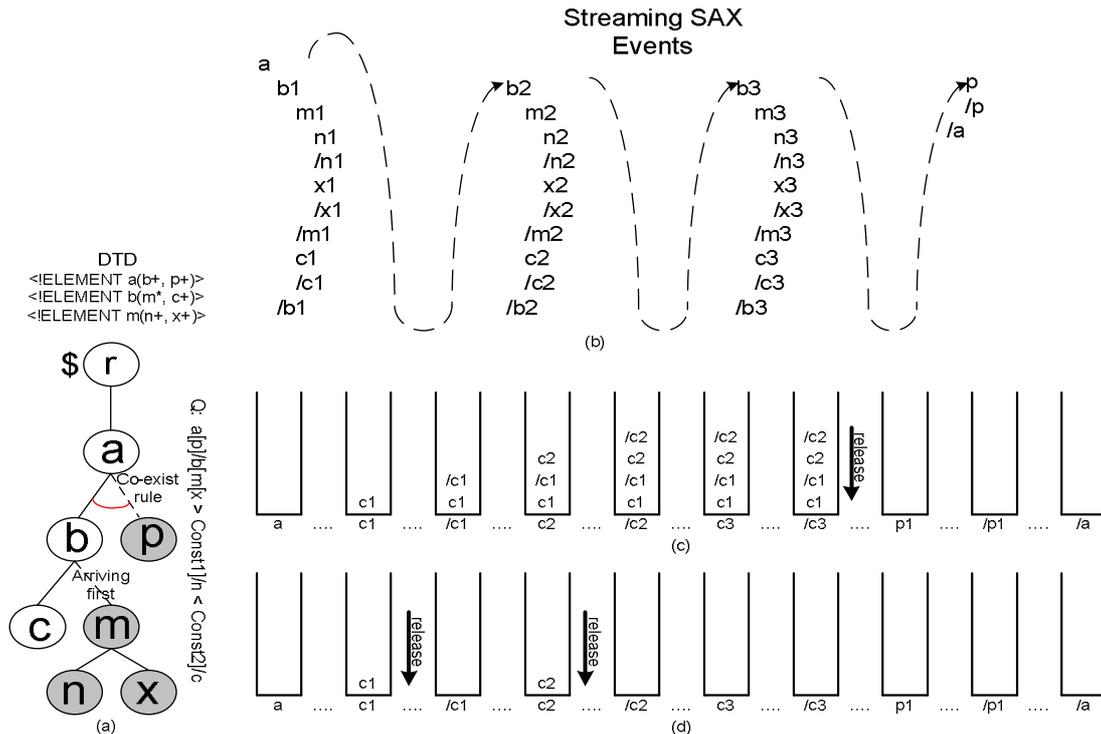


Figure 3 Buffer management optimization

<!ELEMENT a(b+, p+)>, which clarifies that the appearance of *b* or *p* will determine the appearance of each other.

In Figure 3(b), we give an example XML stream document for the query algorithm demonstration. The root element of the whole document is element *a*. From the root, along the arrow direction, the incoming XML stream comes to an end tag '/a'. With the reference to this flow, in Figure 3(c) and Figure 3(d) we demonstrate the detailed steps of buffer operation.

Without the help of semantic information, the basic algorithm will store most of *c*, which can be seen in buffer table of Figure 3(c). The basic algorithm is developed according to the concurrency lower bound. Therefore, all the live *c* elements that belong to the subtree of the current *a* element have to be stored. The reason is that without the confirmation of arriving of *p*, the query processor can not determine whether all the buffered *c1*, *c2* elements are qualified for the query expression. Predicate *p* is placed at the high level of the query tree, even the predicate belonging to element *b* is qualified, the state of elements *c1* and *c2* will still keep 'live' unless *p* is also evaluated. With respect to predicate expression belonging to query node *b*, because the number of element *m* is not clear, if there is no arrival of the end tag '</b>' and the qualification information of the predicate belonging to query node *b*, elements *c1* and *c2* need to be buffered. The processing of predicate under *b* is as follows: as soon as an element *m* is encountered, the expected elements will be set to *x* and *n*. If the next incoming element is one of the expected elements, a Boolean value will be set for it. The similar explanation can be used to describe the processing of predicate under

element *a*.

In Figure 3(a), DTD <!ELEMENT b(m\*, c+)> shows, to any *b*, all possible *m* elements need to arrive ahead all the *c* elements. To any *a* element, <!ELEMENT a(b+, p+)> shows that when the query processor encounters an element *b*, there must exist a qualified element *p*. The above analysis shows that we can treat the match of query node *a* as a non-predicate query evaluation. With the above semantic information, we can optimize the usage of buffer to a linear level. As shown in Figure 3(d), because the semantic information helps the query processor change the query with predicate *p* into linear one, it will not cause any buffer usage here. Consequently, with the arrival of 'c1' element, if under current *b* there is still no element *m* i.e., satisfying the predicate, the whole buffer could be emptied because of no suitable predicate for current element *b*. On the other hand, if under current *b* there is qualified element *m* satisfying the predicate, all the buffered 'c1' or 'c2' will be output. By the comparison of buffers in Figure 3(c) and Figure 3(d), the algorithm using information from DTD cuts the buffer space significantly.

According to the theoretical concurrency lower bound,  $\Omega(\text{CONCUR}(D,Q))$  bits of space is unavoidable. Normally, the  $\text{CONCUR}(D,Q)$  is the repetitive frequency of element *c*. In this example, the  $\text{CONCUR}(D,Q)=n$  where *n* is the maximum number of *c* under any *b* and the lower bound of the algorithm is  $\Omega(n)$ . But when the DTD information is used, the buffer space complexity can be dramatically reduced. In this example, the actual  $\text{CONCUR}(D,Q)$  of elements in the buffer during the evaluation becomes a constant 1 which is the linear query lower bound.

Query	Forward XPath Expressions
Q1	<code>/site/regions/asia/item/name</code>
Q2	<code>/site/people/person[profile/business]/watches/watch</code>
Q3	<code>/site/regions/africa/item/mailbox/mail[date&gt;2002]/text/keyword</code>
Q4	<code>/site/regions/africa/item[incategory="category18"]/mailbox/mail/text/keyword</code>
Q5	<code>/site/regions/africa/item[shipping]/description/parlist/listitem/text/keyword</code>
Q6	<code>/site[people]/regions/africa/item[description/parlist/listitem/text/keyword]/mailbox/mail[date&gt;2001]/text/keyword</code>

Figure 4 Test queries

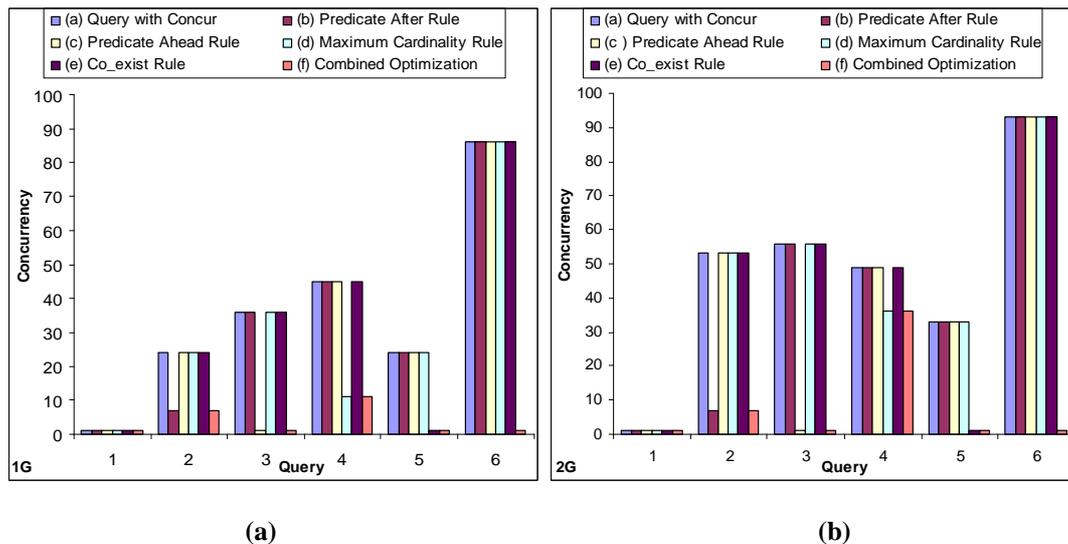


Figure 5 Maximum buffer scale for 1GB and 2GB XML dataset

## 4 Experiments

We implement the concurrency lower bound algorithm and the algorithms incorporating all the semantic rules in *SwinXSS* using Java. Experiments are conducted on an Intel P4 3GHz PC with 512 MB memory.

### 4.1 Data and Queries

We generate documents using the XMark document generating tool<sup>1</sup> with the XML DTD *auction.dtd* as input. We then design 6 queries in Figure 4 that are used to test the effectiveness of each semantic rule. We use bold font to exist rules. The purpose of Q6 is to verify the combined effects when all the semantic rules are applied together. For each of Q2-Q4, we assume that the corresponding rule is applicable. Furthermore,  $\text{count}(\text{watch}) > 6 \rightarrow \text{business}$  holds for Q2 and  $\text{MAXI}(\text{child}(\text{item})=\text{incategory}, 3)$  for Q4.

### 4.2 Comparison of Maximum Buffer Scale

We evaluate all the above queries on the generated documents of 1GB and 2GB in size, respectively. The experimental aim is to compare the peak values of buffer

scales before and after the deployment of semantic rules. Figure 5 (a) and Figure 5 (b) show the experimental results for evaluating Q1 – Q6, on the 1G and 2G documents. The 6 bars from left to right stand for the results for the lower bound algorithm, the individual algorithms for Rules 1-4, and the algorithm for applying all the rules for combined optimization, respectively.

For Q1, we can see that the document concurrency for all algorithms is 1 for both 1G and 2G documents because there is no predicate. For Q3 and Q5, we can see that the magic effect of applying the Predicate Ahead rule and the Co-exist rule, which make the document concurrency reduced to 1 for both 1G and 2G documents while the lower bound algorithm can achieve 36 and 24 in 1G document for Q3 and Q5, and 56 and 33 in 2G document for Q3 and Q5. For Q2, the document concurrency only depends on the constraint between person's two descendant child elements *business* and *watch*, i.e.,  $\text{count}(\text{watch}) > 6 \rightarrow \text{business}$ . So the document concurrency is 7 for both 1G and 2G documents. For Q4, the reduction effect is highly related to the specific content of document, which can be seen by comparing the document concurrency achieved for 1G document with that for 2G document. The former is better than the latter. Obviously, for Q2-Q4, the algorithm for combined optimization takes

<sup>1</sup> <http://monetdb.cwi.nl/xml/index.html>

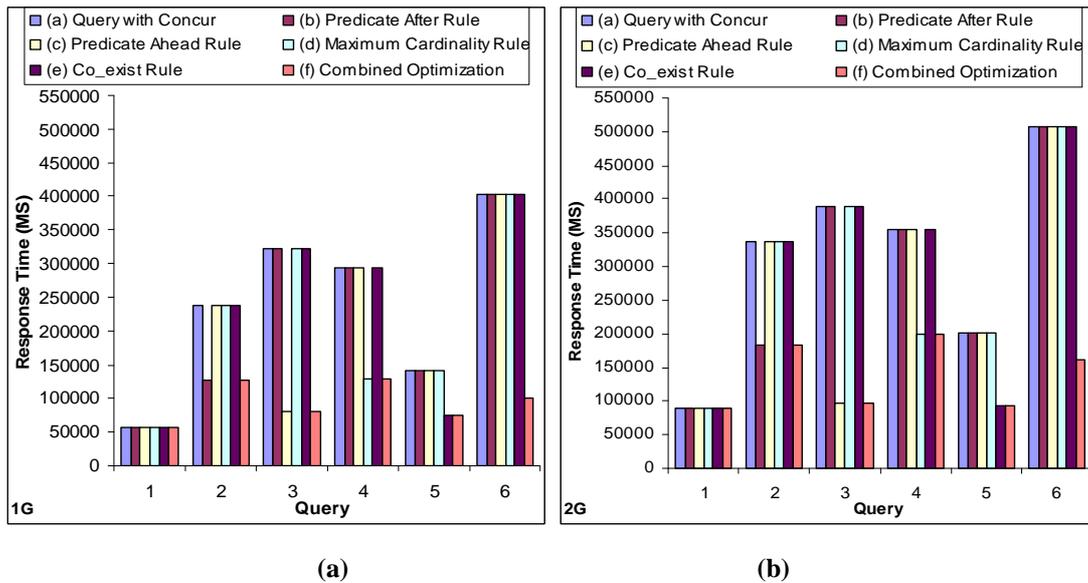


Figure 6 Response time for 1GB and 2GB XML dataset

the best document concurrency of those algorithms that apply the individual rules. The collective effect of buffer reduction is demonstrated in Q6, where each algorithm that applies individual rule does no reduction while the combined optimization algorithm performs perfectly. This is because there are three predicates in Q6. The selected state of an output element keyword depends on three ancestor elements site, item and mail. The application of each individual rule may not pre-determine the predicates of all three elements.

Figure 5 tells us when there exist predicates in a query and useful semantic information the lower bound can be broken by the algorithms that apply semantic rules. We can see that the combined optimization algorithm always outperforms the lower bound algorithm, which is consistent with our expectation.

### 4.3 Comparison of Response Time

Figure 6 shows the experimental results for execution time needed for evaluating Q1-Q6 using 6 different algorithms. Because of the savings in buffer processing, the algorithms using semantic rules outperform the lower bound algorithm. From the figure, we find that the combined optimization algorithm and the algorithms that apply the Predicate Ahead and Co-exit rules perform better than the algorithms that apply the Predicate After rule and Maximum Cardinality rules because the former three algorithms use fewer buffer processing time. In general, the reduction in response time is less than the reduction in buffer consumption because each algorithm has to scan the whole document and the only saving in time comes from the saving in buffer processing.

## 5 Conclusions

Query evaluation for data streams is basically main-memory based. Efficient buffer management is therefore fundamentally important for stream query

processing. An interesting work in query evaluation over XML streams is the theoretic proof of the concurrency lower bound that any algorithm cannot break. Through an empirical study, we showed that this lower bound can be broken if we take the advantage of semantic information available in the schema associated with the XML document. We developed a best effort algorithm that is in line with the concurrency lower bound. Then we explored several semantic rules for buffer reduction and incorporated them into the lower bound algorithm. Our experiment results showed that the algorithms incorporating semantic information significantly outperform the lower bound algorithm.

## 6 Acknowledgement

The work described in this paper was supported by grant from the Research Grant Council of the Hong Kong Special Administrative Region, China (CUHK418205).

## 7 References

- ALTINEL, M. & FRANKLIN, M. J. (2000) Efficient Filtering of XML Documents for Selective Dissemination of Information. *International Conference on Very Large Data Bases (VLDB)*. Cairo, Egypt, Morgan Kaufmann.
- BAR-YOSSEF, Z., FONTOURA, M. & JOSIFOVSKI, V. (2004) On the Memory Requirement of XPath Evaluation over XML Streams. *PODS*.
- BAR-YOSSEF, Z., FONTOURA, M. & JOSIFOVSKI, V. (2005) Buffering in Query Evaluation over XML Streams. *PODS*.
- BOSE, S. & FEGARAS, L. (2004) Data Stream Management for Historical XML Data. *SIGMOD*.

- FEGARAS, L., LEVINE, D. & BOSE, S. (2002) Query processing of Streamed XML Data. *CIKM*.
- GRAY, J. (2004) The Next Database Revolution. *SIGMOD*.
- JIAN, J., SU, H. & RUNDENSTEINER, E. A. (2003) Automaton Meets Query Algebra: Towards a Unified Model for XQuery Evaluation over XML Data Streams. *ER*.
- LUDASHER, B., MUKHOPADHYAY, P. & Y.PAPAKONSTANTINO (2002) A Transducer-Based XML Query Processor. International Conference on Very Large Data Bases (VLDB).
- PENG, F. & CHAWATHE, S. S. (2003) XPath Queries on Streaming Data. *SIGMOD*.
- SU, H., RUNDENSTEINER, E. A. & MANI, M. (2004) Semantic Query Optimization in an Automata-Algebra Combined XQuery Engine over XML Streams. International Conference on Very Large Data Bases (VLDB).
- SU, H., RUNDENSTEINER, E. A. & MANI, M. (2005) Semantic Query Optimization for XQuery over XML Stream. International Conference on Very Large Data Bases (VLDB).

