A Query System for XML Data Stream and its Semantics-based Buffer Reduction

Chi Yang

School of Computer Science & Software Engineering
The University of Western Australia, Perth, WA 6009, Australia
cyang@csse.uwa.edu.au

Chengfei Liu and Jianxin Li

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

Jeffrey Xu Yu

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

Junhu Wang

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

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 has 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 built a SAX-based XML stream query evaluation system and designed an algorithm that consumes buffers in line with the concurrency lower bound. After a further analysis of the lower bound, we designed several semantic rules for the purpose of breaking the lower bound and incorporated 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, Buffer Management, SAX
ACM Classification: H.2.4

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 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 (Babcock *et al*, 2002). 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. Some high performance query engines and systems (Peng *et al*, 2005; Diao *et al*, 2003) are developed. 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 an automaton-based algorithm is that it becomes difficult when XPath queries are 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-Youseff *et al* (2004) and Bar-Yossef *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 $\Omega(CONCUR(D,Q))$. 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 *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 and build a query platform for it based on our previous work. 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 (Yang *et al*, 2008).
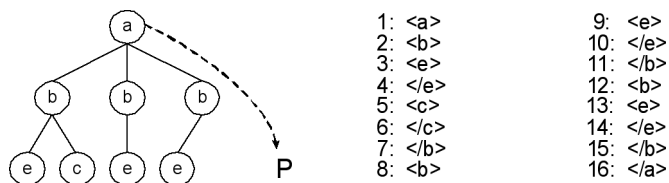

1: <a>	9: <e>
2: 	10: </e>
3: <e>	11:
4: </e>	12:
5: <c>	13: <e>
6: </c>	14: </e>
7: 	15:
8: <b>	16:

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

The main contributions of this paper are two fold: Firstly, aiming to break the concurrency lower bound and to reduce buffer space consumption, we designed several semantic optimization rules and algorithms based on these rules. With that, a SAX-based XML data stream query system is implemented. Secondly, we conducted 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. Furthermore, the query language parser and the query evaluation processor for that system will be discussed intensively. Then, our query evaluation algorithm that consumes buffer space in line with the concurrency lower bound is designed. In Section 3, we further analyze the currency lower bound for obtaining guidelines for designing semantic rules. It follows with the algorithms that incorporate the semantic rules into the lower bound algorithm. The correspondent example and analysis will be given to describe the algorithm efficiency. 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 built 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. The SwinXSS implements a subset of XPath 2.0 called Forward XPath similar to Bar-Yossef *et al* (2005). The Forward XPath parser takes a Forward XPath query as well as semantic information as inputs and generates a query tree. The semantic information mainly comes from the analysis of DTD and Schema. 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-tag*, *end-tag*, etc. and activates the corresponding event handlers. Two event handlers, *startElement* and *endElement* are most relevant to this study, because they are implemented and activated to handle
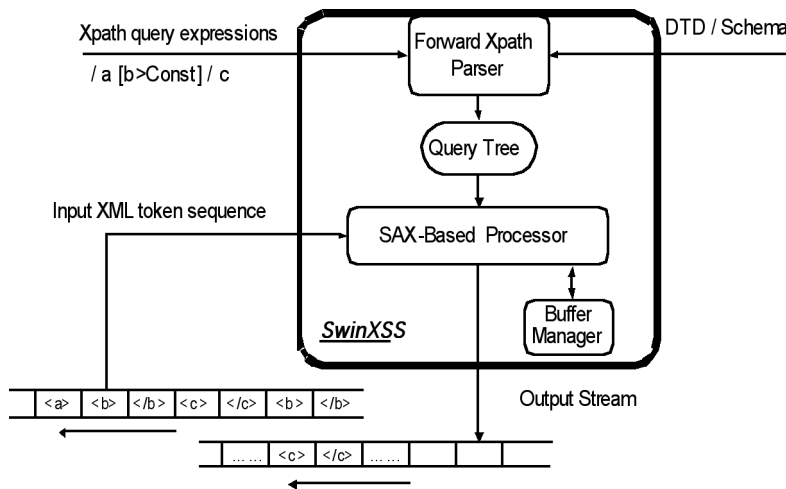

Figure 2: Architecture of SwinXSS

two types of common events, *start-tag* and *end-tag* events respectively in most of XML data streams. However, other important event handlers are also implemented in the processor to support the query syntax of the Forward XPath.

2.1 Restricted Forward XPath Query Parser

To process queries, a parser is developed in SwinXSS. It is difficult to support the full XPath for XML data stream. Similar to the work from Bar-Yossef *et al* (2005), we simplify the XPath2.0 to get Forward XPath, formally a conjunction of Univariate XPath, Subsumption-free XPath and Symmetric XPath. It supports the forward axes only.

SwinXSS implements a restricted Forward XPath. It supports multiple atomic univariate predicates and nested predicates within query expressions. However, it does not permit recursively defined elements, "*" and "//".

In SwinXSS, documents and Forward XPath queries are all represented as trees. We use u to represent the nodes from query expressions, use v to represent the node at instance level. We use D as a representation of a document and T as a tree structure representation of a document. For any $v \in T$, PATH(v) is the sequence of nodes on the path from the root to v. Child(v) is the set of child elements of the element v. ROOT(D) is the root of document tree T. Q is a query tree that consists of all the legal XML node names from an XPath expression. S is the set of all finite length strings of UCS (universal character set) characters.

In Forward XPath language, each node u in the query tree has the following properties:

- **AXIS(u):** Due to the restriction of Forward XPath, AXIS(u) takes child axis only.
- **LABEL(u):** Because we do not permit wildcards, LABEL(u) is from set Q.
- **PREDICATE(u):** PREDICATE(u) itself is a tree whose internal nodes are tagged by logical, comparison, arithmetic, or functional operators, and whose leaves are tagged by constants from S.

Based on all the constraints and definitions above, the table in Figure 3 shows the formal grammar of the restricted Forward XPath used in SwinXSS. In SwinXSS, the restricted Forward XPath parser decomposes query expressions into the grammar units defined in Figure 3. Then, the generated grammar units are organized to form a query tree. Every node of the query tree represents an XPath element and its relevant information. Each node consists of several fields: '*nodename*', '*operator*', '*noderule*', '*leftchild*', '*rightchild*', '*parent*', '*leftconditiont*', '*rightcondition*'. They are useful for matching elements from XML data streams during query evaluation. Their functions are listed below:

- *nodename:* to record the name of an element from the stream
- *nodecontent:* to record any constant from S in a query node
- *operator:* to record the operation on the current node if it exists

---

Path := /Step | /Step Path
Step := Element | Element"["Pred"]" | Func"("Element")"
Pred := Element | Element Oper Const |"!" Pred | Pred"&&" Pred |
Pred"||" Pred
Oper := "<" | "≤" |">"|"≥"|"≠"|"="
Const is any string from S
Func() is any predefined computation
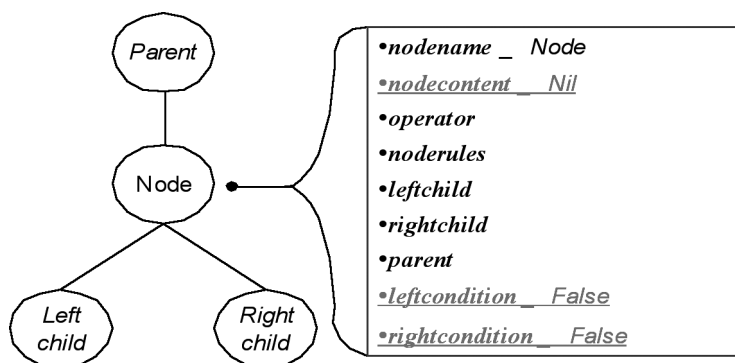

Figure 3: Restricted Forward XPath Grammar

Figure 4: Inner Representation of a Node in Query Tree

- *noderules:* to record the semantic rules applied on the current node if it exists
- *leftchild:* to record the left child of the current node if it exists
- *rightchild:* to record the right child of the current node if it exists
- *parent:* to record the parent of the current node if it exists
- *leftcondition:* a Boolean value to record the returned value from its left child if it exists
- *rightcondition:* a Boolean value to record the returned value from its right child if it exists

Figure 4 shows how a query node can be correlated with other nodes using its inner fields to form a query tree. Based on the restriction of our proposed XPath, every query node has at most three direct correlated nodes which are its left child, right child and parent as shown on the left side of Figure 4. To keep this ancestor-descendent relationship, *leftchild*, *rightchild* and *parent* fields are used. All fields are static after the generation of the query tree, except for three underlined fields. However, these three underlined fields can be changed and reused by the different elements from XML streams during query evaluation. Specifically, *nodecontent* will record the information carried by different elements with the same *nodename* and will be set 'Nil' before the start of query evaluation. *leftcondition* and *rightcondition* will be reused by every Node element. Their initial values are set as 'False' before the start of query evaluation.

2.2 SAX-based Query Processor
In Figure 2, the SAX-based processor sits in the centre of the architecture and acts as a core unit in SwinXSS. It takes the streaming XML data and the query tree generated from the Forward XPath parser as input, fulfilling the task of matching nodes from the query tree with nodes from the incoming stream and buffer management, then pushes out the selected data as a result. SwinXSS uses SAX parser as the basic tools to divide the streaming XML data into XML grammar units.

SAX represents XML data as a sequence of events and pushes them to their registered content handlers through function callbacks. Because SAX parsers push out events in a broadcast fashion, it is efficient for parsing large XML documents using limited memory. The fact is that SAX acts as a basic XML processing platform for most of the advanced APIs. So, it is popular and wise to choose SAX to build up an XML stream processor. Hence, SwinXSS also uses a SAX based processor.

As a standard stands across different XML parsers, SAX APIs has a specific class to be implemented varying from parser to parser. But all those parsers have some important interfaces in common. They are listed as follows:
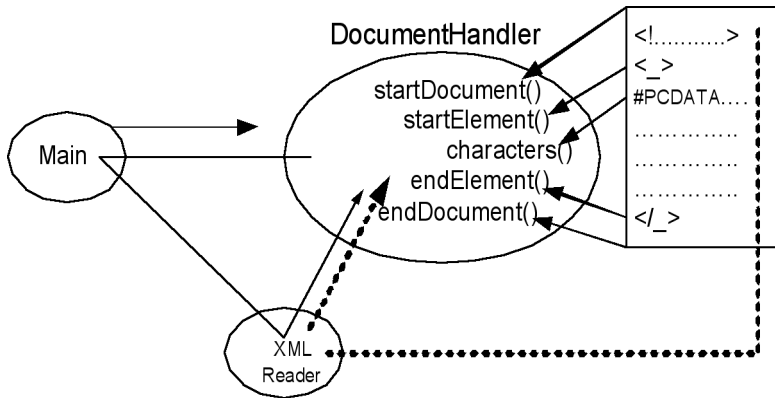
Figure 5: XML stream processor implementing SAX interface in SwinXSS

- **XMLReader interface:** An interface implemented by the user to create an object. The object acts as a parser to decompose incoming XML data streams.
- **ContentHandler interface:** An interface implemented by the user to define the methods for different event reactions. The application has to implement all the methods belonging to it.
- **DocumentHandler interface:** An interface has the similar function as "*ContentHandler interface*", but there is no need for the application to implement all the methods in it, because the object can be created by default.

In SwinXSS, an object created by XMLReader reads a document from beginning to end. At the same time, a pointer on the query tree moves back and forth to synchronize the query node matching. During this matching process, it may encounter start-tags, such as document start or element start tags.

All the matching start events will lead to the one step deeper in child axis. It can also encounter end-tags such as document end or element end tags which will conversely lead to pointer one step back in ancestor axis. All the grammar units such as text, comments, processing instructions and entities are treated as events. If an event is triggered, the XML Reader will call the corresponding method in the Documenthandler to make the reaction.

As shown in Figure 5, regardless of what the tag name is, when a start tag is encountered by the XMLReader parser, the startElement() method in the *DocumentHandler* is called. Our proposed algorithm implementing the startElement() function will process all the start tags. If the method body is null, the XMLReader object pushes out the start tag directly. When the #PCDATA is encountered, the chataters() function in DocumentHandler is invoked to process the values. Because we assume that in any mixed element, the content of that element will always appear as a whole, there is no need to wait for the end tag of the current element. When an end tag is encountered, the endElement() in DocumentHandler is called. Within the endElement() function, we encapsule all the remaining operations and processing belonging to the current element. So the correspondent algorithm within endElement() is relatively complicated and difficult to follow. The definations of startDocument() and endDocument() functions are quite similar to those of startElement() and endElement(). However, the startDocument() is mainly used for initializing global variables and external data structures.

2.3 Query Evaluation Algorithm with Concurrency Lower Bound

In SwinXSS, the leftmost path of a query tree is called the *main path* of the query and the lowest node in this path is called the *query output node*. Other paths of the query tree are called the *predicate paths*. To compute the concurrency lower bound, we developed 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 use a content event handler (*DocumentHandler*) and its methods to deal 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 *qNode* may have a *parent*, *leftChild*, and *rightChild* and 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

---

Function startElement(e)

```
1.   if (!stackFlag) { // stackFlag = false
2.     if (qNode.leftChild = e) { // the node is a main path node
3.       qNode = qNode.getLeftChild(); // move 1 step forward
4.       if (qNode = outputNode) output(e); // output a qualified query result
5.       if (qNode.leftChild.rightChild != null) { // encounter a node with predicate
6.         stackFlag = true; entryNode = qNode; // change status variables
7.         stackNode = qNode.getLeftChild(); // record entry node for buffering
8.       }
9.     }
10. }
11. else { // stackFlag = true
12.   if (!predicateFlag) { // predicateFlag = false
13.     if (qNode.leftChild = e) {
14.       Stack.push(e); // buffer an element
15.       qNode = qNode.getLeftChild(); // move 1 step forward
16.       if (qNode = outputNode) {concur++; if (concur > concurLB) concurLB++;} // record
          concurLB
```

```
17.      }
18.      if (qNode.rightChild = e) { // enter a non-main path branch
19.        qNode = qNode.getRightChild();
20.        predicateFlag = true; predicateNode = e;
21.      }
22.    }
23.    else { // predicateFlag = true
24.      if(qNode.leftChild = e) qNode = qNode.getLeftChild();
25.      if(qNode.rightChild = e) qNode = qNode.getRightChild();
26.    }
27. }
```

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 processing finishes, *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.

Function endElement(e)

```
1.   if (qNode = e) {
2.     if (!stackFlag) {qNode = qNode.getParent(); qNode.leftCondition = true;} // output directly
3.     else { // stackFlag = true
4.       if (!predicateFlag) { // predicateFlag = false
5.         checkedResult = checkLeftRight(qNode); reset(qNode); // prepare for moving back
6.         if (checkedResult) { // both leftCondition and rightCondition are true
7.           if (qNode = stackNode) {
8.             while (Stack.size != 0) {
9.               t = Stack.pop(); // release the buffered elements
10.              if (t = outputNode) {output(t); concur -- ;} // reduce the concurrency
```

```
11.        }
12.      }
13.      if (qNode = entryNode) {stackNode = null; stackFlag = false;}
14.      qNode = qNode.getParent(); // move 1 step backward
15.      qNode.leftCondition = true; // set the condition of the current node
16.      cNode = qNode; // record current value of qNode in temporal cNode
17.      while (checkRight(cNode)) { // if rightCondition is true
18.        if (cNode = stackNode) { // the condition for stack release is qualified
19.          do {t = Stack.pop(); if (t = outputNode) {output(t); concur -- ;}} until (t = e); //
             output result
20.        }
21.        else cNode = cNode.getParent(); // move 1 step backward
22.      }
23.    }
24.    else { // either leftCondition or rightCondition is false
25.      if (qNode = entryNode) {entryNode=null; stackFlag = false;}
26.      else {
27.        do {t = Stack.pop(); if (t = outputNode) concur --;}until (t = e); // clear the useless
           buffer
28.        qNode = qNode.getParent();
29.      }
30.    }
31.  }
32.  else { // predicateFlag = true
33.    checkedResult = checkLeftRight(qNode); reset(qNode);
34.    if (checkedResult ∧ (predicateNode = e)) { // return to the main path from a predicate
35.      predicateFlag = false; predicateNode = null; // reset variables for predicate nodes
36.    }
37.    qNode = qNode.getParent(); // move 1 step backward
38.    if (qNode.leftChild = e) qNode.leftCondition = true; // set leftCondition of current node
39.    if (qNode.rightChild = e) qNode.rightCondition = true; // set rightCondition of current node
40.  }
41. }
42. }
```

3. SEMANTIC BUFFER REDUCTION

Given that buffering may constitute a major memory bottleneck on one hand and space complexity measured as concurrency lower bound, it has been theoretically proved as the limit 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 output 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 CONCUR(D,Q)=$f(l)$ where l is the deepest layer an output element may appear in the stream D. Let $MAX(k)$ be the maximum cardinality for all elements at layer k, we have the following.

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

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. 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 cannot 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 PREDAFTER(Child(v)=a, 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 a 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 five windows to observe the market, but a VIP user can open as many windows as he or she wants. If PREDAFTER(Child(*user*)=*window*, Child(*user*)=*VIP*) and the query is /*market*/*user*[*VIP*]/*window*, then once the sixth window arrives for a user, we can immediately output the buffered five 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 PREDAHEAD(Child(v)=a, 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 PREDAFTER to 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 MAXI(Child(*v*)=*e*, *n*). XML Schema provides *maxOccurs* to specify this information. If we have MAXI(Child(*user*)=*interest*, 4) and the query is */market/user*[*interest*='golf']/*stock*. The fourth end-tag of *interest* and *interest*!='golf' will enable us to discard all buffered *stock* elements and future arriving *stock* 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 COEX(Child(*v*)=*a*, Child(*v*)=*b*). For example, if we have COEX(child(*user*)=*VIP*, child(*user*)=*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 EXC(Child(*v*)=*a*, 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 PREDAFTER(Child(*v*)=*a*, 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 PREDAHEAD(Child(*v*)=*p*,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 6, we have Q= a[p]/b[m[x> const1]/n < const2]/c, which could be normalized into query expression Q= a[p]/b[m/x > Const1 && m/n < Const2]/c, and our algorithm treats the above two expressions as equal queries. In brief, a recursive nested predicate in a query can be computed as a non-recursive one. The tree structure representation of the query can be seen in Figure 6(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 6(b), the query will select qualified *c* elements. The semantic information belonging to query node *b* is <!ELEMENT b(m*, c+)>, of which the structure and predicate sequence can determine the optimization. From Figure 6(a), we know node *b* carries a predicate and this predicate will always be evaluated before the arrival of the first *c* element under the current *b* element. Another semantic information for query node *a* is <!ELEMENT a(b+, p+)>, which clarifies that the appearance of *b* or *p* will determine the appearance of each other.

In Figure 6(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 6(c) and Figure 6(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 the buffer table of Figure 6(c). The basic algorithm is developed according to the concurrency lower bound. Therefore, all the live *c* elements that belong to the sub-tree of the current *a* element have to be stored. The reason is that without the confirmation of arriving of *p*, the query processor cannot 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 '' 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. A similar explanation can be used to describe the processing of predicate under element *a*.
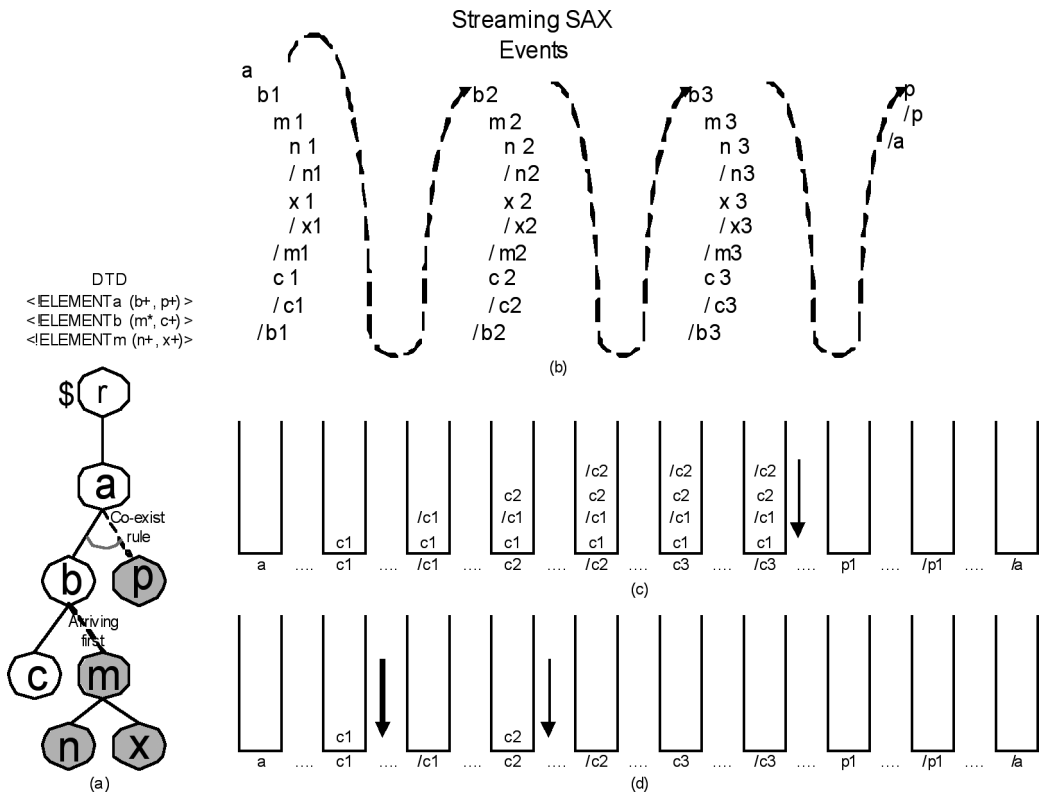

Figure 6: Buffer management optimization

In Figure 6(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 buffer usage to a linear level. As shown in Figure 6(d), because the semantic information helps the query processor change the query with predicate *p* into a 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 a qualified element *m* satisfying the predicate, all the buffered 'c1' or 'c2' will be output. By the comparison of buffers in Figure 6(c) and Figure 6(d), the algorithm using information from DTD cuts the buffer space significantly.

According to the theoretical concurrency lower bound, (CONCUR(D,Q)) bits of space is unavoidable. Normally, the CONCUR(D,Q) is the repetitive frequency of element *c*. In this example, the CONCUR(D,Q)=n where n is the maximum number of c under any *b* and the lower bound of the algorithm is (n). But when the DTD information is used, the buffer space complexity can be dramatically reduced. In this example, the actual 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	*/site/regions/asia/item/name*
Q2	*/site/people/**person**[profile/business]/watches/watch*
Q3	*/site/regions/africa/item/mailbox/mail[date>2002]/text/keyword*
Q4	*/site/regions/africa/**item**[incategory="category18"]/mailbox/mail/text/keyword*
Q5	*/site/regions/africa/**item**[shipping]/description/parlist/listitem/text/keyword*
Q6	*/site[people]/regions/africa/**item**[description/parlist/listitem/text/keyword]/mailbox /**mail**[date>2001]/text/keyword*

Figure 7: Test queries

4. EXPERIMENTS

We implement the concurrency lower bound algorithm and the algorithms incorporating all the semantic rules in *SwinXSS* in 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[1] with the XML DTD *auction.dtd* as input. In auction.dtd, the maximum depth of the generated documents is nine steps without taking recursive structures into consideration. This is useful to test the buffer usage effectively. The width of the document is also sufficient and structure is variable. In addition, based on the analysis of *auction.dtd*, we found all the necessary XML document instances being eligible to test our semantic rules.

We then designed six queries in Figure 7 that are used to test the effectiveness of each semantic rule. We used bold font to nodes where one or more rules are applied. 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, $count(watch)>6 \rightarrow business$ holds for Q2 and MAXI(child(*item*)=*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 experiment targets to compare the peak values of buffer scales before and after the deployment of semantic rules. Figure 8 (a) and Figure 8 (b) show the experimental results for evaluating Q1 – Q6, over the 1G and 2G documents. The six 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 one 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 one for both 1G and 2G documents while the lower bound algorithm can achieve 36 and 24 in a 1G document for Q3 and Q5, and 56 and 33 in a 2G document for Q3 and Q5. For Q2, the document concurrency only depends on the constraint between the person's two descendant child elements business and watch, i.e., count(watch)>6→business. So the document concurrency is

seven for both 1G and 2G documents. For Q4, the reduction effect is highly related to the specific content of the document, which can be seen by comparing the document concurrency achieved for a 1G document with that for a 2G document. The former is better than the latter. Obviously, for Q2-Q4, the algorithm for combined optimization takes 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 an 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 8 (a) and (b) tells us when there exist predicates in a query and useful semantic information, the lower bound can be broken by the algorithms that apply to semantic rules. Focusing on the buffer expenses of the lower bound algorithm and the algorithm applying all four semantic rules, we get two performance curves in Figure 8 (c) and (d). Whatever the data size is, 1G or 2G,
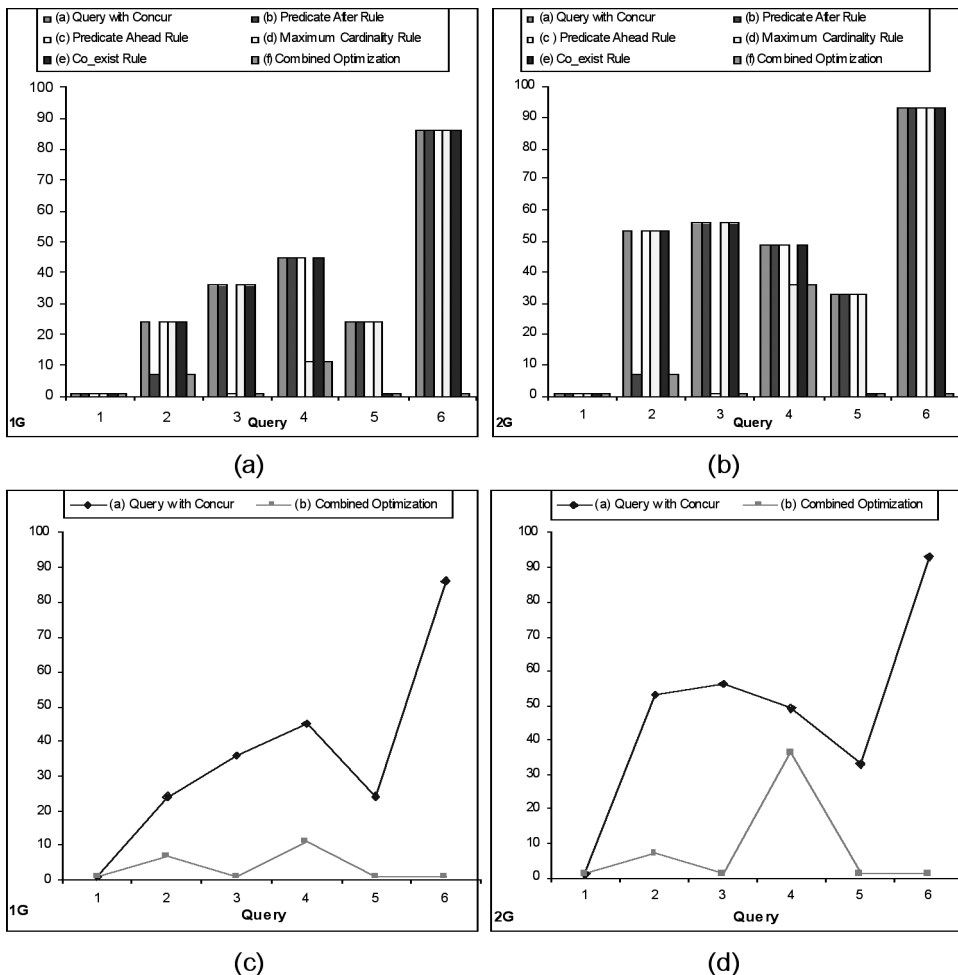

(a)

(b)

(c)

(d)

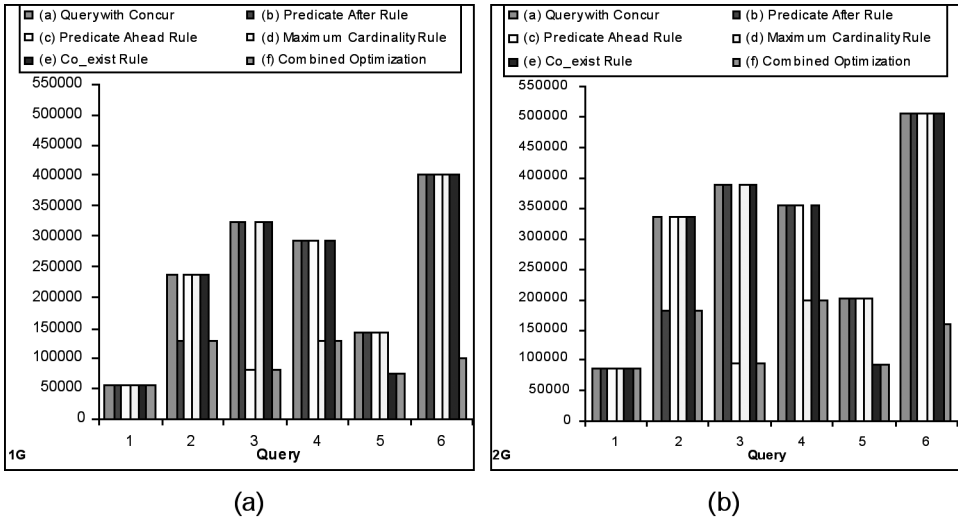**Figure 8: Maximum buffer size for 1GB and 2GB XML dataset**

Figure 9: Response time for 1GB and 2GB XML dataset

Combined Optimization curve runs totally below Query with Concur curve. It is clear that the combined optimization algorithm applying all of our proposed semantic rules outperforms the lower bound algorithm significantly, which is consistent with our expectation. Furthermore, the empirical study in Figure 8 (c) and Figure 8 (d) indicates that the more complicated the query is, the more optimized buffer size can be anticipated. To explain this, on one hand a more complicated query carries more semantic information which leads to a higher opportunity for optimization. On the other hand, the more complicated an original query is, a more significant buffer decrease can be observed after optimizing the original query into a linear query due to the comparison between exponential and linear increase.

4.3 Comparison of Response Time

Figure 9 shows the experimental results of execution time needed for evaluating Q1-Q6 using six different algorithms. Because of the savings in buffer processing, the algorithms using semantic rules outperform the lower bound algorithm. From Figure 9, 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 over 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 outperformed the lower bound algorithm.

6. ACKNOWLEDGEMENT

7. REFERENCES

ALTINEL, M. and FRANKLIN, M.J. (2000): Efficient filtering of XML documents for selective dissemination of information. *International Conference on Very Large Data Bases (VLDB)*. 53–64. Cairo, Egypt, Morgan Kaufmann.

BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R. and WIDOM, J. (2002): Models and issues in data stream system. In *The 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. 1–16.

BAR-YOSSEF, Z., FONTOURA, M. and JOSIFOVSKI, V. (2004): On the memory requirement of XPath evaluation over XML streams. *Symposium on Principles of Database Systems (PODS)*. 167–178.

BAR-YOSSEF, Z., FONTOURA, M. and JOSIFOVSKI, V. (2005): Buffering in query evaluation over XML streams. *Symposium on Principles of Database Systems (PODS)*. 216–227.

BOSE, S. and FEGARAS, L. (2004): Data stream management for historical XML data. *SIGMOD*. 239–250.

DIAO, Y., ALTINEL, M., FRANKLIN, M.J., ZHANG, H. and FISCHER, P. (2003): Path sharing and predicate evaluation for high performance XML filtering. *ACM Transaction on Database Systems (TODS)*. 28: 467–516.

FEGARAS, L., LEVINE, D. and BOSE, S. (2002): Query processing of Streamed XML Data. CIKM. pp: 126-133.

GRAY, J. (2004): The next database revolution. *SIGMOD*. 1–4.

JIAN, J., SU, H. and RUNDENSTEINER, E.A. (2003): Automaton meets query algebra: Towards a unified model for XQuery evaluation over XML data streams. *ER*. 172–185.

LUDASHER, B., MUKHOPADHYAY, P. and PAPAKONSTANTINOU, Y. (2002): A transducer-based XML query processor. *International Conference on Very Large Data Bases (VLDB)*. 227–238.

PENG, F. and CHAWATHE, S.S. (2003): XPath queries on streaming data. *SIGMOD*. 431–442.

PENG, F. and CHAWATHE, S.S. (2005): XSQ: A streaming XPath engine. *ACM Transaction on Database Systems (TODS)*. 30(2): 577–623.

SU, H., RUNDENSTEINER, E.A. and MANI, M. (2004): Semantic query optimization in an automata-algebra combined XQuery engine over XML streams. *International Conference on Very Large Data Bases (VLDB)*. 1293–1396.

SU, H., RUNDENSTEINER, E.A. and MANI, M. (2005): Semantic query optimization for XQuery over XML stream. *International Conference on Very Large Data Bases (VLDB)*. 277–288.

YANG, C., LIU, C., LI, J., YU, J. and WANG, J. (2008): Semantics based buffer reduction for queries over XML data streams. In *Proc. Nineteenth Australasian Database Conference (ADC)*, CRPIT. 75. 145–153.

BIOGRAPHICAL NOTES

Chi Yang received his BS in computer science from Shandong University, China, in 2004. He received his MS (by research) in computer science from Swinburne University of Technology, Melbourne, Australia, in 2007. Currently, Chi Yang is a full-time PhD student at the University of Western Australia, Perth, Australia. His major research interests include the XML data stream query processing and optimization, the scientific workflow and wireless sensor network (WSN) query systems.

Chi Yang

Chengfei Liu is currently a full professor and the head of the web and data engineering research group in the Centre for Complex Software Systems and Services, the Faculty of Information and Communication Technologies, Swinburne University of Technology, Melbourne, Australia. He received the BS, MS and PhD degrees from Nanjing University, China in 1983, 1985 and 1988, respectively, all in computer science. Prior to joining Swinburne in 2004, he taught at the University of South Australia and the University of
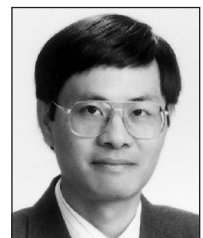
Chengfei Liu

Technology Sydney, and was a research scientist at the Cooperative Research Centre for Distributed Systems Technology (DSTC), Australia. He also held visiting positions at the Chinese University of Hong Kong, the University of Aizu in Japan, and IBM Silicon Valley Lab in USA. He has published more than 100 peer-reviewed papers in various journals and conference proceedings and has served on technical program committees and organizing committees of about 60 international conferences or workshops in the areas of database systems, web information systems, and workflow systems.
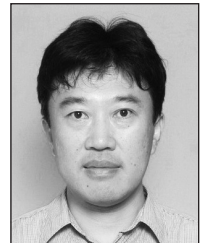
Jianxin Li received his BE, ME degrees in computer science from the Northeastern University, China, in 2002 and 2005, respectively. Currently, he is a PhD candidate at Swinburne University of Technology, Australia. His major research interests include XML query relaxation, optimization and keyword query processing.

Jeffrey Xu Yu received his BE, ME, and PhD in computer science, from the University of Tsukuba, Japan, in 1985, 1987 and 1990, respectively. Currently, Dr Yu is a professor in the Department of Systems Engineering and Engineering Management, the Chinese University of Hong Kong. His current main research interests include keywords search in relational databases, graph mining, graph query processing, graph pattern matching, uncertainty query processing, and data stream processing. Dr Yu has published over 190 papers including papers published in reputed journals and major international conferences, and served/serves in over 150 organization committees and program committees in international conferences/workshops.

Junhu Wang received his BS degree in mathematics from Hebei University, China in 1982, and his PhD in computer science from Griffith University, Australia in 2003. He is a senior lecturer at the School of Information and Communications Technology, Griffith University, Australia. His current research interests include query transformation and optimization in XML databases, keyword search in structured data, web services, and data quality management.

Jianxin Li

Jeffrey Xu Yu

Junhu Wang