2007-08-25

Semantic Query Optimization for Processing XML Streams with Minimized Memory Footprint

Ming Li
Worcester Polytechnic Institute

Semantic Query Optimization for Processing XML Streams with Minimized Memory Footprint

by

Ming Li

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

August 2007

APPROVED:

Professor Murali Mani, Advisor

Professor Elke A. Rundensteiner, Reader

Professor Michael Gennert, Head of Department

Abstract

XML streams have become increasingly prevalent in modern applications, ranging from network traffic monitoring to real-time information publishing. XQuery evaluation over XML streams requires the temporary buffering of XML elements, which not only utilizes system buffer and CPU resources but also causes un-necessary output latency. This thesis presents a semantic query optimization solution to minimize memory footprint during XQuery evaluation by exploiting XML schema knowledge. In many practical applications, XML streams are generated conforming to pre-defined schema constraints typically expressed via a DTD or an XML schema specification. Utilizing such constraints enables us to on-the-fly predict the non-occurrence of a given pattern within a bound context. This helps us to release buffered data earlier or possibly avoid ever storing it, thus achieving a minimized memory footprint. In this work, we focus on one particular class of constraints, namely, the Pattern Non-Occurrence (PNO) constraint. We develop an automaton-based technique to detect PNO constraints at runtime. For a given query, optimization opportunities which can be triggered by runtime PNO detection are explored for memory footprint minimization. Optimization decisions are encoded using our proposed Condition-Action Graph (CAG). The optimization-embedded execution strategy is then proposed to execute an optimized plan by detecting PNO constraints at run-time and then triggering the corresponding encoded actions when certain predefined conditions are satisfied. To ensure the efficiency of such PNO-triggered optimization, we propose a method for shrinking the CAGs by utilizing constraint knowledge during the query plan compiling phase. We implement our

optimization technique within the Raindrop XQuery engine. Our system implementation processes XQuery utilizing the Raindrop algebra. It is efficiently augmented by our optimization module, which uses Glushkov automaton technique to capture and monitor PNO constraints in parallel with the query-driven pattern retrieval. Finally, we conduct experimental studies using both real and synthetic data streams to illustrate that our techniques bring significant performance improvement in both memory and CPU usage as well as improved output latency over state-of-the-art solutions, with little overhead.

Acknowledgements

I would like to express my gratitude to my advisor Professor Murali Mani and my thesis reader Professor Elke A.Rundensteiner for everything they have done to make this thesis possible. I deeply appreciate their great help and great patience in guiding my study at Worcester Polytechnic Institute.

I am grateful for the help offered by other faculty and staff members from the Computer Science Department at WPI. Thanks a lot!

I thank all the lovely DSRGers, who are fun, smart, energetic, creative, easy-going, hardworking, knowledgable and all sharing the same belief that a week has seven days instead of only five.

My thanks also go to all the teachers who ever taught me during my kindergarten, primary school, middle school, high school, college and graduate school. OMG, I cannot believe I have attended that many that many that many different schools.

My parents receive my deepest gratitude. They have always believed in me and encouraged me. They don't know too much about computer science (the statement staying TRUE even when the word "science" is removed), but they have been teaching me how important "knowledge" is since I was 1-year-old.

My "*A Po*"(grandma at father's side) passed away two years ago. I miss her and sincerely dedicate this thesis to her.

Contents

List of Figures

Chapter 1

Introduction

XML and XQuery [W3C04] have been widely accepted as the standard data representation and query language for web applications (Figure 1.1) such as web services and on-line data delivery. Encoded XML streams are passed through network for data exchange between applications and/or users in a real-time infrastructure, which has the property of short response time and limited CPU/memory resources.

(a) News Publishing (b) On-line Auction

Figure 1.1: Application of Streaming XML

State-of-the-art XML stream engines for XQuery evaluation employ automaton for pattern retrieval and result construction. The in-time evaluation strategy is widely applied in the current XML stream engines for XQuery evaluation, 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 entirely different from node-selecting XPath [OMFB02], 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 large amount of XML tokens, the main memory buffer needed could be significant. Besides that, the CPU consumption on data buffering can also be significant. To provide real-time responses, as often required by applications to take prompt actions, serious challenges in CPU and memory utilization are faced by the XQuery evaluation over XML streams.

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

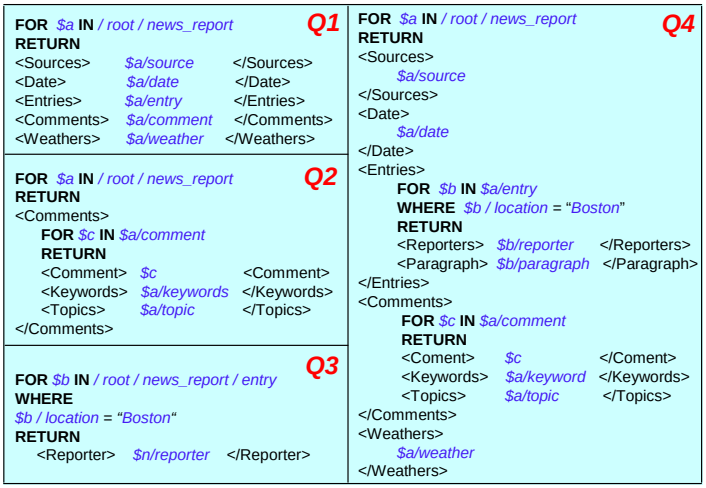

Figure 1.2: XQuery Examples

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. *Example 1* below illustrates such optimization opportunities.

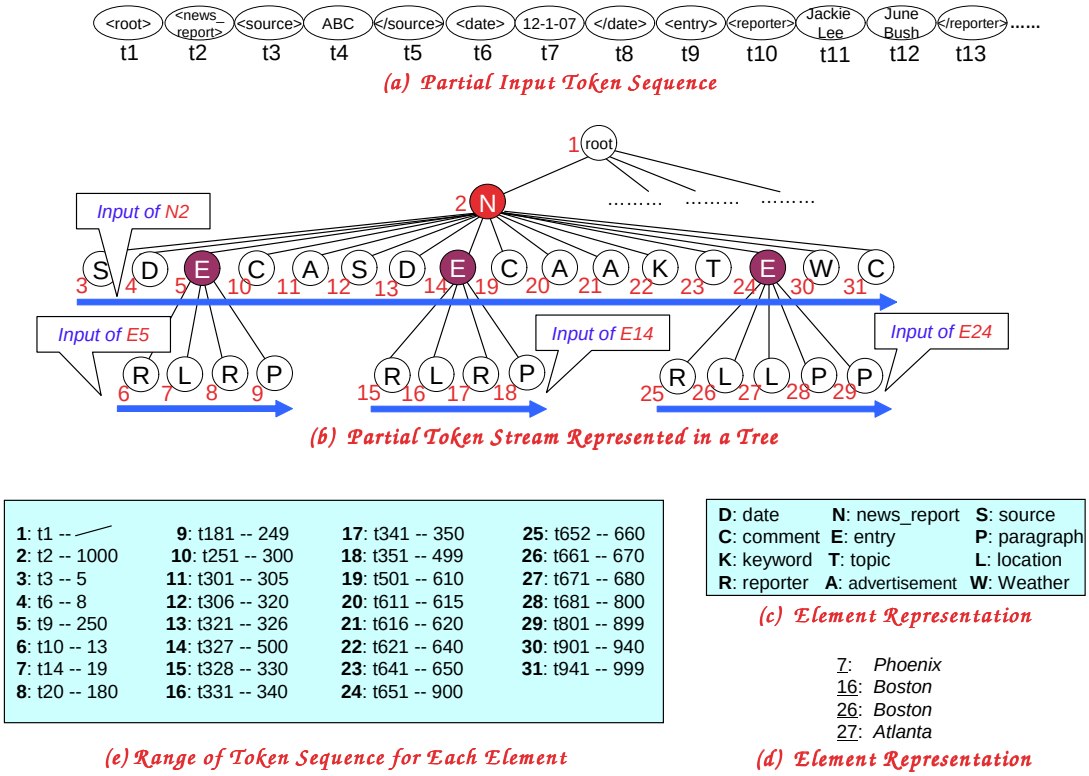

Figure 1.3: Input XML Token Stream

Example 1 (Motivating Example). Suppose that we are evaluating the three example XQueries *Q1* to *Q3* shown in Figure 1.2 over the example input XML token stream in Figure 1.3.

For each *news* element, (1) *Q1* lists the collection of its child *source*, *date*, *entry*, *comment* and *weather* elements; (2) *Q2* pairing each of its child *comment* elements with the collection of its child *keyword* and *topic* elements; (3) *Q3* returns the collection of the child *reporter* elements for each of its child *entry* elements which contain at least one *location* as "Boston". There are three types of token input are being

considered: the *start tag*, *PCDATA* and the *end tag*. Figure 1.3(a) shows the first 13 tokens from the input token sequence. Suppose 1000 tokens have been received. Figure 1.3(b) shows the equivalent XML tree representation. Each element node in the XML tree starts with one start tag token and ends with one end tag token. For simplification, the element nodes in the XML tree are shown by capitalized letters, with the corresponding description in Figure 1.3(c). Figure 1.3(e) shows such token sequence range for each element node. The two numbers associated with the node are the token IDs for the start and end tag token respectively. (An ID number is assigned to each input token by their input order for convenience of description)

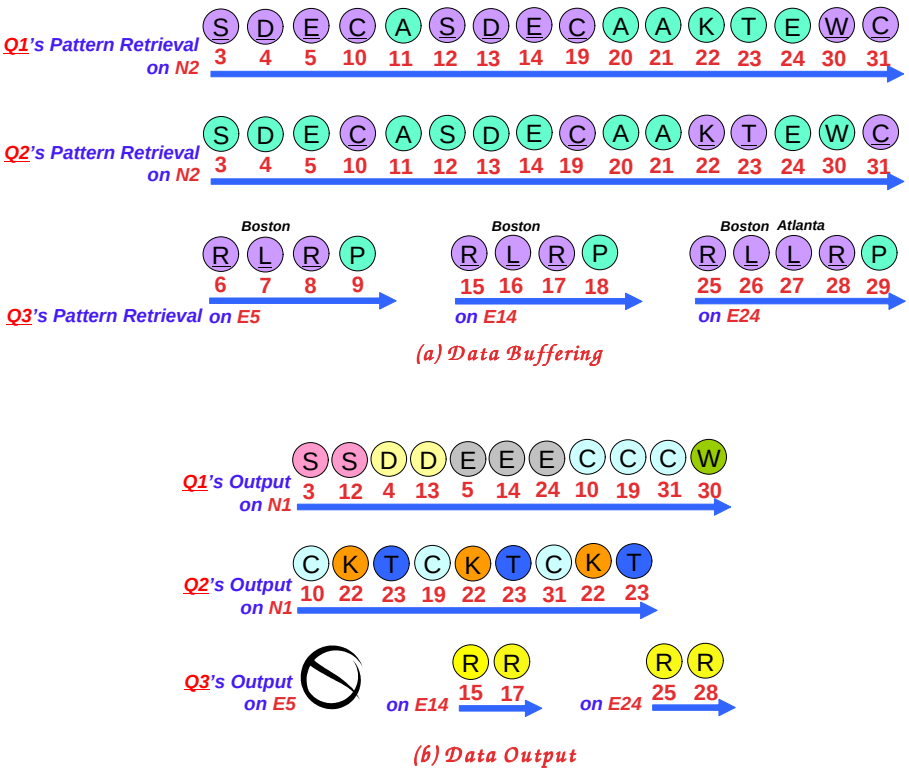

Figure 1.4: Data Buffering and Data Output in Evaluating Q1 to Q3 by Just-in-Time Strategy

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*, *comment* and *weather* are expected patterns under the binding on *news_report*. Child elements of expected patterns (the child elements marked by underscore in Figure 1.4) will be located during pattern retrieval on the input stream. The output for these three queries over the given example data is also shown in Figure 1.4. Similarly, *Q2* also collects all *news_report* bindings (*N2* in the example), *Q3* instead binds on each *entry* element (*E5*, *E14* and *E24* in our example).

We make the following three observations:

1. for *Q1* there is an order requirement on the output element types within each binding, i.e., the complete list of child *sources* needs to output before all the *dates* within a *news_report*;

2. for *Q2*, the query requires nesting between each *comment* element and the complete *keywords* plus *topics* list within a *news_report*, thus all the *keyword* and *topic* elements must be seen before outputting any *comment*);

3. for *Q3*, a predicate on pattern *location* needs to be satisfied before any output on an *entry* binding can be done.

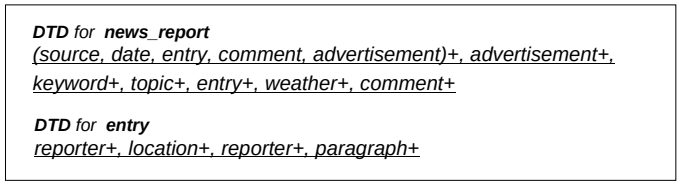

Figure 1.5: DTD Constraints

For *Q1*, among the expected patterns, only the *source* elements can avoid data buffering. The *date* elements must be kept until the completeness of *source* elements and *entry* elements are kept till the completeness of both *source* and *date* elements

and so on. Thus elements of these patterns need to be kept till the element being bound has been completely received from the stream (end tag token of *N2* (token no. 1000) is reached). If the DTD constraint in Figure 1.5 is given, when we reaching child element *A21*'s start tag (token no.616), 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 release the buffered *date* and *source* elements (*D4*, *D13*, *E5* and *E14*). Furthermore, token sequence of the *entry* element(s) coming in future (*E24*) can be directly output without being buffered. Thus, tokens no. 651 to no. 900 can be output without any buffer footprint. Similarly, while reaching element *W30*, from 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. Future *comment* elements(s) (*C31*) can be directly output. Finally, after the binding has been completely seen (receiving token no.1000), the *weather* element(s) (*W30*) in the memory can be output and released.

In *Q2*, elements of all the expected patterns require data buffering due to the nesting on output sequence. If the DTD constraint in Figure 1.5 is given for *news_report*, when the child element *E24* is met (by reaching token no. 651), we know that all the *keyword* and *topic* elements have been completely met under the current binding. Thus, the buffered *comment* elements (*C10* and *C19*) can be output pairing with the buffered *keyword* and *topic* elements (*K22* and *T23*). After that the two *comment* elements can be released from the buffer. Any *comment* element(s) arriving thereafter (*C31*) in our example can be directly output without buffering, by appending the buffered *K22* and *T23* also to the output. After the binding has been completely seen (by reaching token no. 1000), *K22* and *T23* can be released from the buffer.

In *Q3*, whether an *entry* element satisfies the predicate filtering is only known

when 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*. Suppose the DTD constraint in Figure 1.5 is given for the *entry* elements. For entry *E5*, when the child *reporter* element *R8* is met (by reaching token no.20), 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* can not pass the predicate verification. At this stage, all the buffered *location* and *reporter* elements can be simply discarded and released from the memory. Similarly for *E14*, the arrival of *R17* (at reaching token no. 341) guarantees no more *location* elements will come under this binding. Predicate verification gets satisfied for *E14*. Thus 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 any buffering. The same optimization process is as well applied to the *entry* element *E24* for evaluating *Q3*.

Figures 1.6, 1.7 and 1.8 show the data buffering of the above approach with semantic query optimization for evaluating *Q1* to *Q3*. The "</N>" / "</E>" indicate receiving the end tag of a binding *entry* element. The corresponding incremental data output is also shown on the bottom of each figure.

Obviously, the memory footprint is reduced by applying such semantic query optimization. If we can capture such runtime constraints that would help us to minimize the memory footprint with reasonable overhead cost. We note that as a side effect, CPU performance on query evaluation can also be improved (the execution time will be shortened and thus the output latency of the query will be minimized). Our goal in this work is try to use constraints to minimize the memory footprint in order to improve memory and CPU performance. We observe from the example that although the event constraints are known statically at the query

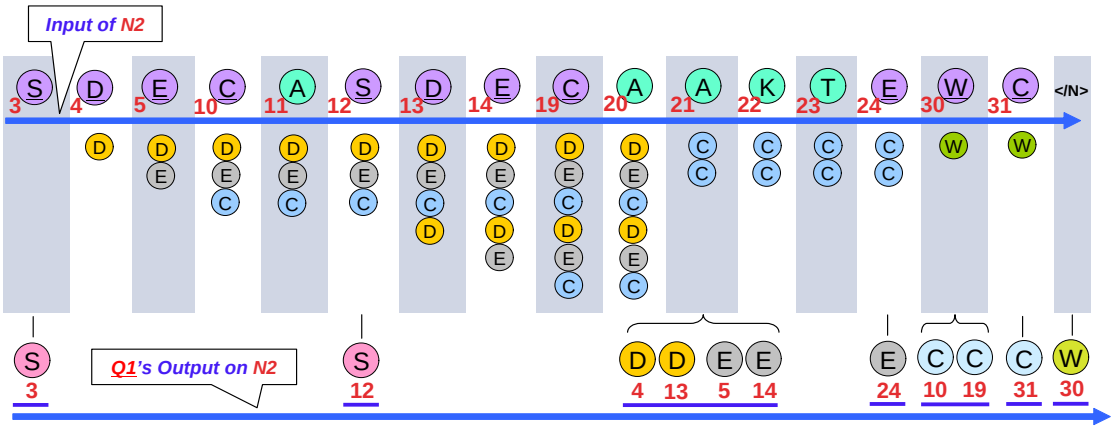

Figure 1.6: Memory Footprint in Evaluating Q1 with Semantic Query Optimization

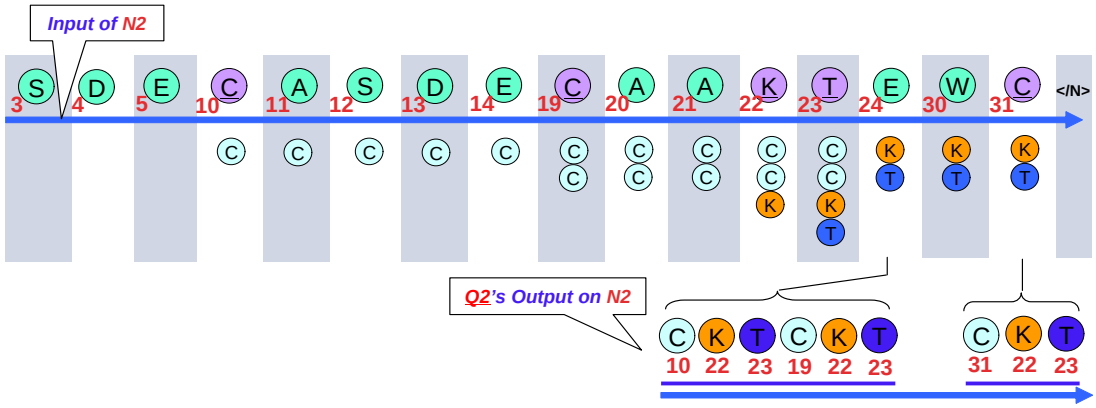

Figure 1.7: Memory Footprint in Evaluating Q2 with Semantic Query Optimization

compilation time, the real optimization opportunities only emerge at *runtime*, i.e., *A21* in *Q1*. Simply trying to detect the appearance of a certain pattern type [SRM05] cannot be a generic approach, as it doesn't fully use the semantic knowledge of the XML schema. Instead a runtime strategy is needed to detect such constraint knowledge about pattern completeness at runtime.

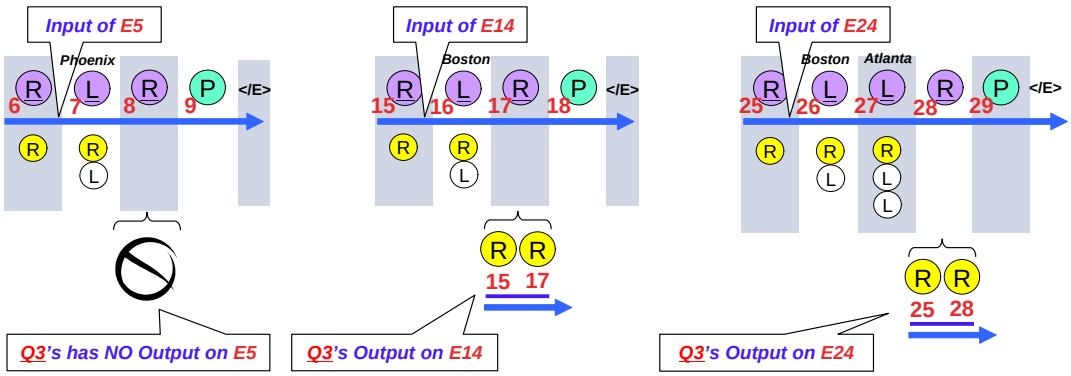

Figure 1.8: Memory Footprint in Evaluating Q3 with Semantic Query Optimization

Reducing the memory cost is very important for stream applications, as it can enable us to support more application functionalities as well as to yield better memory / CPU performance. Only a limited number of XML stream processing engines [BCCN06] [GC04] [SSK07] [DF03] [SRM05] and [KSSS04] have looked at the SQO opportunity focusing on the memory footprint minimization. Among them, SQO in [BCCN06] [GC04] is not stream specific while SQO in [SSK07] [DF03] [SRM05] and [KSSS04] are stream-specific but have drawbacks such as limited support for queries and limited optimization cases. The state-of-the-art will be further discussed in Chapter 8.

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

1. We reason the pattern non-occurrence constraint (*PNO Constraint*) and develop an automaton-based technique to effectively utilizing schema knowledge for runtime PNO constraint detection.

2. For a given query, we explore the optimization opportunities that could arise in a query expressed by our XQuery model for memory footprint minimization. Optimization decisions are encoded using our proposed Condition-Action Graph (CAG).

3. We propose an efficient plan execution strategy for realizing embedded runtime PNO constraint detection and runtime plan optimization.

4. To ensure the efficiency of such PNO-triggered optimization, we propose a mechanism to shrink the CAG by utilizing order constraints during the query plan compilation.

5. We implement our SQO technique within the Raindrop XQuery engine. Our system implementation processes XQuery expressions utilizing the Raindrop stream algebra. Our system is efficiently augmented by our optimization module, which uses Glushkov automatons to extract PNO constraints in parallel with pattern retrieval.

6. We conduct an experimental study using both real and synthetic data streams to illustrate that the proposed techniques bring significant performance improvements in both memory and CPU usage over state-of-the-art solutions.

Outline. We introduce our supported language and the basic just-in-time XQuery evaluation strategy in Chapter 2. In Chapter 3 we introduce the pattern non-occurrence constraint. We also propose mechanism to runtime detect such constraints based on given XML constraint knowledge. In Chapter 4 we propose the optimization model which utilizes pattern non-occurrence constraints to minimize the memory footprint. Chapter 5 introduces the mechanism to ensure the SQO efficiency by cutting the optimization overhead. In Chapter 6 we discuss the imple-

mentation design of our optimization model. Experimental results are analyzed in Chapter 7. Chapter 8 introduces the related work and the conclusion and future work are presented in Chapter 9.

Chapter 2

Preliminary

In this chapter, we first introduce the supported language We then propose a query tree representation to capture the pattern retrieval in an XQuery. Thereafter we will define an XQuery subset called *pattern query* which is the focus of our semantic query optimization. Finally we introduce the document type definition (*DTD*).

2.1 Supported Language

In our XQuery engine we support a subset of XQuery as shown in Figure 2.1. 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.

2.2 Query Tree

We propose query trees to represent the structural patterns in an XQuery. Figure 2.2 shows the query trees for *Q1* to *Q4* in Figure 1.2. XPaths in "FOR" clauses describe required patterns, e.g., in Figure 2.2(a), the "FOR" clause must not evaluate

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 2.1: Grammar of Supported XQuery Subset

to empty for the FWR expression to return any result. In contrast, XPaths in "RETURN"/"WHERE" clauses describe optional patterns, e.g., in Figure 2.2(a) even if *$a/source* evaluates to empty, a result element will still be constructed. XPaths in "WHERE" clauses describe predicate patterns for checking "existence" of an expected value, e.g., if *$b/location* contains any location element equal to "Boston", the bound *$b* on *entry* will be constructed. In the query tree, a solid (resp. dashed) line indicates the child is required (resp. optional) in its parent. A thick (resp. thin) line indicates the child is an predicate (resp. return) pattern. Query correlation is represented by a dash box. The returned patterns (nodes being connected by thin lines) are listed from left to right, following the required return order. In Figure 2.2(d) a thin dashed line connects *nodes 1*, *2*, *5*, *6* and *7* with their respective parent (*node 0*) indicating these are optional pattern types appearing in the "return" clause of each */root/news_report* binding. A thin solid line connecting the *nodes 3* and *4* with *node 0* indicates *$b/entry* and *$c/comment* appear in a "for" clause inside the outer binding on */root/news_report* to be a inner FWR binding. A thick dashed line connecting *nodes 3* and *8* shows the predicate on *$b/location*. Each binding on *$c*

joins each *entry* with the *keyword* and *topic* elements from the outer binding. Such correlation is represented using a dashed box.

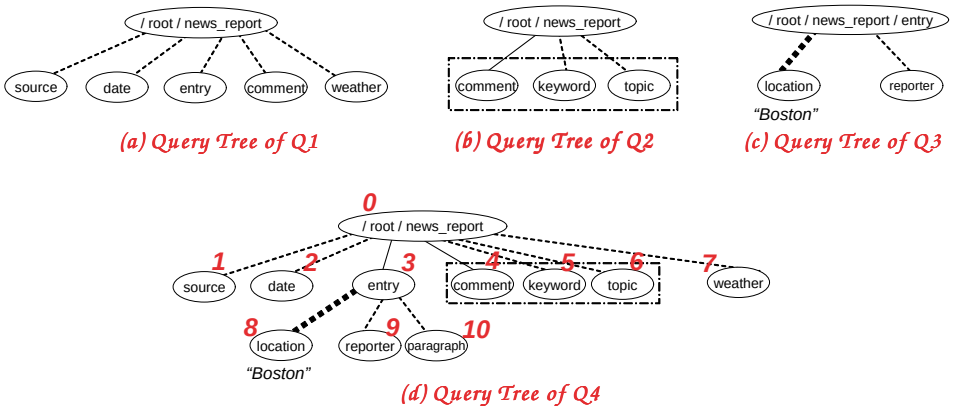

Figure 2.2: Query Tree of the Given XQuery Examples Q1 to Q4

2.3 Pattern Queries

Pattern Queries. A *pattern query* follows the *FWR template*

```
FOR $a IN xpath
WHERE where_1 AND where_2 ...  where_m  (m >= 1)
(called ‘‘WHERE list’’)
RETURN return_1  return_2  ...  return_n  (n >= 1)
(called ‘‘RETURN list’’)
```

and satisfies restriction (a) to (d) as listed below:

(a). *xpath* represents a path from the root;

(b). *where_i* above is an expression of the form $\$a/child_i =$ "$value_i$", where $child_i$ is a direct child pattern of the FOR binding ($\$a$);

(c). $return_i$ is either

1. $\$a/child_i$, where $child_i$ is a direct child pattern of the FOR binding $\$a$,

2. a pattern query with FOR binding IN $\$a/child_i$,

3. a one-level query correlation, as "FOR $\$a'$ IN $\$a/child_i$,
   RETURN $\$a'$ $\$a/inner\_return\_1$ $\$a/inner\_return\_2$ ... $\$a/inner\_return\_k$",
   where the $child_i$ are direct child patterns of the FOR binding $\$a$, the $inner\_return\_1$ to $inner\_return\_k$ are direct child patterns of the FOR binding $\$a$ **or** a pattern query with FOR binding IN $\$a/child_i$.

(d). no direct child pattern $child_i$ is both in the WHERE list and the RETURN list.

Single-Level Pattern Query. We further define the *single-level pattern query (SPQ)*, which is a pattern query with a loosened restriction in (c) as the $return_i$ is either one of the following:

1. $\$a/child_i$, where $child_i$ is a direct child pattern of the FOR binding $\$a$, (type 1)

2. a one-level query correlation, as "FOR $\$a'$ IN $\$a/child_i$,
   RETURN $\$a'$ $\$a/inner\_return\_1$ $\$a/inner\_return\_2$ ... $\$a/inner\_return\_k$",
   where the $inner\_return\_1$ to $inner\_return\_k$ and the $child_i$ are direct child patterns of the FOR binding on $\$a$. (type 2)

Below we define three simple types of SPQs which will be used in our further discussion:

Sequence SPQ if the query does not contain any WHERE clause and the RETURN list contains only $return_i(s)$ of type 1;

Nested-Sequence SPQ if the query does not contain any WHERE clause and the RETURN list contains one and only one $return_i$ of type 2;

Filter SPQ if there consists a WHERE clause in the query.

We can see that XQuery *Q1* to *Q3* in Figure 1.2 are all single-level pattern queries. *Q1* is a *sequence SPQ*, *Q2* is a *nested-sequence SPQ* and *Q3* is a *filter SPQ*.

The query tree of a SPQ has a depth of 2. We can see from Figure 2.2 that the query tree of *Q1* to *Q3* all have a depth of 2 (the root node and the child nodes). This is why it is referred to as "single-level".

Multi-Level Pattern Query (MPQ). A pattern queries which has a query tree with depth of at least 3 is called a *Multi-level Pattern Query (MPQ)*. We can see that *Q4* is an example of a MPQ. Its query tree is also given in Figure 2.2.

2.4 Document Type Definition

A *Document Type Definition* (*DTD*) [BPSM$^+$06] can be represented as a tuple $D =$ *(Eset; P; root)*, where *Eset* is a finite set of element types (equivalent to tag names); *root* is a distinguished type in *Eset*, called the *root type*; *P* defines the element types: for each element *E* in *Eset*, *P(E)* is a regular expression and $E \rightarrow P(E)$ is called the production of type *E*. To simplify the discussion, we consider *P(E)* of the form:

α ::= PCS | ϵ | B_1, B_2, ..., B_n | B_1 or B_2 or ... or B_n | B$*$ | B+

where PCS denotes the string (PCDATA) type, ϵ is the empty word, B is a type in *Eset* (referred to as a *subelement type* of E), and "*or*" and "," denote disjunction and concatenation respectively. "$*$" represents Kleene star and "+" represents one or more than one occurrence.

Let Σ be a set of symbols (equivalent to tag names). DTD is an extended context free grammar over Σ. Each production in a DTD is unambiguously identified by a tag name in Σ. As our data model, we consider the fragment of XML without attributes; it is trivial to incorporate attributes into the framework. The production of *news_report* and *entry* is shown in Figure 1.5.

Chapter 3

Pattern Non-Occurrence Constraints

By the previous examples, we can see that under a bound element, the non-occurrence of certain child patterns predicted at runtime can trigger the optimization leading to memory footprint minimization. In this chapter, 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 about applicable PNO constraints can be monitored at runtime. Thereafter, we introduce the monitoring algorithm for detecting PNO constraint evolution.

3.1 Definition

3.1.1 Element Types

As described in Sectionr̃efdtdintro, a type E is represented an atomic symbol, *P(E)* represents as the regular expression for type E ($E \rightarrow$ *P(E)*). *SymbSet(P(E))* is the set of atomic symbols that occur in *P(E)*. *L(P(E))* denotes the language defined by

P(E), which means the set of words over *SymbSet(P(E))* that can be recognized by *P(E)*. A word $ele \in L(P(E))$ consists of a sequence of symbols, where ele_i denotes the *i-th* symbol of *ele*.

Under the XML context, the type is equivalent to an element type, 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 child element types of type *E*. An element can be represented using a *type sequence*: it is a sequence of its child elements, where each element is represented by its type. Thus ele_i is the *i-th* child element in element *ele*. As example, let's look at Figure 3.1. The DTD of type *news_report* *P(news_report)* and a *news_report* element *N2* are given. Based on the given DTD, we can see that the possible types of a *news_report*'s child element are *source*, *date*, *entry*, *element*, *comment*, *advertisement*, *keyword*, *topic* aned *weather* (*SymbSet(P(news_report))*). $N2_{11}$ represents *ele*'s 11th child element, which is of type *advertisement*.

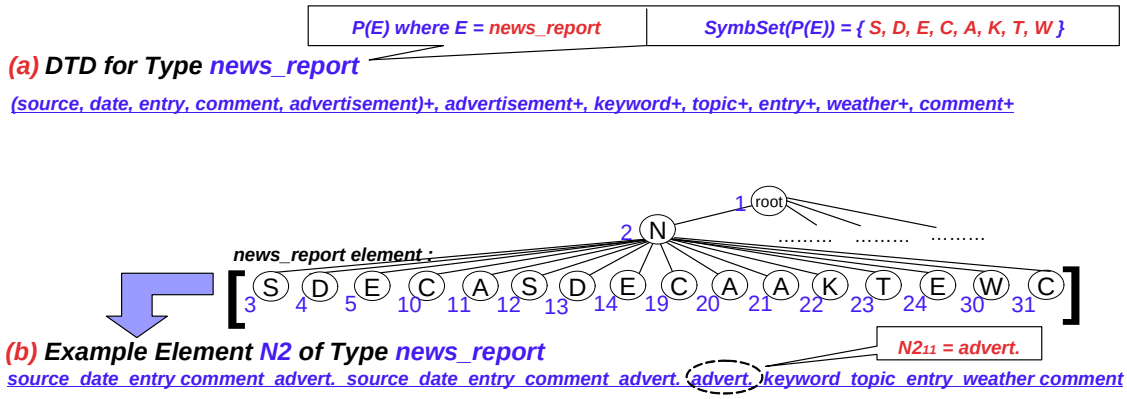

Figure 3.1: Element Types

3.1.2 Element Prefix and Element Evolution

Element Prefix. Through the stream input, an element is received on the fly child element by child element. 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 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 child elements. *Growth(p, E)* is the set of all possible evolved portion: given $p \in$ *Prefix(E)*, for any $pq \in$ *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 3.2. The type sequence $q =$ "*advertisement advertisement keyword topic*" is in p's *Growth* set. p evolves to the new element prefix p' by $\Rightarrow$*(p, q, E)*.

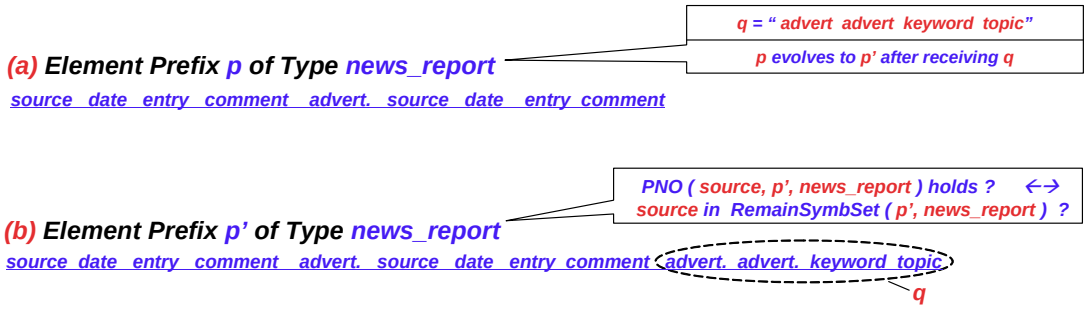

Figure 3.2: Element Prefix and PNO Constraint

3.1.3 Pattern Non-Occurrence (PNO) Constraint

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

Remaining Symbol Set. For p in *Prefix(E)*, its *remaining symbol set*, denoted as *RemainSymbSet(p)*, is the set of symbols which can appear in *Growth(p, E)*. Obviously, *PNO(symb, p, E) = FALSE (resp.TRUE)* implies *symb* is within (resp. not within) *RemainSymbSet(p, E))*. For example, given element prefix p in Figure 3.3, to determine whether *PNO(source, p, news_report)* holds is equivalent to determining whether *source* is not in the set *RemainSymbset(p, news_report)*.

Obviously, *PNO(symb, ϵ, E)* doesn't hold for any symbol in *SymbSet(P(E))*. (ϵ represents an empty element where no child element has been received yet.) Given an non-empty element prefix of type E, we want to determine the PNO constraint for a type in *SymbSet(P(E))*. However, the constraint cannot be simply determined by looking at the element prefix. For example, for prefix p or p' in Figure 3.2, *PNO(source, p(or p'), news_report)* cannot be simply determined by looking at p and its schema. A more sophisticated algorithm is needed, which will be discussed in the next section.

3.2 PNO Constraint Checking

In this section, we first model the semantic knowledge expressed by a DTD for a given element type using a deterministic finite automaton model. We then propose the PNO checking algorithm.

3.2.1 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))* and without redundant states . Given a regular expression, its equivalent DFA can be constructed in polynomial time [Koz03]. [Koz03] gives an algorithm to construct an equivalent DFA from a given regular expression. For a element prefix p of type E and a given DFA A in *AutoSet(P(E))*, *RS(p, A)* denotes the automaton state in A reached by running p on A.

As example, DFA A in Figure 3.3 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 *S8* on A. Suppose p' keeps evolving by taking in one new child element *entry*. The state transits from *S8* to *S9*.

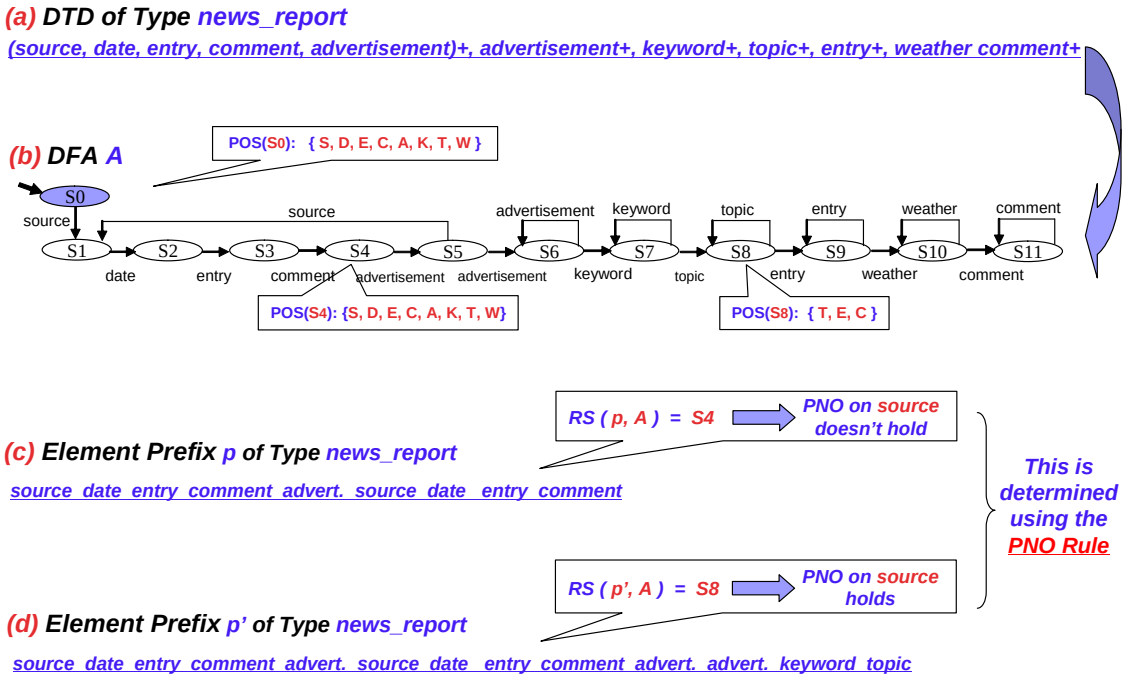

Figure 3.3: Regular Expression Represented by Deterministic Finite Automaton

3.2.2 PNO Rule

Possible Occurrence Set (POS) of a DFA State. For 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 [SAV95] can be applied for calculating POS for the states in a given DFA. The algorithm takes a DFA A as input and outputs the POS for each automaton state of A. We refer to such algorithm as *POS_Compute(A)*. POS for some example states $S0$, $S4$ and $S8$ of DFA A is shown in Figure 3.3. Take the start state *S0* as example. Obviously, *POS(S0)* equals to *SymbSet(news_report)*.

Theorem 1. (Equivalence between RemainSymbSet and POS) For element prefix p of type E and any $A \in AutoSet(P(E))$, *RemainSymbSet(p, E)* $=$ *POS(RS(p, A))*.

Proof. This theorem can be proven by contradiction. Because $A \in AutoSet(P(E))$, the language of A is equivalent to *L(P(E))*. Suppose *RemainSymbSet(p, E)* != *POS(RS(p, A))*. Thus there exist an element *ele* which is in *L(P(E))* but cannot be accepted by A or which is not in *L(P(E))* but can be accepted by A. Contradiction. Hence, *RemainSymbSet(p, E)* $=$ *POS(RS(p, A))*.

Based on *Theorem 1*, we propose the PNO rule which determines the satisfaction of a given PNO constraint based on the above DFA for *L(P(E))*:

PNO Rule. Given element prefix p of type E, any A in *AutoSet(P(E))* and symbol *symb*, *PNO(symb, p, E)* holds **iff** $symb \notin$ *POS(RS(p, A))*.

Whether *PNO(symb, p, E)* holds can be determined by a simple application of

the above PNO rule. Given prefix p and DFA A in *AutoSet(P(E))*, for symbol *symb*, the rule application on *PNO(symb, p, E)* is a simple POS check as Algorithm 1.

Algorithm 1 PNO Rule Application

Procedure:
PNO_Checking()
Input:
(1) DFA A in *AutoSet(P(E))*
(2) symbol *symb*
Output:
TRUE / FALSE indicating whether *PNO(symb, p, E)* holds

S = POS(RS(p, A))
if *symb* $\in S$ **then**
  return FALSE
else
  return TRUE
end if

As example we apply the PNO rule to the element prefix p in Figure 3.3 to determine whether *PNO(source, p, news_report)* holds. By running p on DFA A, state *S4* (*S4 = RS(p, A)*) is reached. Because *source* is contained in state *S4*'s *POS* (*source*, *date*, *entry*, *comment*, *advertisement*, *keyword*, *topic*, *weather*), by the PNO rule the constraint *PNO(source, p, news_report)* thus does't hold. By checking the constraint *PNO(source, p', news_report)*, we determine that the constraint holds since *source* is not in *POS(S8)*.

3.3 PNO Constraint Evolution

3.3.1 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 element prefix 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 $PNO(symb', p, E)$ doesn't, there is a *PNO constraint evolution* on $symb'$ at sg, denoted as $\xi(symb')$ at sg.

As the example shown in Figure 3.4, an element grows from prefix p to $p1$, then $p2$, $p3$ and at last to p' through a series of singleton element evolutions. We can see that a PNO constraint evolution on *source* occurred for $p2$ because while the symbol *source* is contained in $POS(S5)$, it is no longer being contained in $POS(S6)$.

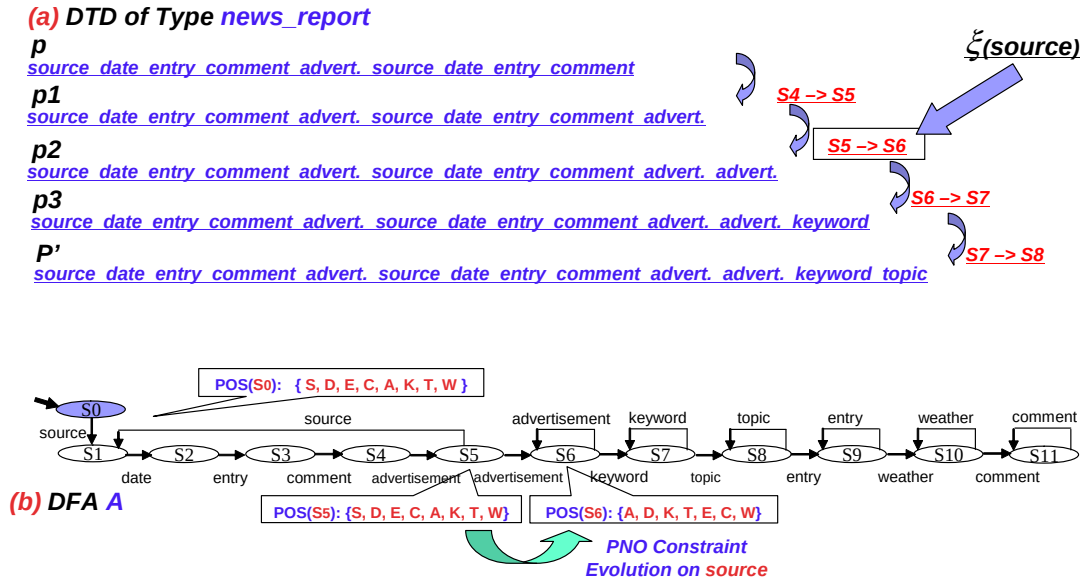

Figure 3.4: Evolvement of PNO Constraints

3.3.2 Monitoring PNO Constraint Evolutions

Theorem 2. (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.

Proof: The theorem can be proved by contradiction. Given element prefixes *p1*,

p2 of type E and $p1$ is $p2$'s prefix. Let A be a DFA in $AutoSet(P(E))$. Because $p1$ is $p2$'s prefix, *POS(RS(p2, A))* must be a subset of *POS(RS(p1, A))*. Suppose *PNO(symb, p1, E)* holds but *PNO(symb, p2, E)* doesn't for symbol *symb*, then there exists $symb'$ in *POS(RS(p2, A))* which is not in *POS(RS(p1, A))*. Contradiction. Hence the satisfaction *PNO(symb, p1, E)* implies the satisfaction of *PNO(symb, p2, E)*.

Based on *Theorem 2*, the following theorem is straightforward:

Theorem 3. 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.

Through the stream input, any non-empty prefix p of type E is on the fly constructed through $|p|$ steps of singleton element evolution sg_1, sg_2, sg_3, ..., sg_k(let k = $|p|$), where sg_1 is $\mapsto(\epsilon, p_1, E)$ and $sg_i(i > 1)$ is $\mapsto(p_1\ p_2\ ...,\ p_{i-1}, p_i, E)$. sg_i is the process of *receiving the i-th child of the current element*. Given element prefix p of type E and symbol *symb* in *SymbSet(L(E))*, if *PNO(symb, p, E)* holds, by *Theorem 2* we can conclude the following two facts:

(a). Through the singleton element evolution steps, there exists one and only one PNO evolution on *symb*.

(b). Assume the PNO evolution above is at sg_i. sg_i is the earliest moment to guarantee that child elements of type *symb* will not be seen in the remaining portion of the current element.

We then propose the following algorithm to monitor 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.

SEQ corresponds to the incremental growth of an element of type E. Algorithm 2 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 (the $End\_of\_Binding$ is received), PNO on $symb$ will be notified.

Algorithm 2 Monitoring Process of PNO Constraint Evolution

Procedure:
PNO_Monitoring()
Input:
(1) DFA A equivalent to $L(P(E))$ with S_0 as the start state;
(POS for each state in A has been pre-computed)
(2) symbol $symb$
(3) runtime input – a well-formed symbol sequence SEQ of type E received sequentially plus the termination message $End\_of\_Binding\ T$ received at the end
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 automaton transit function of A)
if S != S' (transiting to a new state in A) **then**
  $S = S'$
  if $symb \notin POS(S)$ **then**
    return the notification of $\xi(symb)$
  end if
end if
on receiving $End\_of\_Binding\ T$:
return the notification of $\xi(symb)$

As the example in Figure 3.4 shows, we can see that the PNO constraint on *source* holds at p' however the PNO evolution happens at $p2$. While the second *advertisement* arrives, the automaton transits 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 then stays TRUE for the remainder of pro-

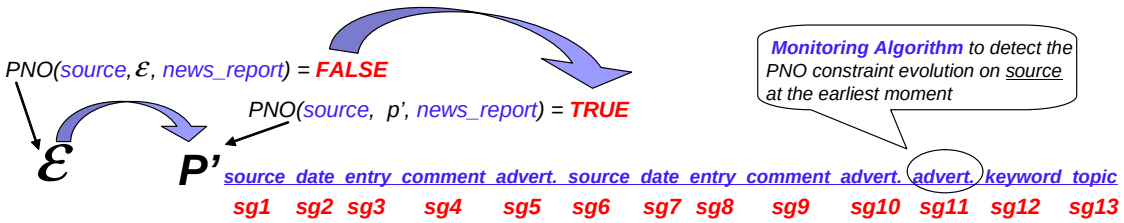

Figure 3.5: Monitoring of PNO Evolution

cessing the current element. Figure 3.5 shows an example of this PNO monitoring process on symbol *source* through the element growing from ϵ to p'.

Chapter 4

Memory-Oriented Optimization Utilizing PNO

4.1 Optimization of Single-Level Pattern Queries

4.1.1 General Guideline

The Basic Evaluation Strategy

As discussed earlier, the query semantics such as the *output sequence order* (*Q1*), *output sequence nesting* (*Q2*) and *predicate verification* (*Q3*) requires elements of expected patterns to be temporarily buffered within a binding until the processing of the bound parent element has been completed. The *just-in-time* execution strategy [SRM06] follows this paradigm of evaluation. Under this strategy, pattern retrieval is performed to locate and buffer all the child elements during the processing of each bound element. After a bound element has been completely received from the input stream, predicate checking and data output will be performed next and only after that the buffered child elements will be released from the memory. Thus, this strategy divides the query execution into two phases: (1) retrieving and

buffering expected child elements and (2) follow-up computation related to predicate verification, data output and buffering release. By this approach, within a binding all the elements of expected child patterns will need to be buffered until the binding element has been completely met.

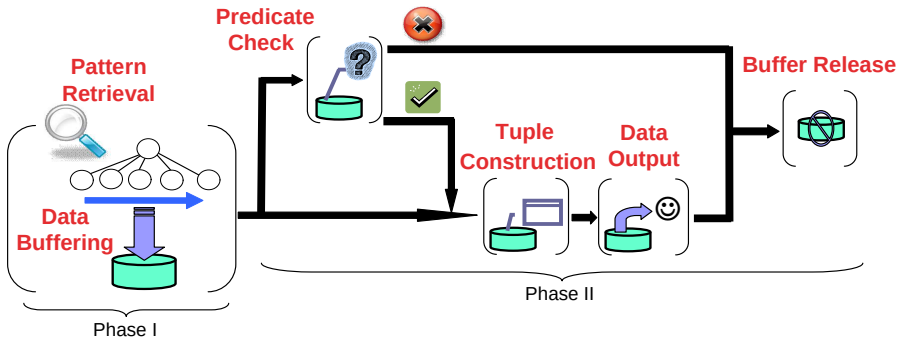

Figure 4.1: Strategy with the Basic Evaluation (Just-in-Time Strategy)

There are six types of computations in evaluating a single-level pattern query under such execution strategy, namely (1)*pattern retrieval*, (2)*data buffering*,(3)*predicate checking*, (4)*tuple construction*, (5)*data output* and (6)*buffer release*. For a query that does not contain any predicate filtering (*Q1* and *Q2*), step(3) will not be done. For a query that contains predicate filtering (*Q3*), if the predicate verification is not satisfied in step(3), i.e., for the binding element *E5*, buffer releasing (step(6)) will be directly taken. If the predicate filtering is satisfied, i.e., the binding element *E14* and *E24*, step(6) will be taken after a result tuple has been constructed and output (step(4) and step(5)). We can see that the computation under this strategy is separated into the two phases introduced above for evaluating a binding element: (1) and (2) correspond to the first phase which occurs until completely meeting the

binding element and then (3) to (6) corresponds to the second phase. Algorithm 3 sketches the just-in-time execution strategy and Figure 4.1 shows its execution flow.

Algorithm 3 Procedure of the Just-in-Time Execution Strategy

Procedure:
JustInTime_Execution_Strategy()
Input:
(1) token sequence within a binding, terminated by T
(2) single-level pattern query Q
Output:
query result of the binding

on receiving a new child element e from the input stream:
if type E (e's element type) is an expected child element type
then
  buffer the token sequence of e
else
  discard directly the token sequence of e upon receival
end if
on receiving binding termination T:
perform **predicate checking**, **tuple construction**,
data output and **buffer release**

We call the method of handling elements of an expected pattern the *handling mode* of this pattern. Under this just-in-time execution strategy, the retrieved expected patterns are buffered first. Such handling mode is referred to as *HOLD*. For instance, for *Q3*, the just-in-time execution strategy executes both the pattern *location* and *reporter* in the mode of HOLD.

Optimized Evaluation Strategy

By *Example 1*, intuitively we can see that a strategy with semantic query optimization of the memory footprint minimization needs to support the following mechanisms while evaluating a single-level pattern query:

1. **(Pattern Retrieval.)** Retrieve elements of expected patterns from the input stream;

2. **(Data Buffering.)** Buffer a retrieved child element within a binding;

3. **(Incremental Checking.)** Incrementally perform predicate verification on a buffered child element during the process of pattern retrieval. For example, we may need to check *L7* after reaching *R8* when evaluating the binding *E5* in evaluating *Q3*;

4. **(Incremental Output.)** Incrementally perform data output on a buffered child element during the process of pattern retrieval. For example, in *Q1*, we should output *D4*, *D13*, *E5* and *E14* after reaching *A21* when evaluating *N2*;

5. **(Incremental Release.)** Incrementally purge a buffered child element from the memory during the process of pattern retrieval, for example, in *Q1*, we should release the buffered child elements *D4*, *D13*, *E5* and *E14* as soon as output of these elements has been completed when evaluating *N2*;

6. **(Direct Output.)** Directly output the input token sequence of a retrieved child element. For example, *E24* in *Q1* and *C31* in *Q2* are output at the token granularity without any buffering;

7. **(Direct Releasing.)** Directly discard the input token sequence of a retrieved child element. For example, in *Q3 R8* is directly discarded token by token (by the token granularity) without any buffering during *E5*'s evaluation.

Algorithm 4 execution strategy with optimized evaluation

Procedure:
Optimized_Execution_Strategy()
Input:
(1) token sequence within a binding, terminated as T
(2) single-level pattern query Q
(3) PNO Monitor M (running procedure *PNO_Monitoring*)
Output:
query result of the binding

on receiving a new child element e on the input: **//EVENT**
pass E (e's pattern type) to M
while optimization opportunities arise **//CONDITION**
(based on M's feedback on PNO evolution)
safely perform actions (**a1**) **//ACTION**
if E is an expected child element type **then**
  perform corresponding action defined by E's handling mode (**a2**)
else
  discard the token sequence of e
end if
on receiving binding termination T:
pass T(the *End_of_Binding* message) to M

The following we introduce the proposed execution strategy. Algorithm 4 depicts the evaluation procedure under this strategy. There are two key differences between the Optimized_Execution_Strategy and the JustInTime_Execution_Strategy:

1. While a new child element is started, actions on the buffered data might be taken (a1). The tuple construction is no longer needed. Within a binding, all data output, buffer release and predicate verification are performed during the process of pattern retrieval (thus being called incremental check, output and release). By action (a1), the buffered elements can thus be released earlier than in the just-in-time strategy. For example, when *A21* is met, the buffered *D4*, *D13*, *E5* and *E14* can be output and the memory can then be released.

2. With the optimized evaluation, retrieved elements of an expected pattern do not always need to be buffered. Elements of an expected pattern can be instead directly output, such as *E24* in *Q1*, or directly released, such as *R8* in *Q3* ((a2) in the above procedure). By this, data buffering on some elements can be completely avoided. The handling mode for an expected pattern is no longer always "HOLD". It can instead be *DIRECT_OUTPUT* or *DIRECT_RELEASE*. The mode can be changed at runtime, such as for the *entry* pattern in *Q1*. At the beginning the entire pattern is required to be buffered, thus with the mode HOLD. After reaching *A21*, its mode is switched from HOLD to DIRECT_OUTPUT. The mode change is triggered by the action in (a1). For example, when *A21* is reached, the mode change on *entry* will be triggered from HOLD to DIRECT_OUTPUT.

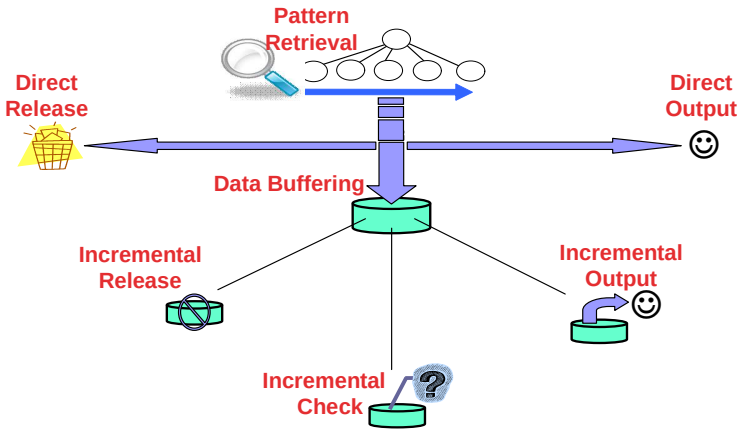

Figure 4.2: Strategy with Optimized Evaluation

Figure 4.2 shows the execution flow for the above strategy in handling single-level pattern queries. We can see that the computations, such as predicate verification

and data output, are happening in parallel with the pattern retrieval. Optimization is driven by the actions you take at $a1$, which conducts the output/release as well as the change of the handling mode of the corresponding expected patterns. Such mode change will further affect the execution step at $a2$ as well. The action taken at $a1$ is triggered by the runtime evolution of the PNO constraints, detected by runtime PNO Monitoring (procedure *PNO_Monitoring*).

Our proposed execution strategy follows the Event Condition Action (*ECA*) rule-based programming model. It consists of the following three parts:

1. **Receiving Events** (*EVENT*) which consumes the sequence of input child elements.

2. **Detecting Constraint** (*CONDITION*) which monitors PNO evolution on the expected patterns over the input (events) within a binding;

3. **Taking Action** (*ACTION*) which performs actions on the satisfaction of corresponding conditions.

Figures 4.3, 4.4 and 4.5 show the comparison in data buffering between the basic and optimized strategies for evaluating queries *Q1* to *Q3*.

4.1.2 Optimization for Sequence SPQ

Sequence SPQ Q_{seq} shown in Figure 4.6 returns the list of child E_0 elements to child E_n elements. Such required order among output types is referred to as *output sequence order*. Each return pattern has a certain position in the pattern sequence without repetition. For example, in Q_{seq}, the list of elements of type E_i needs to be output earlier than the list of elements of type E_k, if $i < k$. Straightforwardly, for elements of type E_1, they can be output directly without any buffering. For $k > 0$, before any output of the elements of type E_k, all the E_1, E_2, ..., E_{k-1} elements

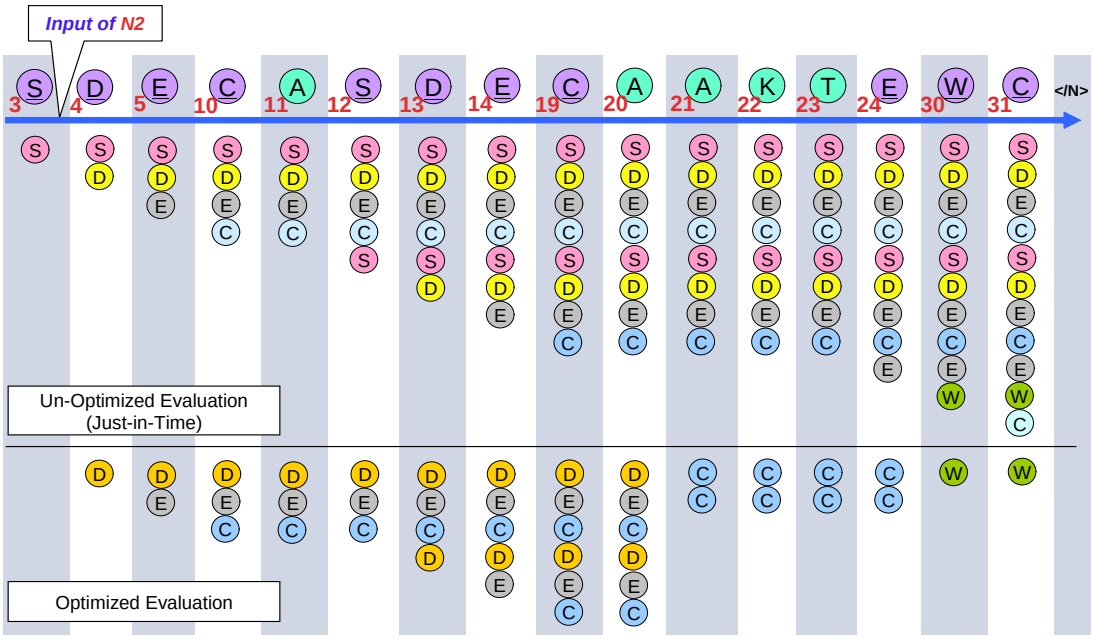

Figure 4.3: Comparison on Memory Footprint between the Optimized and the Basic Evaluation for Q1

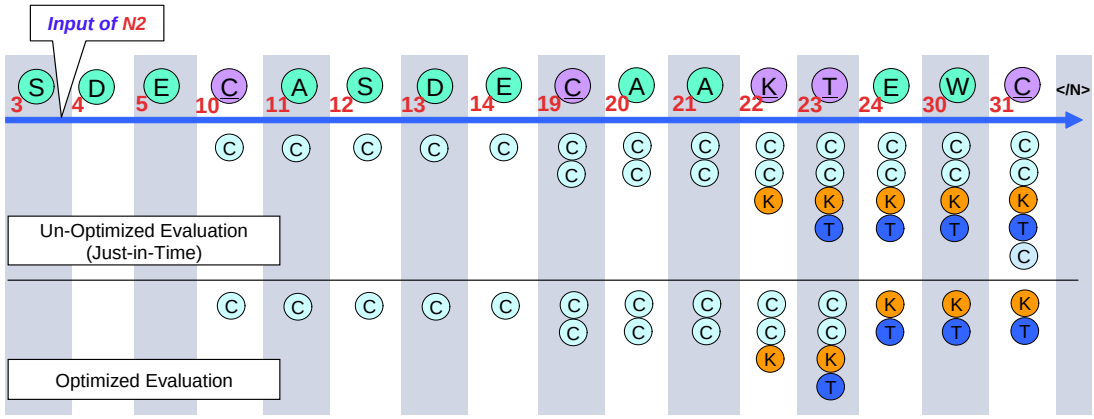

Figure 4.4: Comparison on Memory Footprint between the Optimized and the Basic Evaluation for Q2

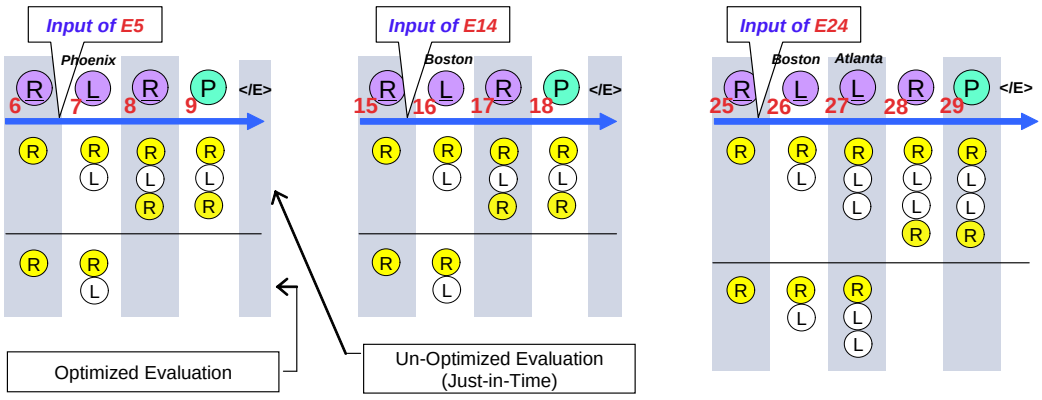

Figure 4.5: Comparison on Memory Footprint between the Optimized and the Basic Evaluation for Q3

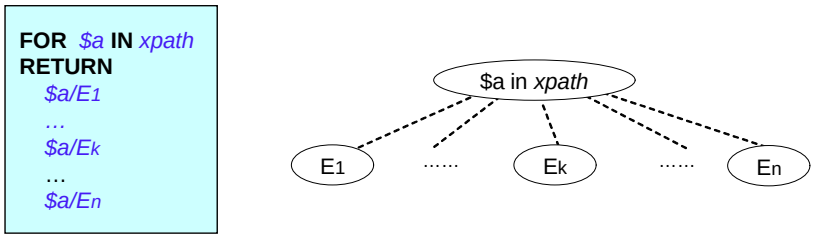

Figure 4.6: Sequence SPQ Q_{seq}

must already have been output. Hence the elements of type E_1 to E_{k-1} need to be completely met (which is equivalent to 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 E_k have to be buffered. Once such PNO condition is satisfied, if the execution strategy guarantees that all elements of types E_1 to E_{k-1} have been output, we can perform the following action on E_k:

1. Output the buffered E_k's;

2. Release the buffered E_k's;

3. Change the *handling mode* for E_k from HOLD to DIRECT_OUTPUT (to signal that future E_k elements are directly output without any buffering).

Based on the above description, we can define the following n conditions and their corresponding actions. Optimization starts from the just-in-time approach, where every pattern initially assumes the HOLD handling mode. Thus, the action on E_1 doesn't require any condition.

Condition 1 (C1): $\emptyset$

Action 1 (A1): change the *handling mode* for E_1 from HOLD to DIRECT_OUTPUT.

Condition 2 (C2): PNO holds on E_1.

Action 2 (A2): output and then release the buffered E_2 elements, change the *handling mode* for E_2 from HOLD to DIRECT_OUTPUT.

Condition 3 (C3): PNO holds on E_1 and E_2.

Action 3 (A3): output and then release the buffered E_3, elements, change the *handling mode* for E_3 from HOLD to DIRECT_OUTPUT.

.....

Condition n (Cn): PNO holds on E_1, E_2, ..., E_{n-1}.

Action n (An): output and then release the buffered E_n elements, change the *handling mode* for E_n from HOLD to DIRECT_OUTPUT.

As previously discussed, any element under type E_1 can simply be directly output without buffering. Thus it doesn't require any memory footprint in the evaluation, which obviously is optimal for handling child elements of this pattern in terms of memory usage. For elements of type E_k ($0 < k <= \text{n}$), if Action A_k above is taken when Condition C_k is satisfied, the handling on E_k is optimal memory-wise because it only keeps the E_k elements that must be kept plus the buffered E_k are released at the earliest possible moment.

From Chapter 3 we know that if we apply the PNO monitoring algorithm on types E_1 to E_{k-1}, we can detect the earliest possible moment when C_k gets satisfied. The question is whether taking the action A_k can guarantee the result correctness. Obviously, for taking the action on type E_k, all elements of types E_1 to E_{k-1} under the binding need to be totally met and output.

Claim. Action A_k ($k > 0$) can be safely taken if condition C_k is satisfied and action on type E_{k-1} has already been taken.

Proof Sketch: The claim can be proved by showing that if the actions on E_{k-1} are taken, after the PNO condition on type E_k has been satisfied, the whole list of child elements E_1 to E_{k-1} elements will have been completely output.

As an example, let's look at the evaluation of *Q1*. When *A21* is reached (token no. 616, <advertisement>), the PNO monitor indicates that the PNO constraint evolution happens on both type *source* and *date*. Thus, C_2 and C_3 get satisfied at the same time. A_1 needs to be taken before A_2.

4.1.3 Optimization for Nested-Sequence SPQ

The nested-Sequence SPQ $Q_{nested-seq}$ shown in Figure 4.7 returns child pattern elements with nesting. Straightforwardly, any output on the child ER elements requires that all the child E_1, E_2, ..., E_n have been completely met. Thus, before

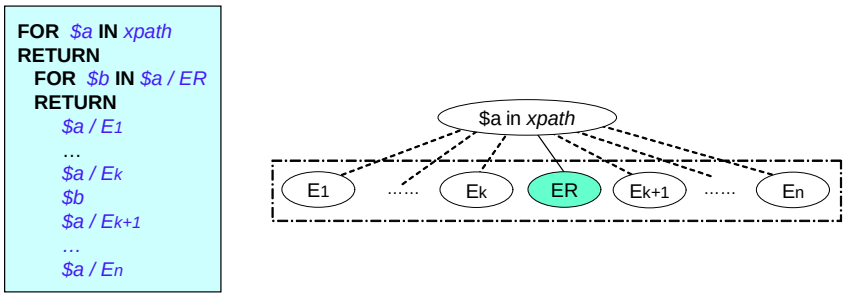

Figure 4.7: Nested-Sequence SPQ $Q_{nested-seq}$

the current element satisfies the PNO constraint on E_1 to E_n, elements of type ER have to be buffered. With the satisfaction of the PNO constraints on E_1, E_2, ..., E_n (**Condition 1**), we can perform the following action (**Action 1**):

1. Supposing the buffered ER elements are er_1, er_2, ..., er_m, in their arrival order, for each er_i, in the order from 1 to m, output the element list of E_1, the element list of E_2, ..., the element list of E_k, er_i, the element list of E_{k+1}, ..., followed by the element list of E_n.

2. Release all the buffered elements of ER.

3. Change the *handling mode* for ER from HOLD to DIRECT_OUTPUT.

The above step of mode switching is slightly different from the one in the evaluation of the previously discussed sequence SPQs. Additional action for appending the buffered collection of E_1 to E_n are needed besides directly outputting each newly arriving ER element. For example, for *Q2*, when *C31* arrives, it will be output directly however after that the list of buffered *keyword* and *topic* elements needs to be appended. If the query is changed to output "$a/keyword, $a/topic, $c", such

appended output will be performed before the direct output on *C31*: while the start tag token of *C31* is met, we output the buffered *keyword* and *topic* list, then output the start tag token as well as the following input tokens of *C31*.

After the PNO of E_1 to E_n as well as the PNO of ER all have been satisfied (**Condition 2**), the buffered elements of type E_1 to E_n can be released (**Action 2**) after producing the corresponding result. Similarly to the handling of sequence SPQs, handling of ER as well as E_1 to E_n are optimal if we apply the PNO monitoring algorithm to detect the earliest possible moment for when the expected condition becomes first satisfied. Obviously, *Action 1* needs to be taken before *Action 2*, in the case that both conditions get satisfied at the same event.

4.1.4 Optimization for Filter SPQ

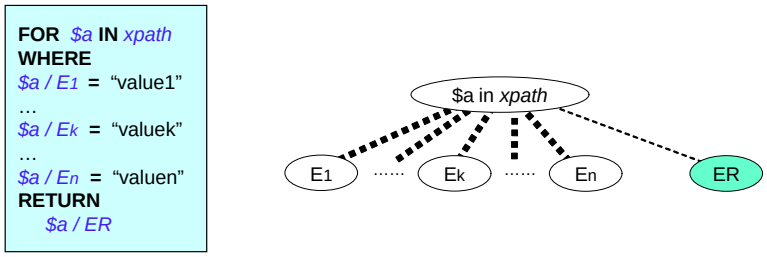

Figure 4.8: Filter SPQ Q_{filter}

The filter SPQ Q_{filter} shown in Figure 4.8 returns child elements of ER for the binding $\$a$ which matches the conjunctive existence predicate verification on E_1 to E_n. Obviously, once the predicate is determined to be unsatisfied, the buffered child elements under this binding thus far can now be purged directly. Also no more buffering would be needed for any pattern under the current binding. Once

the predicate checking is determined to be satisfied, the remaining conditions and actions will be the same as in the case of a simple non-filter SPQ.

For a conjunctive query, if the predicate is satisfied, a required match on each single predicate pattern can already determine the success of the binding. However, for avoiding such context switching, we assume that predicate checking on a predicate branch E_k ($1 <= k <= n$) can only be undertaken while the PNO on E_k has been satisfied. Based on this assumption, the above approach can be shown to be optimal memory-wise.

Thus as in *Q3*, for *E5* (which fails the predicate checking), its child *reporter* elements which are potentially needed to be consumed for constructing the result output sequence are no longer useful due to the predicate being finalized as FALSE when reaching *R8*. Thus, the buffered *reporter* / *location* element(s) *R6* and *L7* can be purged instead of waiting for the end of *news*. Also, the *reporter* element *R8* that occur afterwards can be omitted without any storage.

4.2 Optimization on Multi-Level Pattern Queries

Figure 4.9 shows the query tree for the multi-level pattern query *Q4*, while Figure 4.10 shows two different scenarios of optimization:

1. The inner binding is with the handling mode set to DIRECT_OUPUT or DIRECT_DISCARD: this inner binding can be treated in the same fashion as the top most binding for buffer optimization (E_{24} in Figure 4.11);

2. The inner binding is with the handling mode set to HOLD: if the binding does not contain any predicate checking or with a satisfied predicate checking, no buffer optimization can be applied on this inner binding in terms of memory consumption (E_{14} in Figure 4.11); else, when the predicate checking is deter-

mined to be failing, the buffer for the inner binding can be released and no further buffering on this binding is further needed (E_5 in Figure 4.11).

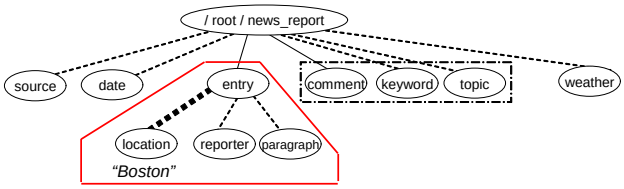

Figure 4.9: The Inner Subtree of Q4's Query Tree

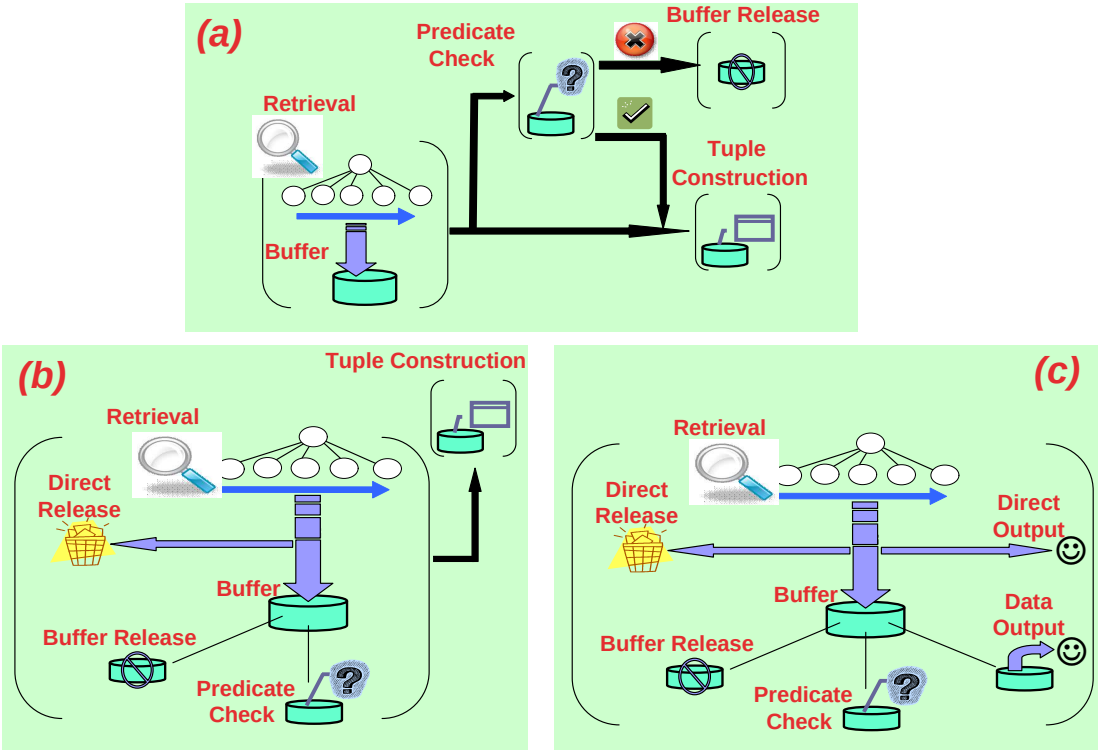

Figure 4.10: Comparison between Two Strategy in MPQ Evaluation

4.3 Condition Action Graph (CAG)

We propose our algorithm based on a data structure called *CAG* (Condition-Action Graph) to efficiently check the conditions and to ensure that an action is taken when

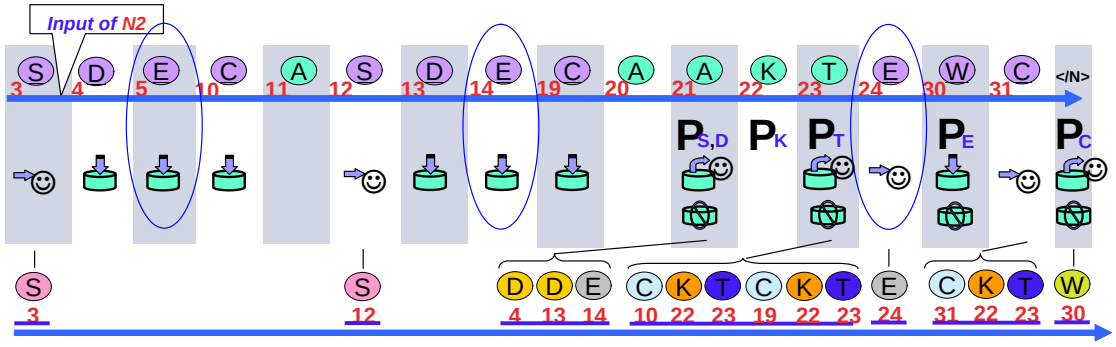

Figure 4.11: Query Evaluation of Q4

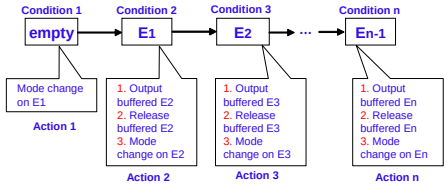

Figure 4.12: CAG of Sequence SPQ

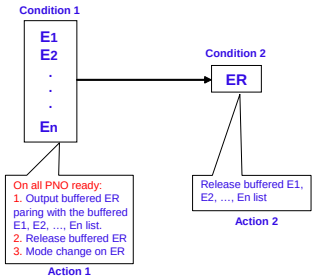

Figure 4.13: CAG of Nested-Sequence SPQ

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 PNO constraints get satisfied.

The CAG of Q_{seq} (Figure 4.6) is shown in Figure 4.12 and the CAG of $Q_{nested-seq}$ (Figure 4.7) is shown in Figure 4.13. The construction algorithms are straightforward, which is shown in Algorithm 5 and 6.

Algorithm 5 CAG Construction for Sequence SPQ

Procedure:
CAG_Q_{seq}()
Input:
sequence SPQ Q_{seq} (Figure 4.6)
Output:
CAG CAG_{seq}

i = 1
while i != n
construct condition state S_i:
if $i > 1$ **then**
  put in PNO requirement PNO_k (1 <= k < i)
else
  set the condition as empty
end if
encode action for state S_i:
(1)output buffered E_k's
(2)release buffered E_k's
(3)change the *handling mode* for E_i from HOLD to DIRECT_OUTPUT
(1 <= k < i)
if i != 1 **then**
  connect S_{i-1} to S_i
end if

For filter SPQ Q_{filter} (Figure 4.8), each predicate branch will be mapped to a condition. Its corresponding action is to release all the buffer and to avoid all the future buffering on the binding when PNO is received and the associated predicate is not satisfied. When all PNOs have been received and no branch fails, the CAG then moves to the next state. Figure 4.14 shows the construction of the filtering CAG state.

We already have examined three different categories of SPQs. Each of them

Algorithm 6 CAG Construction for Nested-Sequence SPQ

Procedure:
CAG_$Q_{nested-seq}$()
Input:
nested-sequence SPQ $Q_{nested-seq}$ (Figure 4.7)
Output:
CAG $CAG_{nested-seq}$

construct condition state S_1:
put in PNO requirement PNO_k (1 <= k <= n)
encode action for state S_1:
(1)produce and output join result on the buffer ER's and E_1's to E_k's
(2)release buffered ER's
(3)change the *handling mode* for E_k from HOLD to DIRECT_OUTPUT
construct condition state S_2:
put in PNO requirement PNO_{ER}
encode action for state S_2:
release buffered E_k's (1 <= k <= n)
connect S_1 to S_2

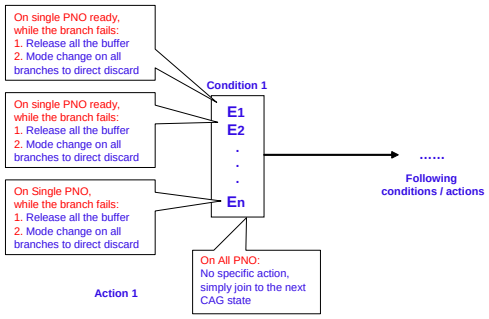

Figure 4.14: CAG of Filter SPQ

represents one reason for why data needs to be hold during the SPQ evaluation, namely, *holding on ordered output sequence*, *holding on nested output sequence* and *holding on predicate verification*. Some child elements might need to be buffered within the bound parent element for more than one reason. In such case, given the root node of a pattern query tree, we can conduct a single left to right scan of its child branches for the CAG construction. Figure 4.15 shows an example of a CAG construction for such a combined data holding situation.

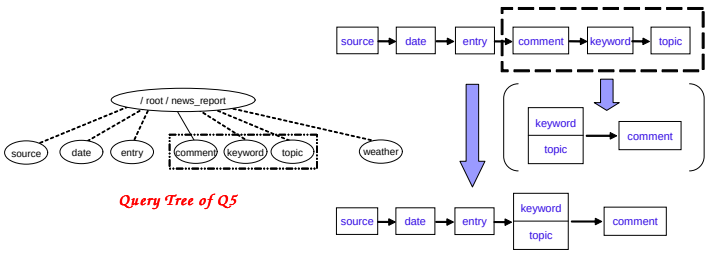

Figure 4.15: Combined CAG Construction

Based on the CAG approach, the execution monitors the input token sequence using our monitor algorithm (Algorithm 2) for detecting the PNO evolution of the element types contained in the CAG state. When a condition is satisfied, its corresponding action will be taken and the CAG state jumps to the next. The condition

on the subsequent state will be checked and its action will be taken if it is satisfied. By such CAG-based process, the execution will monitor the patterns associated with the current CAG state only, instead of monitoring every expected child pattern. Algorithm 7 describes this procedure.

Figure 4.16 shows the CAG construction for *Q4*. Each level of the query tree maps to one CAG component. CAGs on different level are connected through corresponding inner binding.

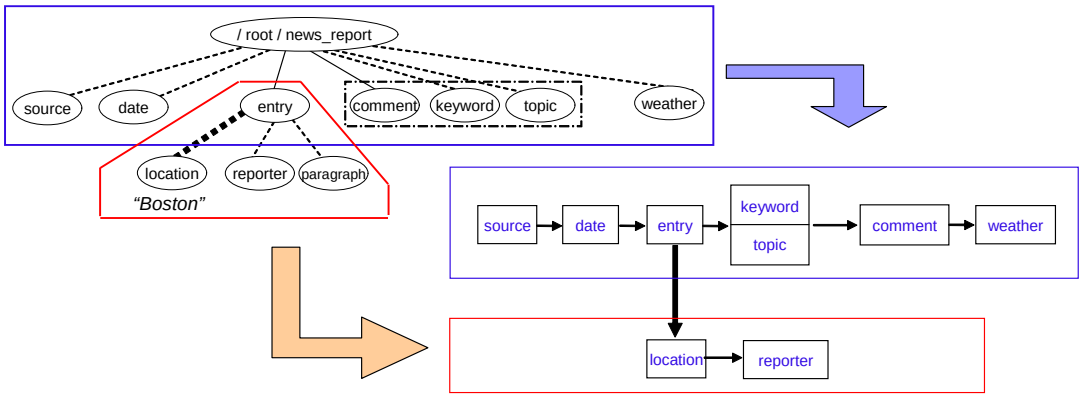

Figure 4.16: CAG construction for Q4

Algorithm 7 Monitoring Process of PNO Constraint Evolution during CAG-Based Execution

Procedure:
CAG_Driven_PNO_Monitoring()
Input:
(1) a well-formed symbol sequence *SEQ* of type E received sequentially
at runtime plus the termination message $End\_of\_Binding\ T$ received at the end
(2) DFA A equivalent to *L(P(E))* with S_0 as the start state;
(POS set for each state in A has been pre-computed)
(3) symbol set *MonitorSymbSet* received at runtime
Output:
notification of $\xi(symb)$, where symb in *MonitorSymbSet*

state $S = S_0$
on receiving symbol input $symb_{input}$:
S' = *tf(S,* $symb_{input}$*)* (tf as the automaton transit function of A)
if S != S' (transiting to a new state in A) **then**
  $S = S'$
  for each symb in *MonitorSymbSet*
  if *symb* $\notin$ *POS(S)* **then**
    throw notification of ξ*(symb)*
    remove symb from *MonitorSymbSet*
  end if
end if
on receiving updates on *MonitorSymbSet*:
update *MonitorSymbSet*
on receiving $END\_OF\_BINDING\ T$:
notification of ξ*(symb)*

Chapter 5

Towards an Efficient SQO

5.1 Considering Constraint Knowledge at CAG Construction

In our previous algorithm in constructing the CAG, we only consider the query while the constraint knowledge is omitted. Considering the constraint knowledge at compilation time when the CAG is constructed will lead to a more efficient SQO by re-structuring the CAG.

5.1.1 Cutting the CAG by Cutting Unreachable States

Unreachable states should be removed from the CAG to avoid the corresponding monitoring process so they would guarantee to be non-beneficial. For instance, in the example shown in Figure 5.1, GA states starting from state 3 are removed from the CAG because from the constraint knowledge we know that the constraint in *Condition 3* (PNO of pattern 3) cannot be satisfied until we reach the end of the binding.

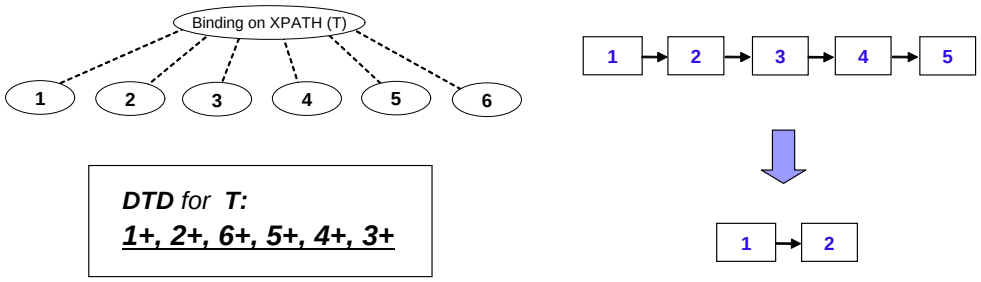

Figure 5.1: Cutting the CAG by Cutting Unreachable States

5.1.2 Shrinking the CAG by Applying Global Ordering Knowledge

In some cases it is not necessary to notify about pattern completeness at the earliest possible moment even though at this moment the PNO has been satisfied.

For any two sibling types in the DTD, an order constraint can be inferred. Order constraints between two patterns, defined as $Ord(E_m, E_n; E)$, indicates that under an element of type E, no elements of type E_n appear before encountering all the elements of type E_m.

Order constraints can be used to indicate the completeness of a certain pattern. For instance, in the example shown in Figure 5.2, GA states can be merged to avoid any unnecessary context switches during PNO monitoring and in some cases even avoid PNO monitoring. For instance, in the example shown in Figure 5.3, the PNO monitoring can be completely avoided.

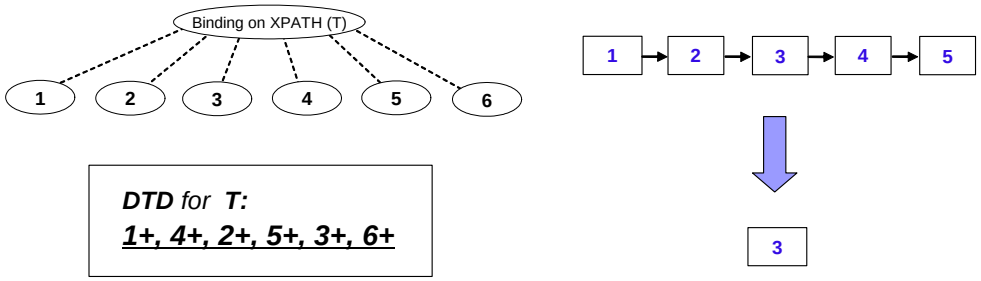

Figure 5.2: Shrinking the CAG by Applying Global Order

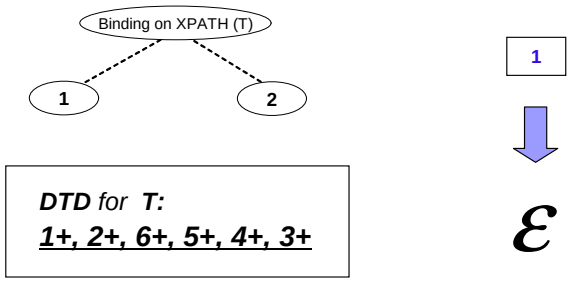

Figure 5.3: Shrinking the CAG by Applying Global Order (Cont.)

5.2 Applying Ending Marks

One alternative approach to using an automaton to detect PNO constraint at runtime is to introduce new patterns into the pattern retrieval itself [SRM05]. For instance, the appearance of any *keyword* can serve as *source*'s ending mark in *Example 1*. Obviously, such ending marks might reduce the monitoring overhead but it might not be optimal memory-wise.

Chapter 6

Implementation

We have incorporated the proposed optimization strategy into the Raindrop system [SRM06], an XQuery stream processing engine. Below we first describe the Raindrop engine and then review our SQO extension to the engine.

6.1 Raindrop XQuery Engine

6.1.1 Raindrop Algebra

Raindrop presents an XQuery expression as an algebraic plan. The algebra consists of XML specific operators such as *Navigate*, *ExtractColl*, *ExtractUnnest*, *StructuralJoin* and *Tagger*. It also consists SQL like operators such as *Select*, *Projection*, *Join* and *Aggregate*. The input and output of the operators are a collection of tuples. A cell in a tuple can contain a token, a single XML node or a collection of XML nodes.

Raindrop evaluates an XQuery by the just-in-time execution strategy [SRM06]. As the discussion in our previous examples, XQuery evaluation generally consists of two phases, namely *pattern retrieval* and *result construction*. In the *pattern retrieval* phase, elements of the expected patterns are located and the token sequence

of return elements and predicate element are extracted. *TokenNav* operator is used for locating an pattern. For simplification, here we will combine the different *TokenNav* operators and abstract them as one single operator *Navigate*, which is used for locating elements of the expected patterns from the input token stream. The *Extract* operator then passes the token sequence of each located element to the result construction operator (*Tagger*). *Select* operator is used for predicate checking on the input of a predicate branch. Based on the checking result of its child *Select* operator, *SJoin* produces an output unit for each bound element by consuming the data passed up from its child operators based on the required re-construction requirements such as sequence ordering and sequence nesting. Figure 6.1 shows the Raindrop algebra. For further description, please refer to [SRM06].

Op	Symbol	Semantic
Selection	σ_{pred}	Filter tuples based on the predicate *pred*
Projection	Π_{v}	Filter columns in the input tuples based on the variable list *v*
Join	$\bowtie$	Join input tuples based on the predicate *pred*
Aggregate	Δ_{f}	Aggregate over input tuples with the aggregate function *f*, e.g., sum and average
Tagger	T_{pt}	Format outputs based on the pattern *pt*, i.e., reconstruct XML tags
Navigate	$\Phi_{p1,\ p2}$	Take input elements of path *p1* and output ancestor elements of path *p2*
ExtractColl	Ψ_{p}	Identify all the elements of path *p* from the input stream within a binding
ExtractUnnest	Ψ_{up}	Identify an element of path *p* from the input stream within a binding
Structural Join	$\bowtie_{SJ}$	Join input tuples on their structural relationship, e.g, the common parent relationship *p*

Figure 6.1: Raindrop Query Algebra

6.1.2 Automaton-Based Pattern Retrieval Implementation

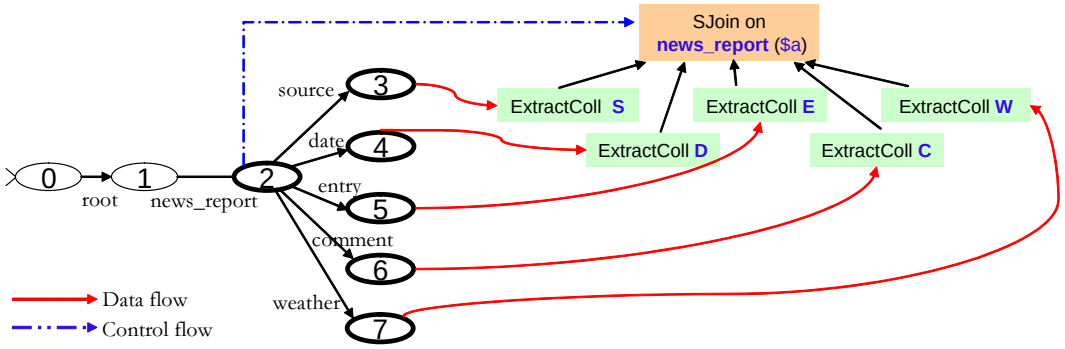

Figure 6.2: Raindrop Query Automaton

Automaton is a widely used technique for pattern retrieval over XML token streams. Here we describe the common features of automaton that serve as the core of most automaton models in pattern retrieval [DF03] [SRM06] [GMOS03]. Figure 6.2 shows the automaton for retrieving the patterns in *Q1*. Each expected pattern ends in a final state. A stack is used to store the history of state transitions. Figure 6.3 shows the snapshots of the automaton stack after the first 13 tokens (*Example 1*) have been processed for *Q1*. The Extract operators extract the *source*, *comment* elements, etc.

6.2 Optimization Modules

6.2.1 Extended System Architecture

Figure 6.4 shows the *Raindrop-Plus* system framework, which is built upon the Raindrop XQuery engine.

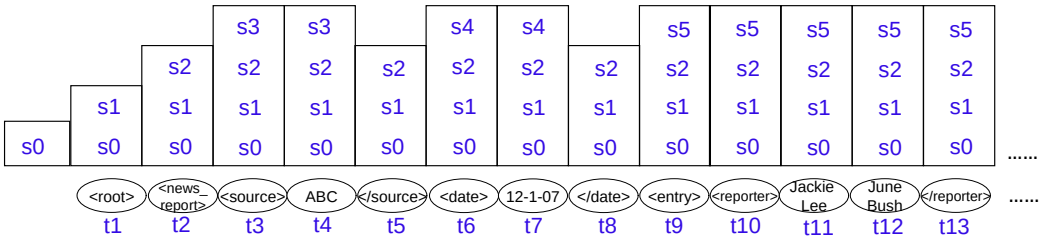

Figure 6.3: Stack Storing Automaton State Transitions

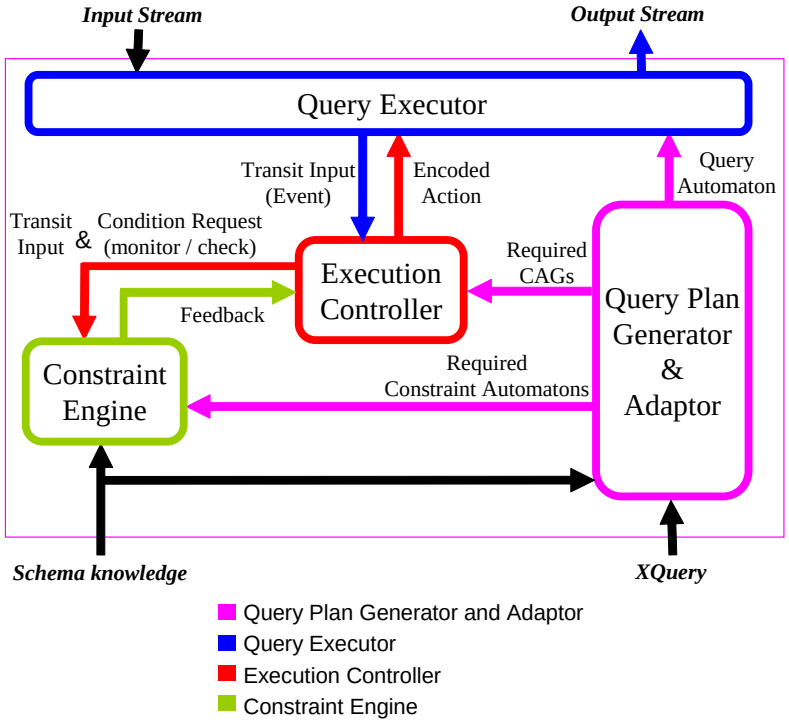

Figure 6.4: Raindrop-Plus System Architecture

6.2.2 Constraint Engine Based on Glushkov Automaton

Glushkov Automaton. For an one-unambiguous regular expression, an equivalent deterministic finite automaton called the Glushkov automaton (GA) can be constructed in quadratic time [KD98]. Glushkov automata have the properties that:

(1) every state in a Glushkov automaton corresponds to a symbol in the marked regular expression, and (2) every transition has one and only one destination state. For a formal definition of Glushkov automata and its construction, please refer to [KD98].

The automaton in Figure 3.3 is in fact a Glushkov automaton for the pattern *news_report*. We can observe that in a GA, there is a one-to-one mapping from its automaton state to the symbols in the corresponding regular expression. The algorithm of computing POS (Section 3) is the same as before. Due to the convenience of its construction and simplified automaton states, in our *Constraint Engine*, GA is used for each binding type where PNO constraints are being monitored.

Chapter 7

Experimental Evaluation

7.1 Experimental Setting

We have implemented the SQO techniques in Raindrop [SRM04] using Java 1.4. Experiments are run on two Pentium 4 3.0G machines, both with 504MB of RAM. One machine sends the XML stream to the second machine, i.e., the query engine (Figure 7.1). 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. The following studies the experimental result of our proposed techniques.

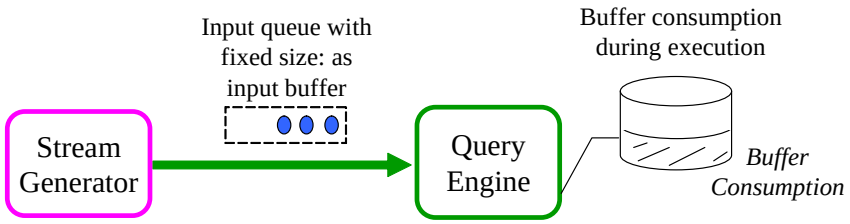

Figure 7.1: Experimental Setting

7.2 Experimental Results

We now report the performance of our SQO techniques on *news* data shown in our previous examples. We applied a generated *on-line news* data set for the experiments. We design a set of queries with predicate filtering. By changing the predicate position and selectivity, the proposed SQO technique should be able to minimize the amount of data that is buffered: with a smaller selectivity or an earlier predicate (position is smaller), less data needs to be buffered. The experimental data shown in Figure 7.2 provides the verification.

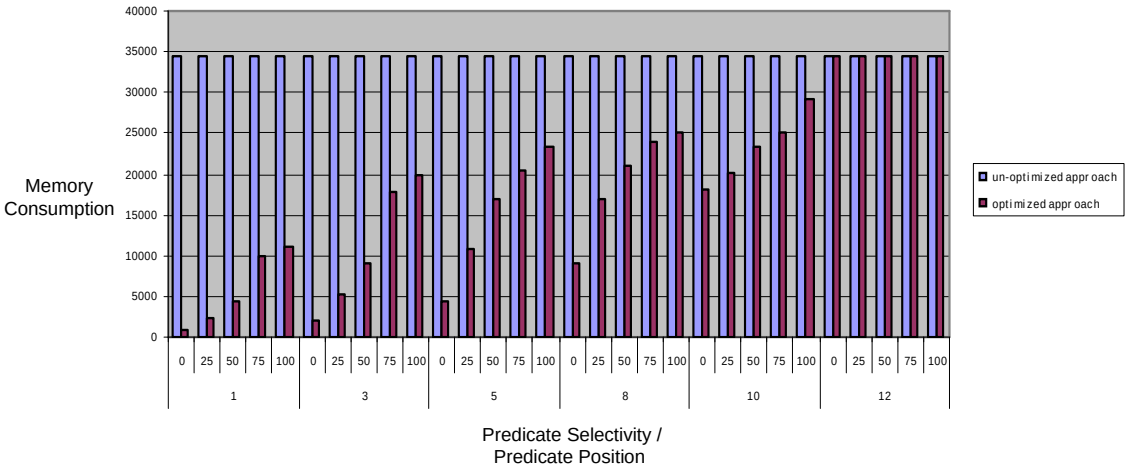

Figure 7.2: Buffer Avoidance by Applying

How much the proposed SQO technique can enhance the query evaluation depends on how much buffering has been avoided. We control the portion of the buffering that can be avoided based on the above design in the above query set. Figures 7.3 reports the results for our given dataset. In the generated data, each element has the same size and cardinality. 30 different queries, with the predicate position varying from 1 to 12 and selectivity varying from zero to 100% are evaluated.

Figure 7.3 shows the chart combining the two varying factors on predicate position and selectivity. We can see that more avoidance on data buffering generally leads to a bigger enhancement in CPU performance. 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%.

By fixing the selectivity at 100% and the predicate position at the right-most end, we can show the overhead of our proposed SQO techniques (Figure 7.4 and 7.5). For example, while the selectivity is 100% when the predicate position is at the right-most, none of the monitoring checking will ever lead to any buffer avoidance. The performance difference between such a plan and the original plan is then the worst case overhead of SQO in the worst case. Due to the introduced overhead of PNO monitoring, the optimization execution approach becomes more expensive than the un-optimized solution.

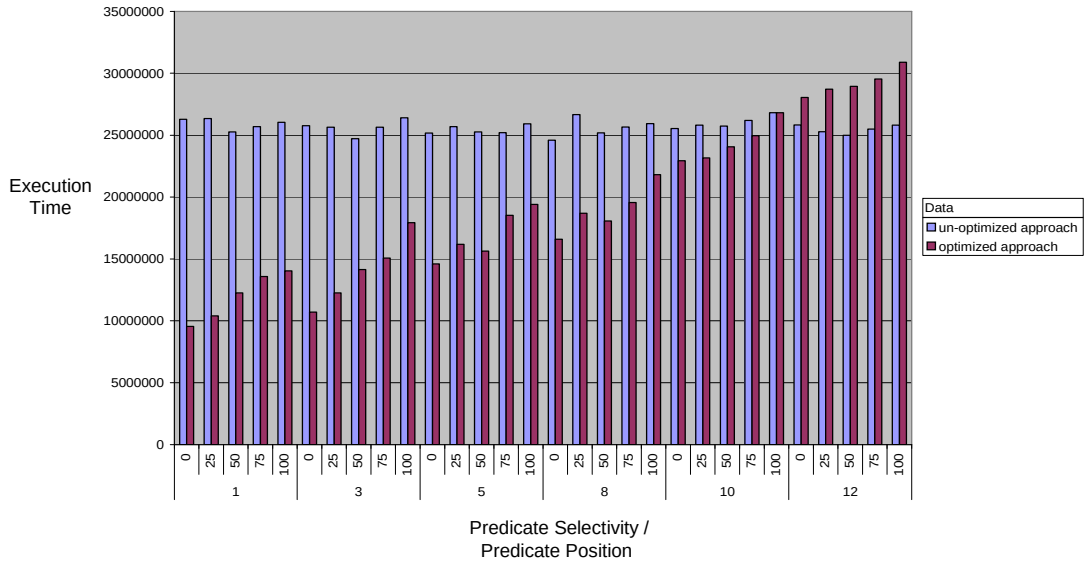

Figure 7.3: Gain on CPU Performance by Buffer Avoidance

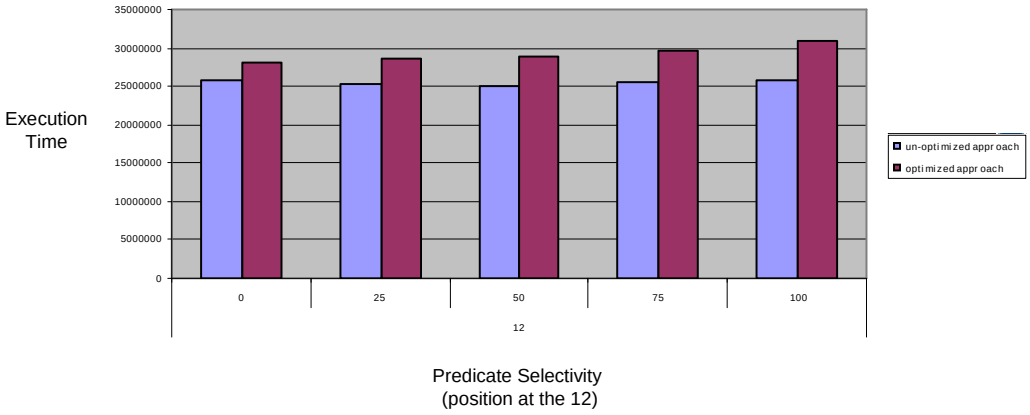

Figure 7.4: Overhead of the Proposed SQO Technique (I)

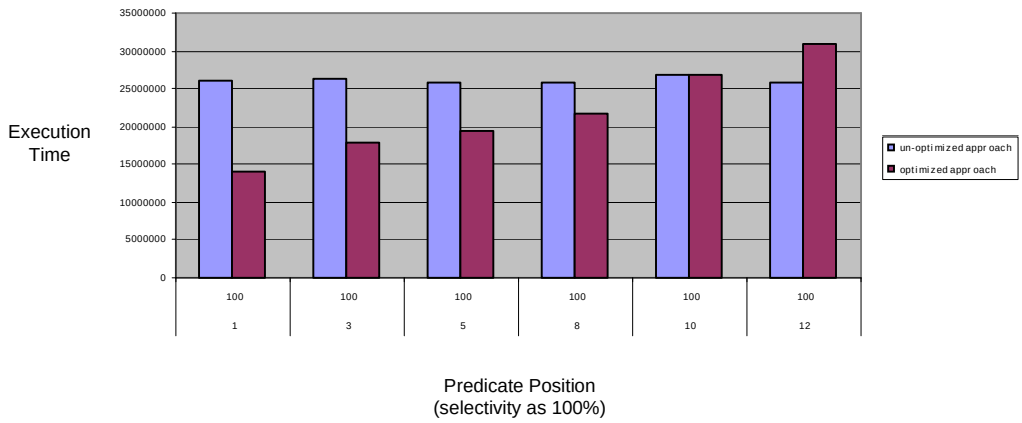

Figure 7.5: Overhead of the Proposed SQO Technique (II)

Our experiments reveal that the proposed SQO is practical in two senses. First, the technique can surely reduce the memory consumption. Second, in most cases, the savings brought by the techniques on CPU performance can be significant. We can

observe that in the following situation(s) where the memory consumption might be considerably large, the effect of avoiding the unnecessary memory footprint during query evaluation is not trivial:

(1) Target patterns have large element sizes. For example, in *Q1* above, storing *comment* element may be costly if the *comments* are large;

(2) Target patterns of large cardinality. For example, many *comment* elements may exist in a returned news element of *Q1*;

(3) With deep level of nesting. For example, with a query like *FOR $o in bound path P, RETURN P1,P2,...,Pn*, P1 to Pn are all under such "FOR...RETURN" structure and so on. In such a query with deep nesting structure, execution without an efficient buffer strategy might be very costly.

Chapter 8

Related Work

Projecting XML [MS03] [BCCN06] [SSK07] aimed to address the problem of reducing memory by pre-filtering the data from the input stream based on the paths from the query. [BGKS03] utilized a pre-computed index to reduce the memory and CPU costs. However, these solutions do not meet the requirement of typical stream applications, where a large amount of data input is processed on the fly and no pre-computed metadata is provided.

On-the-fly query evaluation for XPath queries has been studied in [GGM+04] [CDZ06] [CDZ05]. Such techniques are not suitable for XQuery evaluation. A data-transformation query language such as XQuery, which raises new challenges for the query evaluation, has been studied in several projects [DF03] [PC03] [LA05] [LMP02] [SJR03] [SRM06]. Commonly these XQuery engines try to address XQuery on streams using automaton / transducer-networks for pattern retrieval and introducing stream-specific operations to perform data filtering and data result re-construction.

XHints [GC04] extends SIX by supporting predicates and online index generation using only partially buffered streams. However, this work requires metadata being embedded in the input XML streams. It aims to avoiding the stack operations when

further pattern retrieval is not needed for an element. This does not help in cutting the unnecessary memory footprint during query evaluation.

[DF03] evaluates multiple XQueries over XML streams using an execution strategy similar to in-time execution strategy. It tries to perform optimization in the XML stream context by using schema knowledge to decide whether results of a pattern are recursion-free and what types of child elements can be encountered respectively. It also tries to avoid unnecessary pattern retrieval instead of cutting memory footprint during evaluation.

[SRM05] and [KSSS04] are the closest to our work in this thesis. The goal of [KSSS04] is to minimize the buffer size by directly outputting tokens of some extracted patterns. It considers only very limited cases. For example, it cannot switch the output strategy of a pattern from "buffer" to "output" at runtime. They also do not support filtering-related computations. [SRM05] 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 utilizes the in-time execution strategies so it cannot perform join-related computations incrementally nor other aspects of the filtering-related optimization except the above early data purging. Further, it cannot completely utilize the constraint knowledge as its algorithm introduces a new pattern to indicate a pattern's "completeness" under a bound element. Hence it does not capture all the complex constraints that can be expressed by a regular expression.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

XML and XQuery have been widely accepted as the standard data representation and query language for web applications such as web services and on-line data delivery. The memory footprint in XML stream processing can be decreased by applying schema knowledge of the input data. We reason about pattern non-occurrence constraint (*PNO* Constraint) and develop an automaton-based technique to utilize schema knowledge for runtime PNO constraint detection.

Second, we identify possible optimization opportunities for memory footprint minimization, which can be triggered by runtime PNO detection. We introduce the condition-action graph (*CAG*) to encode optimization decisions and propose optimization-embedded execution strategy to execute an optimized plan. We also propose algorithms for shrinking a given CAG to ensure the optimization efficiency.

We implement our SQO technique within the Raindrop XQuery engine, and conduct an experimental study to illustrate that these techniques bring significant performance improvements in terms of memory usage. We also perform experiments to show that generally low memory consumption coincides with with a short

evaluation time (thus a better CPU usage).

9.2 Future Work

Future work includes: (1) supporting optimization on a more comprehensive query language subset than the pattern queries; (2) applying the proposed SQO technique to explore the optimization opportunities in XSLT evaluation (type-based XSLT evaluation); (3) applying the proposed SQO technique to explore the optimization opportunities in XML document projection; (4) comparison between the generic automaton-based SQO approach and the approach simply using ending marks; (5) studying the "incremental computing" model for evaluating XQuery over XML streams by a hybrid output granularity.

Bibliography

[BCCN06] Véronique Benzaken, Giuseppe Castagna, Dario Colazzo, and Kim Nguyen. Type-based xml projection. In *VLDB*, pages 271–282, 2006.

[BGKS03] Nicolas Bruno, Luis Gravano, Nick Koudas, and Divesh Srivastava. Navigation- vs. index-based xml multi-query processing. In *ICDE*, pages 139–150, 2003.

[BPSM+06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible markup language 1.0 (fourth edition). In *http://www.w3.org/TR/REC-xml/*, 2006.

[CDZ05] Yi Chen, Susan B. Davidson, and Yifeng Zheng. Vitex: A streaming xpath processing system. In *ICDE*, pages 1118–1119, 2005.

[CDZ06] Yi Chen, Susan B. Davidson, and Yifeng Zheng. An efficient xpath query processor for xml streams. In *ICDE*, page 79, 2006.

[DF03] Yanlei Diao and Michael J. Franklin. Query processing for high-volume xml message brokering. In *VLDB*, pages 261–272, 2003.

[GC04] Akhil Gupta and Sudarshan S. Chawathe. Skipping Streams with XHints. In *Technique Report of University of Maryland Collegeg Park, http://www.w3.org/TR/REC-xml/*, 2004.

[GGM+04] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[GMOS03] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing xml streams with deterministic automata. In *ICDT*, pages 173–189, 2003.

[KD98] A.Bruggemann Klein and D.Wood. One-unambiguous regular languages. In *Information and Computation, 142(2)*, pages 182–206, 1998.

[Koz03] Dexter Kozen. Automata and computability. In *W.H.Freeman and Company, New York*, 2003.

[KSSS04] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *VLDB*, pages 228–239, 2004.

[LA05] Xiaogang Li and Gagan Agrawal. Efficient evaluation of xquery over streaming data. In *VLDB*, pages 265–276, 2005.

[LMP02] Bertram Ludscher, Pratik Mukhopadhyay, and Yannis Papakonstantinou. A transducer-based xml query processor. In *VLDB*, pages 227–238, 2002.

[MS03] Amélie Marian and Jérôme Siméon. Projecting xml documents. In *VLDB*, pages 213–224, 2003.

[OMFB02] Dan Olteanu, Holger Meuss, Tim Furche, and Francois Bry. Xpath: looking forward. In *EDBT 2002 Workshops XMLDM, Czech Republic, March 24-28, 2002*, 2002.

[PC03] Feng Peng and Sudarshan S. Chawathe. Xpath queries on streaming data. In *SIGMOD Conference*, pages 431–442, 2003.

[SAV95] Richard Hull Serge Abiteboul and Victor Vianu. Foundations of Databases. 1995.

[SJR03] Hong Su, Jinhui Jian, and Elke A. Rundensteiner. Raindrop: a uniform and layered algebraic framework for XQueries on xml streams. In *CIKM*, pages 279–286, 2003.

[SRM04] Hong Su, Elke A. Rundensteiner, and Murali Mani. Semantic query optimization in an automata-algebra combined XQuery engine over xml streams. In *VLDB*, pages 277–288, 2004.

[SRM05] Hong Su, Elke A. Rundensteiner, and Murali Mani. Semantic query optimization for XQuery over xml streams. In *VLDB*, pages 1293–1296, 2005.

[SRM06] Hong Su, Elke A. Rundensteiner, and Murali Mani. Automaton meets algebra: a hybrid paradigm for xml stream processings. *DKE Journal*, pages 576–602, 2006.

[SSK07] Michael Schmidt, Stefanie Scherzinger, and Christoph Koch. Combined static and dynamic analysis for effective buffer minimization. In *ICDE*, pages 236–245, 2007.

[W3C04] W3C. XQuery 1.0 and Xpath 2.0 formal semantics. http://www.w3.org/TR/query-semantics, 2004.