Masters Thesis

Pattern Matching in XQuery

by
Aayush Deep Garg

Advisors:
Dr. Peter Fischer
Kyunars Sheykh Esmaili

Systems Group
Department of Computer Science
ETH Zürich

April 2010
XML has made a great success as the data format in the field of communication, meta data or documents. Many real world business applications generate streams of XML data items continuously. Many applications are based on analysis of patterns over this sequence of data items, such as knowledge extracted by tracking variances of price ticks in the stock market scenario; or for a HTML document formatter application which works by parsing sequence of tags detected in the document. Hence pattern matching is becoming important over sequences of XML data items. XQuery data model (XDM) is based on sequence of items, hence can be naturally used for representing data streams, so it would be reasonable to have pattern matching capability in the current XQuery model without changing the underlying “sequence of items” data model in any manner.

In this thesis we first analyzed what pattern matching capabilities including semantic expressiveness should be ideally offered by such pattern matching extension clause and derived usecases in support of our argument. Next, we analyzed what pattern matching capabilities are provided by current XQuery model and associated missing features. Further, we formularized our new syntax that offers the capability to express patterns by regular expressions and provides new operators that give freedom to express semantic reasoning in the patterns. We implemented our proposed extensions in the existing MXQuery implementation, a lightweight XQuery engine for small devices. The main contribution of this thesis is design and development of proposed pattern matching extension of XQuery and also analysis of performance comparison of patterns expressed with current XQuery model.
Acknowledgments

I would like to thank my supervisor Dr Peter Fischer for giving me the opportunity to work in Systems Group at ETH, Zürich. His constant engagement and enthusiasm for research has always been a great source of inspiration to me. I thank him for his mentorship, immense support and help throughout the work during this thesis. I have learned a lot from him during the course of our interaction.

I would like to thank Kyumars Sheykh Esmaili for his support and inputs that I had throughout the thesis. Also, he patiently performed thorough review of this thesis and helped me in rectifying errors.

I would also like to thank Nihal Dindar for valuable discussion and contributions to tackle some of our problems.

In the end, I would like to express my appreciation towards all colleagues and friends at D-INFK, ETH Zürich.
Contents

1 Introduction 1
1.1 Motivation ........................................ 1
1.2 Background ........................................ 2
   1.2.1 XML, XPath, and XQuery ...................... 2
   1.2.2 Patterns ...................................... 5
1.3 Related Work ....................................... 7
   1.3.1 Pattern Matching In Relational Database Systems .... 7
   1.3.2 Pattern Matching In Data Streaming Systems .......... 8
   1.3.3 Other Works .................................. 8
1.4 General Requirements For Pattern Matching ............... 8
   1.4.1 Overlapping and Non-overlapping Patterns ........... 9
   1.4.2 Quantifiers To Choose .......................... 9
   1.4.3 Type Of Elements Selected ...................... 9
   1.4.4 Semantic Expressiveness of Extension ............... 10
1.5 Thesis Structure ..................................... 10

2 Existing Pattern Matching Features In XQuery 11
2.1 String Matching .................................... 11
2.2 Windowing Extension ................................ 12
   2.2.1 Overview ..................................... 12
   2.2.2 Using Windowing Extension For Pattern Matching ... 13
   2.2.3 Simple Patterns ................................ 14
   2.2.4 Complex Patterns .............................. 16
   2.2.5 Necessity Of Using Landmark Windows .............. 19
2.3 Need For A New Extension ........................... 20
   2.3.1 Limitations Of String Matching .................. 20
   2.3.2 Limitations Of Windowing Extension ............... 20
2.4 Additional Requirements For The New Extension .......... 22
   2.4.1 General Predicates ............................. 22
   2.4.2 Subsequence Content Access Mechanisms ............ 22
   2.4.3 Quantifiers in Query ............................ 22

3 Extending XQuery with Pattern Matching Facility 23
3.1 Design Possibilities For Extensions ................... 23
3.2 Overview of the Proposed Extension .................... 25
   3.2.1 Window Type Clause ............................ 26
   3.2.2 Type Declaration ............................... 27
   3.2.3 Selection Clause ............................... 27
Chapter 1

Introduction

1.1 Motivation

There is a growing interest in the area of applications that deal with finding patterns over the sequences of data items. Ideally these data items could be
the part of continuous stream being fed into the pattern matching applications. Here we would assume that these applications deal with the sequence of XML
data items. These applications could be typically:

- Security applications where detection of unusual behavior definable with
  regular expressions is needed.
- Financial applications where detecting stock patterns are critical.
- Fraud Detection applications including RFID processing where tracking
  of valid paths for RFID tags is needed.
- HTML document formatter application which parses with sequence of tags
  detected in the document.

Today many pattern matching proposals with different syntax and semantics
exist but most of the proposals focus on the relational model of the data[5, 9]
finding the patterns in sequence of rows. As these proposals deal with the rela-
tional model of the data, most of them propose syntactic extensions to SQL
and specifying pattern using regular expressions.

Here we will explore the possibilities of pattern matching over the sequence of
XML data items. In this case, SQL is not appropriate to process XML data.
It can neither directly read XML data, nor generate XML data if the output is
required to be XML (e.g., an RSS feed or a new paginated document).

XQuery is the standard query language of the XML to select data elements of
interest, reorganize, transform and process it in a flexible way. XQuery data
model (XDM) is based on sequence of data items and in XDM most of the
expressions have input/output as XDM instances. Hence it would be reasonable
to have pattern matching facility in XQuery as it would be nested with same
other XDM instances and this pattern matching facility should well integrate
with other XDM expressions i.e. input/outputs as XDM instances. Also, we are interested in pattern matching in XQuery due to following reasons:

- Sequences of XML data (XDM) are part of many real world applications
- No comprehensive solutions for specifying pattern queries in XQuery exists yet

A closely related concept to pattern matching is predicate based windows\[1\] in which windows over data streams are not defined based on time or frequency of items but user is free to specify opening and closing conditions of a window based on actual values of items. Predicate-based windows cover a significant aspect of the requirements from the relational cases already, since they allow specifying the window bounds in a flexible and expressive way. The limitation of predicate-based windows are that specifications over the contents can only be made when the window has been bound/completed. This also motivates us demanding for a change in the clause.

Pattern matching is in itself a complex and tedious task. A pattern is composed of its constituent sub-patterns. To identify a pattern correctly, it is very important to define the general correct semantics valid for the whole pattern and also valid semantics for the sub-patterns that are part of the main total pattern. This involves representation of the required pattern in the proposed syntax and last but not the least, XQuery Data model should not have any side effect from any new proposal i.e. it should operate on sequence of data items. In this thesis, we will develop such syntax for pattern extension in various phases discussing the role of each construct.

In the following sections, we will start with the basics of XML and XQuery. It is not in the scope of this document to go in every detail of it but we would keep our focus to basics and further extend to our problem of pattern matching. We will also discuss pattern matching related to sequence of XML data items and contribution of XQuery related to it. Further, we will also discuss what pattern matching capabilities are ideally needed from a pattern matching extension and explain these with the help of our usecases.

1.2 Background

1.2.1 XML, XPath, and XQuery

XML is a W3C \[13\] recommendation for document markup and stands for Extensible Markup Language. XML is a markup language much like HTML and it was designed to carry data, not to display data. In XML, information can be described using textual manner and interspersed with markup that fragments the data into hierarchical tree like structure. An XML document is an ordered, labeled tree. Each node of the tree is an XML element and is written with an opening and closing tag. An element can have one or more XML attributes. In the sample XML document below, the book element is enclosed by two tags \(<book>\) and \(</book>\). It has attributes named title, year and one child element named author.
1.2 Background

XML has many interesting characteristics many of which make it attractive for many applications. In general we can list following characteristics of XML:

- XML is human as well as machine readable.
- It is widely accepted as data representation format and used in several applications.
- It is platform independent.
- It is internationalized.
- Due to the hierarchical structure, it can describe trees, list, records etc.

As an XML document is a tree-structured (hierarchical) collection of nodes, it is useful to specify a path that points to a particular node in the hierarchy. XPath expression specifies a “textual” pattern that selects a set of XML nodes or more generally any sequence allowed by the data model. A node is either a document, an element, an attribute, a text, a comment, a processing instruction or a namespace node. XPath [15] expressions can make use of combining index addresses, wildcards and a whole set of string, positional, boolean, conversion and node related functions etc. For example, following XPath expression selects all &lt;author&gt; elements that are children of &lt;book&gt; element in the XML shown above:

\[ \text{book/author} \]

The full range of XPath expressions takes advantage of the wildcards, operators, and functions that XPath defines.

We need to have a declarative query language for XML similar to what SQL is for relational data. Unfortunately relational data and XML data differ in several aspects, for instance relational data is flat while XML data is nested and relational data is regular and homogeneous while XML is not (without an XML schema). Even from processing point of view, relational data is processed at tuple level and that works well for the relational model, where tuples have uniform size. On the other hand, it is not appropriate for XML. A given XML node can be arbitrarily large, for example, document node and this can dramatically vary across nodes. All for these reasons, the well-established query language SQL is not applicable or easily extensible to process XML data and the W3C started to create and standardize XQuery [14] as a special query language.

XQuery is a powerful language for finding, extracting and transforming data in XML. An XQuery query contains always one expression (inclusive a prolog with function and variable declarations) and this expression itself can again contain other (sub-) queries which declare the operators of this expression. It uses XPath to access parts of an XML document, applies operations on the filtered elements and returns a value.
XQuery has its own data model which is shared between XPath and XSLT [16]. In the XPath and XQuery Data Model(XDM), every value is an ordered sequence of zero or more items, which can be either an atomic value or a node. An atomic value is one of the atomic types defined by XML Schema or is derived from one of those. A node is either a document, an element, an attribute, a text, a comment, a processing instruction or a namespace node. XQuery is a declarative, high level, side-effect-free and read-only language to query XML data sources. XQuery is closed under the abstract XQuery data model(XDM), meaning that the inputs, intermediate results, and output of an XQuery expression are all instances of this data model. Several XQueries can be connected to each other, provided that the input to any XQuery represents the XDM instance.

**FLWOR Expressions**

*FLWOR* expression [14] supports iteration and binding of variables to intermediate results. The name *FLWOR*, pronounced “flower”, stands for the keywords *for*, *let*, *where*, *order* by, and *return*. A sample *FLWOR* expression is:

```
for $s in //db/name
let $a := //book[author = $s]
where $a/score > 10
return $a
```

The *for* and *let* clauses in a *FLWOR* expression generate an ordered sequence of tuples of bound variables, called the tuple stream. Therefore, every *FLWOR* expression must have at least one *FOR* or *LET* clause to generate tuple stream. This kind of expression is often useful for computing joins between two or more documents and for restructuring data. The optional *where* clause serves to filter the tuple stream, retaining some tuples and discarding others. The optional *order* by clause can be used to reorder the tuple stream. The *return* clause constructs the result of the *FLWOR* expression. The *return* clause is evaluated once for every tuple in the tuple stream, after filtering by the *where* clause, using the variable bindings in the respective tuples. A *FOR* or *LET* clause also contain one or more variables, each with an associated expression. But unlike a *FOR* clause, however, a *LET* clause binds each variable to the result of its associated expression, without iteration. We can illustrate by the following example [14] showing this property:

```
let $totalBinding := (<one/>,<two/>,<three/>)
return <out> {$totalBinding} </out>
```

The following output is produced:

```
<out>
<one/>
<two/>
<three/>
</out>
```

So as we can see above, *let* here binds without the iteration. Let’s write the similar query but now containing the *for* clause:-

```
for $s in //db/name
let $a := //book[author = $s]
where $a/score > 10
return $a
```

The following output is produced:

```
<out>
<one/>
<two/>
<three/>
</out>
```
1.2 Background

for $s$ in (<one/>,<second/>,<three/>)
return
$out$ {$s$} $</out$

The output in this case would be:-

$out$
  $<one/>$
$</out>
$out$
  $<two/>$
$</out>
$out$
  $<three/>$
$</out$

The tuple stream generated by the for clause is as follows:-

($s = <one/>$)
($s = <two/>$)
($s = <three/>$)

Besides this, for and let clauses may have an optional type declaration and an associated positional variable [14] that is bound at the same time.

XPath Expressions

We earlier mentioned that we can use the XPath within an XQuery as they share the same data model. So we can use XPath expressions as standalone or within an XQuery and both are said to be valid XQuery expressions. Consider the following XPath:

`doc('test.xml'/a/b)`

It selects all the b nodes from the test.xml file which have a as the parent. This standalone can be regarded as XQuery or it could be a part of the XQuery because its output is an XDM instance and that can act as input to another XDM instance i.e. also another XQuery expression.

1.2.2 Patterns

A pattern is a type of theme of recurring events or objects in which elements repeat in a predictable manner[17]. Let’s take a simple pattern in which our required pattern is occurrence of three consecutive 'A' elements i.e. (AAA) in the following sequence $S$ of elements:-

$S = (A, A, A, B, A, A, B, A, A, A)$ (1.1)

Clearly, there are two matches of our required pattern in the above sequence 1.1. For simplicity, let’s assume that the first element is at position number 1. Then our pattern match instances and corresponding positions are:-

- (A, A, A) and the corresponding positions (1, 2, 3)
• (A, A, A) and the corresponding positions (9, 10, 11)

Pattern in sequence 1.1 was simple and clearly observable. But a pattern can itself be complex and may require many semantic parameters to be clearly defined. Suppose we have the following sequence instead of 1.1 and we try to match the same pattern:-


Now, the pattern match instances and corresponding positions can be stated as:-

• (A, A, A) and the corresponding positions (2, 3, 4), (3, 4, 5), (4, 5, 6)
• (A, A, A) and the corresponding positions (9,10, 11), (10, 11, 12)

As we can see above, one pattern element is part of multiple pattern matches, this poses different open questions. In order to properly identify a pattern, it is needed to be defined from the user whether a new pattern matching should be started while a pattern is being matched as in the case of (2, 3, 4) here. Similar other overlapping cases can be defined such as totally overlapping patterns, partially overlapping patterns etc.

Expressing a pattern

Patterns can be represented in a number of ways. The two approaches that are most common of them are:

• Regular expressions are composed of set of variable that uses quantification or boolean operators and are the most established way of defining a pattern. Suppose a regular expression is \( A^+B^+ \) and the sequence is:

\[ S = (B, A, A, B, C, A, B) \] (1.3)

In our regular expression, variable \( A \) denotes the occurrence of element \( A \) in the sequence 1.3. Following are the pattern match instances generated by the above regular expression (considering every possible match):

\{B\}, \{A, A, B\}, \{A, B\}, \{A\}, \{B\}

We can also define regular expressions in which variables imply predicates for the selection of elements from the sequence. These predicates imply the semantic condition on the sub-pattern which it is referring to. Let\( s \) assume that we want to detect a pattern of increasing value of elements in a sequence immediately followed by decreasing value of elements in the same sequence. We can observe that consecutive elements in this pattern are correlated according to the condition specified (here it is increasing sequence i.e. every element is greater than previous element). The regular expression for such a pattern can be represented as, \( A^+B^+ \), where \( A \) and \( B \) represent a subsequence (aka subpattern) of the whole pattern.
1.3 Related Work

We define $A$ as the subsequence representing increasing value, i.e., every element in this subsequence is greater than the previous element (except first element since this subsequence represents the start of the pattern). Similarly $B$ is the subsequence in which every element is less than the previous element (including the first element of $B$) since it starts immediately after the subsequence $A$. Taking a set of elements $S$ as below:

$$S = (1, 5, 10, 8, 2, 2, 4, 6, 3, 1)$$  \hspace{1cm} (1.4)

The pattern match below (shown as *PatternMatch*) for the sequence of elements shown in sequence 1.4 and also individual subsequences (corresponding to variable $A$ and $B$ in regular expression) can be shown as following:

- $\text{PatternMatch} = (1, 5, 10, 8, 2) \rightarrow A = (1, 5, 10), B = (8, 2)$
- $\text{PatternMatch} = (2, 4, 6, 3, 1) \rightarrow A = (2, 4, 6), B = (3, 1)$

- Temporal logic is used to describe any system of rules and symbolism for representing, and reasoning about, propositions qualified in terms of time[18]. Temporal logic has two kinds of operators: logical operators and modal operators to define the logic. In this thesis we are using regular expressions throughout for analyzing the pattern.

Pattern matching over sequences of XML Data Items

In many cases, the sequences of data items on which we are interested in doing pattern matching, are represented in XML format, forming a XML stream. XQuery data model (XDM) is based on sequence of items and binding variables to sequences is its part of natural behavior. Therefore XQuery is the natural candidate to be extended for pattern matching extension due to the XQuery data model (XDM). The binding variables to sequences in XQuery can represent the whole pattern as well as the subsequence within the pattern as we mentioned in the section 1.2.2. Clearly, we need to do the same in the pattern matching i.e. to bind variables to the appropriate sequences. This presents a good base to start pattern matching extension with the XQuery because it does not make any change in the underlying “sequence of items” data model.

1.3 Related Work

There is lot of previous work done in the field of patter matching. We can classify it according to pattern matching in field of relational database systems and pattern matching in Data Streaming systems.

1.3.1 Pattern Matching In Relational Database Systems

ANSI 2007 proposal defines the MATCH_RECOGNIZE clause as a pattern matching extension in SQL[10, 9] and we also studied DejaVu[5] system which is an event processing system that integrates declarative pattern matching over live and archived streams of events on top of novel system architecture. DejaVu system implements MATCH_RECOGNIZE clause and shows promising results in the area of queries for complex event processing (CEP) applications.
We find another SQL language extensions proposal for pattern queries in SQL-TS (Simple Query Language for Time Series) [4]. SQL-TS is a construct for specifying complex sequential patterns.

As we earlier mentioned, SQL is not appropriate to process XML data. It can neither directly read XML data or process it. So these extensions were not directly helpful for analyzing sequences of XML items. Pattern matching for XML is very different from relational world as XML provides much more semantic view over the data in comparison to relational data. Naturally pattern matching over XML items is composed of this semantic knowledge. For example, a pattern matching query for XML data should provide expression to express type of the elements to be selected. But MATCH_RECOGNIZE gives fair idea in general about the pattern matching and concept of some of our general pattern matching constructs are inherited from it.

1.3.2 Pattern Matching In Data Streaming Systems

We will mention two main proposals, SASE+ [3] and Cayuga [2], that propose complex event language for streaming data. SASE+ is a complex event language that supports Kleene closure over event streams, and provide a formal analysis of the expressibility of this language. SASE+ is an agile language and concentrates on streaming data. The semantic model consists of a nondeterministic finite automaton (NFA) combined with a match buffer.

Cayuga is the project from Cornell for scalable event processing. It presents a query language based on Cayuga Algebra for naturally expressing complex event patterns. This query language uses many SQL like constructs. Cayuga is a stateful pub/sub system based on nondeterministic finite state automata.

SASE+ is useful for design of a pattern matching clause and we got the general idea from it about the event selection strategies. Here our aim is to integrate pattern matching with existing XDM as it works on sequence of data items only.

1.3.3 Other Works

We find another interesting work titled XQuery Design Patterns [19] which sounds same because of our title Pattern Matching. This work deals with architectural problems or design pattern problems in XQuery while we are interested in having core pattern matching on XDM sequences. The former is on the same lines as of standard design patterns. Another area, which sounds similar is XSLT node/structure matching, in which standard XQuery/XPath expression are used to identify nodes. Here we are interested in the correlations between the sequence of data items, that inherently defines a pattern.

1.4 General Requirements For Pattern Matching

Pattern matching is in itself complex and poses many challenges. For example same pattern can be disjoint with other pattern or partially/completely overlapping with other pattern, same pattern could be expressed by patterns of different lengths etc. In this section, we will discuss various emerging issues in
pattern matching and also features desirable for any pattern matching extension. We will also see what properties a pattern matching extension could offer in order to overcome these issues.

Let’s also define some terms that we will frequently use here. Suppose a pattern expression is denoted by regular expression such as, \( A^+B^+ \), here \( A^+ \) and \( B^+ \), both together define the pattern. \( A^+ \) is a constituent of the total pattern and hence we can call \( A^+ \) as the sub-pattern or sub-sequence for \( A^+B^+ \).

### 1.4.1 Overlapping and Non-overlapping Patterns

Suppose we need to identify a pattern within any large window boundary. There could be multiple pattern matches. These patterns could be individually distinct from each other, can be partially overlapping or completely overlapping to each other. There are many real-world usecases where any individual/both types of patterns are needed. Suppose we need to find the pattern given by regular expression \( AB^+C^+ \). Considering the following example sequence:

\[
S = (A, B, B, C, C) \tag{1.5}
\]

We have completely overlapping patterns case here as \( (A, B, B, C) \) and \( (A, B, B, C, C) \). Similarly we can show partially overlapping and other cases also. So a pattern matching clause should provide constructs powerful enough to express the choice over types of the pattern selection.

### 1.4.2 Quantifiers To Choose

Quantifiers are postfix operators that express “quantification” of the choices that can be made. For example, the well-known ‘\(*\)’ quantifier express zero or more matches, and ‘\(+\)’ allows to choose one or more matches. There are also two classes of quantifiers, Greedy and reluctant. Greedy quantifiers(‘\(*\)’, ‘\(+\)’) try to match as many elements as possible while reluctant quantifiers(‘\(?\)’, ‘\(?\)+’, ‘\(?\)?’..) try to match as few elements as possible. The pattern matching clause should provide provision to express quantifiers. Now, as we can see there could be always(except single element) multiple matches for greedy and reluctant quantifier. The pattern matching expression should also be powerful enough to express:

- Match longest match of all matches of greedy quantifier.
- Match shortest match of all possible matches of reluctant quantifier.
- Report all possible matches, there should not be effect of quantifier chosen.

### 1.4.3 Type Of Elements Selected

Since we are concerned here with XML elements, pattern clause should be powerful enough to define the type of the elements that needs to be selected for the pattern.
1.4.4 Semantic Expressiveness of Extension

As we earlier mentioned, most of the real world patterns have at least the relation between every element with its other consecutive element and that is governed by the definition of that pattern. Now, there could be many extensions for defining specific sub-patterns, such as the sub-pattern depending on the immediate element preceding the first element of this sub-pattern or simply putting, a sub-pattern depending on some aggregation function on any of the preceding patterns. Clearly, pattern matching extension should have basic semantic expressiveness to define sub-patterns in a clear manner like functions for any pair of consecutive elements of a sub-pattern. This can be done by introduction of additional variables within the pattern clause, with this pattern matching would be wholly modularized in itself when individual sub-patterns just need to be aligned in the right order. Summarily, individual sub-pattern should be semantically totally expressible so that it can wisely be matched with other sub-patterns.

1.5 Thesis Structure

The remainder of the thesis is organized as follows:

- Chapter 2 reviews the currently available pattern matching capabilities in XQuery and the window extension of current XQuery model.
- Chapter 3 describes our pattern matching extension proposal and various examples associated.
- In Chapter 4, we will provide real world examples using our proposed pattern matching extension.
- Chapter 5 describes the implementation of the proposed Pattern matching extension. We will also present some benchmarking results with the current XQuery model.
- Chapter 6 finally derives the conclusion of our work with possible future work.
Chapter 2

Existing Pattern Matching Features In XQuery

We provided initial overview of the pattern matching in the last chapter. In this chapter, we will explore the existing pattern matching capabilities in the XQuery. We can mainly divide existing pattern matching in current XQuery model in two groups. In first part, XPath and XQuery provide traditional pattern matching based on regular expression matching in textual context and in the other, pattern matching provided by the window extension of the current XQuery model[1].

2.1 String Matching

In current model, XPath and XQuery provide traditional pattern matching based on regular expression matching in textual context. All these follow the common model as “whether a string matches the provided regular expression”. For example, the following regular expression based functions are provided:-

- \texttt{fn:matches(subject, pattern)} takes a subject string and a regular expression as input. If the regular expression matches any part of the subject string, the function returns true. If it cannot match at all, it returns false.

- \texttt{fn:replace(subject, pattern, replacement)} takes a subject string, a regular expression, and a replacement string as input. It returns a new string that is the subject string with all matches of the regex pattern replaced with the replacement text.

- \texttt{fn:tokenize(subject,pattern)} is like the “split” function in many programming languages. It returns an array of strings that consists of all the substrings in subject between all the regex matches. The array will not contain the regex matches themselves.

Also, there are other functions similar to \texttt{like} function in SQL. All these functions can be well used wherever string comparison is done in XPath or XQuery. Summarily, all these functions do provide a match but there is no facility of
Existing Pattern Matching Features In XQuery

2.2 Windowing Extension

The window extension of the XQuery gives the facility of selecting subsequences over a possibly infinite stream. It proposes a new language construct for XQuery called \texttt{FORSEQ}, that integrates seamlessly into the \texttt{FLWOR} expression. Detailed description of \texttt{FORSEQ} clause can be found in [1], here we will describe basic features.

2.2.1 Overview

In \texttt{FORSEQ}, four different kinds of bindings are supported to associate the variable to a subsequence. This is called as general binding (General \texttt{FORSEQ}) and three different types of window bindings. The General \texttt{FORSEQ} binds the variable to all possible sub-sequences of the source sequence. The other three types of window bindings are landmark, tumbling and sliding windows. Landmark windows are actually defined as the type of window where the start boundary cannot move, in essence the start position is identical for several windows. Sliding windows excludes all the windows with more than one match for the end expression. Tumbling windows on the other hand guarantee that there can not be any overlapping windows. Figure 2.1 shows the graphical view of window construction[1] for constructs \texttt{let}, \texttt{for}, tumbling, sliding and landmark.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{window_representation.png}
\caption{Sample window representation}
\end{figure}

In addition to the mentioned above, there are \texttt{start} and \texttt{end} expressions in the \texttt{FORSEQ} clause denote the starting and end predicates on a window respectively. Further, each of these expression can apply “previous”, “current” or “next” items that are defined with respect to the current position to fix the starting point and end point of the window. The sample XQuery code for window extension code is given below:

\begin{verbatim}
forseq $s in $seq/stream/event tumbling window
  start curItem $sx when $sx/person eq "A"
  end nextItem $sy when not($sy/person eq "A")
where fn:true()
return
<window>
\end{verbatim}
This sample XQuery above creates a tumbling window over the stream of events. The window is opened with the occurrence of A element and closed whenever next item is not an A element.

Here it is also important to note that the introduction of this new language feature does not anyway changes the XQuery Data Model or other operators. In chapter 3 we will also show that our proposed language feature does not also make any effect on XQuery Data Model and our new feature can easily be introduced by other existing XQuery implementations.

With this short description of the current XQuery model, we can derive that this subsequence selection notion in the form of windows can also be realized to define patterns. After all, a pattern can be thought of as a window containing at least one subsequence. In this chapter, we will see how far we can apply current XQuery window model for the pattern matching. In section 2.2.2, we will further categorize our analysis to simple regular expressions without join and in section 2.2.4 we will work on more complex patterns.

### 2.2.2 Using Windowing Extension For Pattern Matching

In this section we will demonstrate that how we can apply previous XQuery window extension work [1] for the patterns expressed using regular expressions. We will keep ourselves confined in analyzing contiguous patterns only. These regular expressions can be broadly divided in two categories, one that are implemented without using joins over the windows and the others are implemented using joins over the windows. XQuery code for the examples given further is in the MXQuery Testing project and is located in the PatternTests/Queries/.

Also, our XQuery code below follows this sample XML structure:

```xml
<stream>
  <event time="2006-01-01T10:30:00-00:00">
    <person>A</person>
  </event>,
  <event time="2006-01-01T11:00:00-00:00">
    <person>B</person>
  </event>,
  <event time="2006-01-01T11:15:00-00:00">
    <person>A</person>
  </event>,
  <event time="2006-01-01T12:15:00-00:00">
    <person>B</person>
  </event>
</stream>
```

The above XML simply shows the time stamp associated with the entry of each person. For example, time stamp associated with first person “A” is “2006-01-01T10:30:00-00:00”.
2.2.3 Simple Patterns

Here we will analyze simple regular expressions and their corresponding XQuery implementations. These simple regular expressions can be implemented without using joins over the windows.

AB

This expression denotes the pattern in which every A is followed by the B. A sliding window is opened for matching this expression which opens up with a matching A and closes whenever B is encountered. Further in the where clause it is checked that size of the window is 2 (so that window has A and B and nothing else). The XQuery for this pattern is:

```
for $s in $seq/stream/event sliding window
  start curItem $sx when $sx/person eq "A"
  end curItem $sy when $sy/person eq "B"
  where count($s) eq 2
  return
  <window>
  {$s}
  </window>
```

Code for this XQuery is located in the directory mentioned above and the name of the file is seqAB.xq.

ABCDEFG

Extension to the above XQuery can be made to find the similar simple patterns like ABCDE... For example, the pattern ABCDEFG is matched by making a sliding window that starts from A and ends at G. The items in between are matched by accessing individual items of the window (treating it as in array fashion). Code for this XQuery can be found as seqABCDEFG.xq.

A+

For finding a pattern in which one or more A occurs, a tumbling window is created which starts with a matching A and ends whenever there is a variable not equal to A is found. The XQuery for this pattern is:

```
for $w in $seq/stream/event tumbling window
  start curItem $wx, position $wp when $ax/person eq "A"
  end curItem $wy, position $wq $when not($sy/person eq "A")
  return
  <window>
  {$w}
  </window>
```

Code for this XQuery can be found as seqA.xq. We can see that this XQuery gives the output as windows for the pattern A+ as:-

```
<window>
  <event time="2006-01-01T10:30:00-00:00">
```

We can see that there are two occurrences for this pattern match given by tumbling windows. This we can call as also a maximum match given by the tumbling windows as these windows are non-overlapping and in tumbling window a new window is not opened until previously opened window is closed. Another variation of this XQuery can be performed using sliding windows. Sliding window gets opened for every matching start condition and hence it gives all possible patterns in this case. The XQuery for sliding window implementation can be found as seqASliding.xq.

This regular expression denotes the pattern in which \( B \) is followed by at least one or more number of \( A \)'s. The XQuery for this pattern can be given as:

```xquery
for $s in $seq/stream/event sliding window
  start curItem $sx when $sx/person eq "A"
  end curItem $sy when not($sy/person eq "A")
  where $sy/person eq "B"
return
<window>
  {$s}
</window>
```

Code for this XQuery is located under seqA+B.xq.

This regular expression captures all occurrence of either \( A \) or \( B \). The sample XQuery can be written as:

```xquery
for $s in $seq/stream/event tumbling window
  start curItem $sx when $sx/person eq "A" or $sx/person eq "B"
  end when fn:true()
  where fn:true()
return
```

Code for this XQuery is located under seqA+B.xq.
Existing Pattern Matching Features In XQuery

2.2.4 Complex Patterns

In this section we will analyze the patterns that require joins over the windows in the XQuery. The basic strategy is to create different windows for each pattern variable and these windows are joined with the conditions on the window boundaries.

Join-needed patterns

Considering again the following regular expression from section 2.2.3:

\[ A^+ B \]

This regular expression can also be written using joins in the XQuery. The XQuery could be given as:

```xquery
forseq $w in $seq/stream/event sliding window
  start curItem $ax when $ax/person eq "A"
  end nextItem $ay, position $aq when not($ay/person eq "A")
  return
forseq $s in $seq/stream/event tumbling window
  start curItem $bx, position $bp when $bx/person eq "B"
  end when fn:true()
  where $bp eq ($aq + 1)
  return
<window>
  {$w},{$s}
</window>
```

Bounded repetition included patterns

Suppose we have the following regular expression:

\[ A^* B^* C \]

Let’s try to write the XQuery for this regular expression, same on the lines as given in the section 2.2.4. The outer `FORSEQ` has the `start` expression checking the occurrence of the variable A. But due to the star operator there can be zero or more occurrence of A’s. The window will only get opened when `start` expression gets matched. So the previous query like method will not work for the star operator.

There is a need to transform the given regular expression so that ‘+’ operator replaces all the star operator occurrences. Transformation is necessary because a running variable can never be bound to an empty sequence.

For example, the regular expression above can be re-written as:

\[ C \mid A^+ C \mid A^+ B^+ C \mid B^+ C \]  \hspace{1cm} (2.1)
2.2 Windowing Extension

In the expression above '+' operator has replaced the star operator. Now the separate parts of the alternation operator can be implemented using the same way described in the 2.2.3 and 2.2.4 (without or with the need of the join) for the individual regular expressions. In the end output pattern match would be the union of all the windows hence obtained. This implementation is located under PatternTests/Queries/starOperator.xq.

Some other examples:-

\[ A \left( \left( B^* \mid CD^+ \right) \right) E \] can be written as \[ AE \mid AB^+E \mid ACD^+E \]
\[ AB^*C^?D^+E \] can be written as \[ AD^+E \mid AB^+D^+E \mid ACD^+E \mid AB^+CD^+E \]

Negation included patterns

Considering the regular expression:

\[ A^+BD(!B) \]

This regular expression shows the pattern in which A comes first at least once followed by B, D and then any variable except B. Let’s start analyzing different possible queries for this pattern. We will now consider the pseudo XQuery shown in the figure 2.2.

In the XQuery shown in the figure 2.2, we create different windows for each variable and these windows are joined over the condition in the respective where clauses. Code for this XQuery is located under PatternTests/Sources/testTumblingNestedQueries.xq.

First a main tumbling window \$w \$ is created over the sample sequence, this window starts with the occurrence of A and ends with any variable except A (in this way it was ensured we catch all the A’s sequence). Now window \$s \$ is created for variable B and this B must occur immediately after the end of A’s. This was joined with the \$w \$ window in the where clause. In the same manner D should occur immediately after the B (specified in the where clause). Now we have NOT(B) in the regular expression, and any variable except B can occur. This is done in the innermost FORSEQ clause \$q \$.

The output of the above query is located under expectedResults/testTumblingA+BD(!B).xml.

We have tumbling window in the outermost FORSEQ clause \$w \$. There is one more pattern A|B|D|C (in which A if of event time =”2006-01-01T11:00:00-00:00”) was omitted because there was already a tumbling window opened when this A was encountered (tumbling windows are non-overlapping). If we use sliding window instead of the tumbling window, we will get all the three possible patterns for this case. This shows that sliding windows are more powerful for the FORSEQ clauses when we have repetitive nature of the pattern (as in *, +) because tumbling window does not open new windows when a window is already open by matching start clause. Thus when we will change the outer window \$w \$ to sliding one, we will get the all patterns which is as located under expectedResults/sliding.A+BD(!B).xml.

Let’s also take the case of landmark windows in the above example instead of tumbling window. Suppose we have the landmark window instead of tumbling in outermost FORSEQ \$w \$ clause. This produces all windows as we got in the
Figure 2.2: XQuery for the regular expression $A^+ BD(!B)$
sliding window case and extended additionally windows, the results are located under expectedResults/landmark.A+BD!B.xml.

In this way we can see landmark window also gives the non-contiguous pattern match. In some cases, Landmark windows can be applied to the repetitive variables whenever non-contiguous pattern match is also needed. Taking another example of following regular expression:

\[ A^+ (B|C)^+ \]

Here we need to join the windows of \( A^+ \) and \( (B|C)^+ \) in such a way that we get the desired pattern. Code for this XQuery is located under PatternTest/-Queries/seqA+(BORC)+.xq.

### 2.2.5 Necessity Of Using Landmark Windows

Consider the following regular expression:

\[ A^+ (B+C)^+ \]

The possible XQuery can be given as:

```xml
forseq $w in $seq/stream/event tumbling window
  start curItem $ax, position $ap when $ax/person eq "A"
  end nextItem $ay, position $aq when not($ay/person eq "A")
return
forseq $s in $seq/stream/event landmark window
  start curItem $bx, position $bp when $bx/person eq "B"
  end curItem $sy when not($sy/person eq "B")
  where $sy/person eq "C" and $bp eq $aq + 1
return
<window>
  {$w},{$s}
</window>
```

One of the possible way to solve this pattern is using landmark windows for \((B+C)\) pattern because this pattern comes repetitively. This gives us the hint that it could be matched by the windows that grow over the time even after matching the start and end condition. Hence the above regular expression can only be matched by using landmark windows for \((B+C)\) pattern. Landmark windows are well suited for repetitive patterns and have the property of extending. The new windows produced from the match of **start** expression can be eliminated by the condition in the **where** clause at the end. But the above landmark window would also output the different patterns such as **AABBBTSCD** which is clearly not desirable. Code for this landmark window implementation is located under PatternTests/Queries/landmarkA+(B+C)+D.xq.

Other way round of implementing the above pattern is to assume that we will always create a outer landmark window for the repetitive pattern (like \( B+C \) here). Pseudo Code for this XQuery is located under PatternTests/Queries/-landmark2A+(B+C)+D.xq. This landmark window can be nested and joined with the windows of the individual variables. But again it is not always trivial in matching the position variables of this landmark window and the windows
inside it (both have different position pointers). Here we need the placement of continuous $B^+C$ windows inside the landmark window as this is illustrated in the figure. But the XQuery landmark2A+(B+C)+D.xq performs the longest match of $B^+C$ in the landmark window.

![Figure 2.3: Landmark Window for matching $B^+C$](image)

Also, for every outer landmark window the inner FORSEQ’s will start executing from the beginning. There is no way to save the last longest match and performing the next match from there.

So this use case gives us the idea that we need a window which has the capabilities of landmark window, has syntactic sugar to specify directly the match of continuous patterns (like $B^+, C^+$). This type of the window would provide the direct ability to have the maximum match of the continuous pattern that can be easily joined with other windows.

### 2.3 Need For A New Extension

In this section we will define the limitations of the current XQuery model followed by additional requirements.

#### 2.3.1 Limitations Of String Matching

String matching could be a type of pattern matching but in general pattern matching queries are complex representing correlation between the data items. This clearly cannot be achieved by the string matching.

#### 2.3.2 Limitations Of Windowing Extension

As we can see these simple patterns require the nesting of FORSEQ clauses and with other conditions one needs to write more nested FORSEQ’s with the conditional expressions. Also, for every window in the inner loop that produces, all other windows in the innermost loops are also checked every time. This is also not desirable from the performance point of view. Keeping the above examples in mind, we analyze current model’s limitations and requirements.

Hence going by the examples and the different cases of regular expressions above, we can categorize following limitations:
2.3 Need For A New Extension

No support for content based window closure

There is no support in current XQuery model to close the windows based on the contents of window, like if we want to close the window over the count of the number of items.

Taking another example, let’s say a trader has the daily limit of 50 million dollars and this should not shoot up during any particular trading day. We will consider the XQuery for this case using the tumbling windows and landmark windows. In tumbling windows, the window would be closed at the change of the day and then aggregation of the trades will be performed. Clearly for this case, we will get the results only after the ‘change’ of the day is encountered.

Now for the landmark windows implementation, this is opened for the change of the day and closed over every trade and then aggregation is performed. But the overhead is quiet large of having windows for every trade of the day. Clearly some use cases can be solved by this landmark window approach but there is a overhead and limitation. From the higher level, ideally this use case seems to be of tumbling window that should aggregate the incoming trades and report as soon as limit is crossed but this cannot happen in the current model. The XQuery for this example can be found under PatternTest/Queries/TradeLandmark.xq and PatternTest/Queries/TradeTumbling.xq.

No trivial ordering in subsequences

As we already saw that we are trying to do ordering between the windows by the way of joins. On the one hand joins are costly and moreover it is always not trivial to do the ordering between these windows for the pattern. There is no direct support such that boundaries of different window can be remembered over the execution and used in the XQuery. Event the Scripting extenstion of XQuery does not help much with the ordering or binding of the windows. As an example, suppose we want to remember last window boundaries for the pattern \((A^+ B)^+\) in the following XQuery:-

\[
\text{forseq $s$ in $seq/stream/event landmark window}
\text{start curItem $sx$, position $sp$ when $sx/person eq "A"$
\text{end curItem $sy$, position $sq$ when not($sy/person eq "A")$
\text{where $sy/person eq "B"$
\text{return}
\text{<window>}
\text{($s$)}
\text{</window>}
\]

Scripting does not help in this in any way.

High cost of joins

We discussed in the examples that current model uses Join based approach. Suppose we have a pattern given by regular expression \(A^+ B^+ C\), typically instead of using joins there must be the way to parse the sequence of items straight away
for detection of this pattern. Current XQuery model complicates in the manner
to match the individual window boundaries, making joins and complex syntax
covering all these aspects. In section 6.2 we will show with our experiments that
this join based approach comes very expensive.

2.4 Additional Requirements For The New Extension

Based on the above discussion we can frame following requirements to the cur-
rent XQuery model.

2.4.1 General Predicates

As we discussed in the section 2.3.2 there is requirement to close a window based
on general predicates. It will support to close the window based on the contents
of the window. It is understood that checking this predicate over the window
would be too costly but it needs to be explored how it could be optimized.

2.4.2 Subsequence Content Access Mechanisms

Suppose we need to match the \((B^+D^+E^+C)\) pattern. The problem with current
syntax is that we can specify start and end condition of the window in \texttt{start} and
\texttt{end} clauses but there is no clear cut syntax for also specifying the variables that
are only allowed inside the window defined by these \texttt{start} and \texttt{end} conditions.
This is met partially by writing condition specific code in the \texttt{where} clause. Any
direct syntax for matching this kind of pattern would enable to have the pattern
matching for complex expressions easily. Additionally, there can be syntax to
have the maximum matches or all the possible matches of the pattern.

2.4.3 Quantifiers in Query

A regular expression is composed of quantifiers. In current model, non-trivial
logic is to be written to write pattern queries for such regular expressions. It
would be very easy and clear if quantifiers can be specified directly (as like ‘*’,
‘+’ etc) in the extended syntax.
Chapter 3

Extending XQuery with Pattern Matching Facility

In this chapter we will explore various possibilities for the new syntax and propose the new syntax extension keeping in mind the requirements that we discussed in the section 1.4. In last chapter we had gone through with some of the limitations and requirements of the current XQuery model, this prompts us to extend it in many possible ways. We will first define some basic concepts that we will be using in our discussion for the sequence of XML data items:

**Input Stream**: We consider here stream of XML items. Here we are assuming inherent ordering of data items.

**Output Stream**: Output stream also contain sequence of items that are matched to the required pattern.

**Contiguous and Non-Contiguous patterns**: XQuery standard provide the group by clause for partitioning. This group by clause can be used with the pattern matching extension for the non-contiguous matches. We are here working on only contiguous patterns. This shows that we search for consecutive items in the sequence for matching the patterns.

3.1 Design Possibilities For Extensions

In this section we will identify various high level design possibilities of the Pattern matching extension clause and integration within the existing syntax. It is desirable that the new extension should be highly composable with the FLWOR expression so that it could exist as the another clause with presently existing clauses such as for, let etc. We present here three main possible different approaches for the pattern matching syntax extension of the current XQuery model.

**Using pattern extension as part of FORSEQ**

In our first approach, the pattern matching extension is an integral part of the FORSEQ clause. By this way pattern matching clause works on the windows selected by the FORSEQ clause. Taking an example:

```
forseq $w in $seq tumbling window
```
start ......
end ....
<< Pattern matching clause >>

Here we are first selecting the window $w$ over the input stream $seq$ and then pattern matching clause is working on this selected window. Now pattern matching clause will also make subsequences that could be tumbling/sliding in nature over the already existing windows generated from FORSEQ clause. This creates unwanted complexity of looking into overlapped windows issues and setting their range boundaries. But in terms of semantics this approach is clear, that first generate the window according to the usecase provided(suppose windows over the time period) and then finding pattern inside this window. So here we have also control over our choice of defining windows that serves as the input to the pattern matching clause.

Separate pattern matching clause

One other way is to have an independent pattern clause extension(does not have any dependency on the forseq). For example, it will have its own where clause for specifying the predicates. This approach can definitely provide powerful constructs to operate on a pattern. For example, whatever we discussed in the section 1.4 can be realized by the way of special operators in this approach but on the other hand it would just segregate the window queries and pattern matching as two different approaches. It would also create the lot of additional syntaxes in the existing model. So due to the lack of the integration with other powerful constructs available, this approach wont be desirable.

Integration within FLWOR expression

In this case pattern matching extension could be integrated as a new feature in the existing FLWOR expression and is represented same as the grammar mentioned in section 3.2. As we can see for, let and where can be used as it is from the FLWOR and the proposed pattern matching extension seamlessly integrates with the FLWOR. Example pattern can be represented in the following way(more about the exact syntax we will follow up in later sections):

```
AB
```

define pattern $p$ tumbling maximal in $seq$
$a$ as element() pcur $q1$ when $q1/name$ eq "A"
$b$ as element() pcur $q2$ when $q2/name$ eq "B"
where |$b[last()]$/@time - $a[first()]$/@time| lt 1 hour
return {$a$, $b$}

This query also reflects the fact that $q1$ stated in the $a$ sub-pattern are not in the scope for $b$ sub-pattern. Another important thing is, $q1$ is not bound here to the particular element as its counterparts curItem, nextItem etc for a window in [1] in either of the $a$ or $b$ sub-pattern definitions. We will use this integration as a base for our proposed extension.
3.2 Overview of the Proposed Extension

In this section we will introduce the proposed grammar step by step. We propose to integrate \textit{PatternClause} same on the lines of \textit{WindowClause} in the current window extension of XQuery model [14]. This \textit{PatternClause} also does not affect the underlying XQuery Data model, as it also works on sequence of data items. \textit{PatternClause} defines the main overall structure of the clauses to be written for matching a pattern. The initial proposed integration of \textit{PatternClause} in the \textit{FLWOR} expression is as shown in the following snippet of grammar:

\[
\text{FLWORExpr ::= InitialClause IntermediateClause}^* \text{ ReturnClause}
\]

\[
\text{InitialClause ::= ForClause| LetClause| WindowClause| PatternClause}
\]

\[
\text{IntermediateClause ::= InitialClause| WhereClause| GroupByClause| OrderByClause| CountClause}
\]

In the above grammar, except \textit{PatternClause}, every other clause is same and taken as earlier that have been proposed in the grammar of XQuery 1.1 [14]. Next, we need a structure for defining a pattern. We can mainly group this structure on two notions:

- Clauses containing binding variable for the pattern and its associated input stream(XDM instance).
- Clauses defining regular expression and its associated semantics.

In our proposal, we propose to group the constructs as, one clause representing common properties of whole pattern mainly binding variable, associated input stream(XDM instance), general properties of pattern that we mentioned in the sections 1.4 and 1.4.2. Other clauses would represent the subsequence of the pattern having constructs for its type of elements, occurrence indicator, binding variable for that subsequence, predicates and its associated semantics.

We now define the simplest \textit{PatternClause} as following:

\[
\text{PatternClause ::= "pattern" "\$" VarName \text{(WindowTypeClause)} \text{(SelectionClause)} "in" ExprSingle \text{(PatternDefClause)}+}
\]

\[
\text{PatternDefClause ::= "\$"VarName TypeDecl \text{(PatternVars)}? "when" ExprSingle}
\]

Now we have introduced \textit{PatternDefClause}, that represents the individual subsequence of the regular expression. As we mentioned earlier, the general clause representing the common properties of a pattern is shown above as:

\[
"\text{pattern" "\$" VarName \text{(WindowTypeClause)} \text{(SelectionClause)} 'in' ExprSingle}
\]

Here \textit{pattern} keyword identifies the starting of a pattern clause in the composite query. $\text{VarName}$ represents the binding variable for the pattern matched, we will see further how \textit{WindowTypeClause} and \textit{SelectionClause} are defined for a pattern and \textit{ExprSingle} is an instance of the XDM. \textit{PatternClause} iterates over the binding sequence and sequences of tuples are generated matching the pattern from the given conditions in underlying \textit{PatternDefClause}. The binding sequence from the \textit{PatternClause} serves as the input for the \textit{PatternDefClause}.
Let’s analyze a simple XQuery for the regular expression $A^+$ over the sequence $S = (B',A',A',D',A')$ expressed using the pattern matching extension as the following:

```xquery
pattern $a$ tumbling maximal in (B',A',A',D',A')
$p$ as element()+ pcur $s$ when $s$ eq 'A'
return
<p> {$a} </p>
```

The expected output is

```xml
<p> A A </p> <p> A </p>
```

The above query selects the pattern for the regular expression $A^+$. Now we will discuss some of the constructs like tumbling, sliding shown in the query in further sections. Taking a simple example of PatternDefClause as we will keep going into details in further sections:

```xquery
$s$ as element()+ pprev $b$, pcur $c$ when $b < $c
```

The corresponding grammar for the PatternDefClause shown above was:

```
"$VarName TypeDecl (PatternVars)? "when" ExprSingle
```

This is the typical PatternDefClause in which $s$ represents the binding of the individual sub-pattern($VarName$), TypeDecl is represented by element(), OccurenceIndicator is ‘+’, PatternVars like pprev, pcur are used to define predicates over this sub-pattern. Here the result of ExprSingle is an instance of the XDM and already an ordered sequence of zero or more items. Here it is to be noted that this does not mean that items have necessarily time stamp attached.

### 3.2.1 Window Type Clause

In this section, we will discuss about the overlapping, non-overlapping issues that we earlier discussed in the section 1.4.1. We introduce WindowTypeClause in our extension for this purpose.

```xquery
WindowTypeClause := Tumbling| Sliding
```

WindowTypeClause is used in specifying whether to filter out only non-overlapping or overlapping patterns in the sequence of elements or partition once a pattern match has been detected. As we can see in the grammar, it can be specified of two types:

- **Tumbling** specifies that the next pattern would be searched after the last element of the previously matched pattern. In this way any two patterns do not overlap. In summary, Tumbling skips the whole previous pattern first and then starts the search for the occurrence of the next pattern. This is on the same lines of the tumbling window mentioned in the Window Extension of the XQuery [1]. Also in [1], tumbling windows do not overlap.

- **Sliding** can also report the overlapping patterns. It starts matching the next pattern past first element of the previously matched pattern.
Sliding can thus match the next patterns as the subsequences inside a pattern already matched. It is also on the same lines as of sliding window in [1]. Sliding window in XQuery extension for windows [1] also slides for generating subsequent windows.

Let's say there is a sequence of elements like AABBC and the pattern to be matched is $A^+ B^+ | B^+ C$ then:-

- **Tumbling** will output AABB
- **Sliding** will output patterns AABB, ABB and BBC. Here we can see that the pattern ABB is actually a subsequence of pattern AABB already matched. The pattern ABB was matched due to sliding property. (It is assumed that in above pattern example we have SelectionClause as MAXIMAL, we will see more about it shortly)

### 3.2.2 Type Declaration

The TypeDecl in PatternDefClause is given as any element type with an occurrence indicator. This refers to any element node which occurs certain number of times. The grammar for the TypeDecl can be given as:

```
TypeDecl := "as" SeqType
SeqType := "element" "(" ")" OccurrenceIndicator?
OccurrenceIndicator := "+" | "*" | "?"
```

In our proposal we are using OccurrenceIndicator for indicating quantifiers. Suppose if we want to find a pattern in which at least one or more elements having name element as A (pattern: $A^+$) then PatternDefClause would be:-

\[
\text{PatternDefClause} := \text{$a$ as element($+$) pcur $c$ when $c$/name eq "A"}
\]

or other way round, PatternDefClause can also be given as,

\[
\text{PatternDefClause} := \text{$a$ as element($+$) all $c$ when (every $y$ in $c$ satisfies $y$/name eq "A")}
\]

Here we have defined greedy quantifier '+' in the TypeDecl as element($+$), that indicates the required element should occur one more times. In the same way, other quantifiers such as '*,?*' etc can also be specified. It is to be noted that one query can be written in many ways.

### 3.2.3 Selection Clause

In our proposed extension SelectionClause provides three kinds of identifiers for the scenario of streaming XML data items where visible data items increase by one at a time, initially starting from empty partition. These three identifiers are MAXIMAL, INCREMENTAL or ALLMATCH in our pattern extension.

```
SelectionClause := Maximal| Incremental| AllMatch
```
MAXIMAL mode finds the longest matching sequence in the entire binding sequence or partitions. The priority in MAXIMAL mode is over the elements that occur earlier than other elements. By this we can ensure the element that was seen earlier was included in the pattern match. In MAXIMAL mode a match that begins at an earlier element in the partition is preferred over a match that begins at a later element. Also, for greedy quantifiers (that try to match many number of elements: +, *) in the pattern longer match is taken into account while for reluctant quantifiers (that try to match few elements as possible: *?, ??, *? etc) shorter match is taken into account.

Here it is important to note that the matches generated are always dependent on WindowTypeClause. For example, if WindowTypeClause is specified as TUMBLING then all the matches that are overlapping with the previous pattern match are discarded. Similarly, if SLIDING is specified then all the matches overlapping the first element of the previous match are discarded.

In INCREMENTAL mode partition is increased incrementally by a single element (starting from the empty sequence) and applies the MAXIMAL match to every such partition. Basically in INCREMENTAL mode we match to find the maximum pattern length up to every element that is why elements are added incrementally in the partition. Union of all such matches forms the part of the output of the INCREMENTAL mode. This is done until no more matches starting from the given element is found in a partition.

For example, if the pattern to be matched is \( AB^+ \) and the partition is \( ABBC \) then according to above description the patterns would be generated incrementally as:-

1. Partition is EMPTY and pattern NONE
2. Partition is A and pattern NONE
3. Partition is AB and pattern is AB
4. Partition is ABB and pattern is ABB
5. Partition is ABBC and pattern is NONE

Here, from Step 3 to Step 4 patterns are matched but in the extended partition of Step 4 no pattern is matched. At this stage WindowTypeClause comes into the picture and applied over the partition. Then pattern matching is resumed from the further element obtained after sliding according to the mode specified in the WindowTypeClause.

As it appears, INCREMENTAL mode can output multiple matches rooted at the same starting element. When no such pattern can be matched in the extended partition, WindowTypeClause is applied, hence now matching from other element obtained according to the WindowTypeClause.

ALLMATCH gives the all possible matches of the pattern in the binding sequence. ALLMATCH is though not equivalent but is comparable to landmark windows in [1]. In Landmark windows start boundary is fixed and windows continue
to grow. Windows are bound with a matching start condition. In ALLMATCH with every element all possible pattern matches are detected (same as windows are generated with matching start condition and windows also continue to grow). ALLMATCH works the following strategy:-

- Partition is grown same as it was the incremental manner and matches are generated.
- When a non-match or end of sequence occurs, the existing partition is collapsed.
- Now the partition is grown in the same way as mentioned in the 1st point, but after sliding to either the next element of first element of the last successful match OR from the next element to last element of the last successful match according to sliding or tumbling strategy respectively.
- This recurring process stops when no partition can be grown due to end of sequence.

ALLMATCH is same as for (Tumbling, Incremental) and Tumbling, AllMatch combination as we can see from the above description that we slide to the next element of last successful match in both the case, hence either of strategies makes no difference with respect to the pattern detection.

It is also apparent here that longest match for greedy quantifier and shortest match for reluctant quantifier does not also matter here as all possible matches are reported. In our pattern matching extension proposal ordering of the pattern matches takes the priority according to the early element in the stream. Lets start with a simple example in which we want to detect an increasing pattern in two ways showing different SelectionClauses:

```
pattern $a tumbling maximal in (1,2,3,4) $p as element()+ pprev $q, pcur $s when $q < $s return <p> {a} </p>
```

The output for the above query can be shown as:-

```
<p> 2 3 4 </p>
```

The output can be represented in two ways, we will discuss about it in the section 3.2.4.

Suppose now the query is changed a bit, and maximal is replaced by incremental as in:

```
pattern $a tumbling incremental in (1,2,3,4) $p as element()+ pprev $q, pcur $s when $q < $s return <p> {a} </p>
```

The output for the above query can be shown as:

```
<p> 2 </p> <p> 2 3 </p> <p> 2 3 4 </p>
```

These examples clearly mark the difference between incremental and maximal. We can outline the working of incremental selection mode in the following manner:-
• Partition is empty and pattern is NONE
• Partition is {1} and pattern is {1} or NONE (as stated earlier)
• Partition is {1, 2} and pattern is {1, 2} or {2}
• Partition is {1, 2, 3} and pattern is {1, 2, 3} or {2, 3}
• Partition is {1, 2, 3, 4} and pattern is {1, 2, 3, 4} or {2, 3, 4}
• End of sequence (or if not any valid match occurs) encountered, slide according to tumbling/sliding strategy

Now doing the same query with sliding WindowTypeClause and incremental SelectionClause:

```
pattern $a sliding incremental in (1,2,3,4) $p as element()+ pprev $q, pcur $s when $q < $s return <p> {$a} </p>
```

The output for the above query can be shown as:

```
<p> 2 </p> <p> 2 3 </p> <p> 2 3 4 </p>
```

Comparing the outputs of {sliding incremental} and {tumbling incremental}, both are same as we can see that no sliding strategy could be applied because no more elements can be added to the partition in incremental mode as end of sequence occurs and we have matched maximal length pattern up to every element.

Now doing the same query with sliding WindowTypeClause and allmatch SelectionClause:

```
pattern $a sliding allmatch in (1,2,3,4) $p as element()+ pprev $q, pcur $s when $q < $s return <p> {$a} </p>
```

The output for the above query can be shown as:

```
```

Comparing the outputs of {sliding incremental} and {sliding allmatch}, we can see that till the pattern match {2 3 4}, the output is same, from there onwards in sliding allmatch, it slides to the next element and it starts to grow the partition like as from the beginning, and hence it generates the {3}, {3 4}...

As we earlier mentioned, {tumbling incremental} and {tumbling allmatch} are same and hence in this case will also generate the same match.

### 3.2.4 Pattern Variables

We will now discuss about the semantic expressiveness of a pattern as discussed in the section 1.4.4. Generally speaking, since a pattern is defined by the way of relation over its constituent elements, we need to define the variables representing the relation between the elements of a pattern. Hence firstly, two variables can be defined that would represent the consecutive elements in a pattern. The
logical relation between these two variables should be satisfied by the any two consecutive elements in the pattern. Suppose if these variables are 'back' and 'current' (where back = current - 1) and the logical relation is \( \text{back} < \text{current} \) then every current element which is part of the sequence would be strictly less than that of its back (previous) element. In our proposed grammar, we define \( \text{pprev} \) and \( \text{pcur} \) on the same lines as described above. We also define three additional variables that are described below:

\[
\text{PatternVars} ::= ("\text{before}"|"\text{pprev}"|"\text{pcur}"|"\text{after}"|"\text{all}"
\$\text{Varname} ("," ("\text{before}"|"\text{pprev}"|"\text{pcur}"|"\text{after}"|"\text{all}")\$\text{VarName})*
\]

Each \( \text{PatternDefClause} \) represents a subsequence within a pattern (when multiple \( \text{PatternDefClause} \) are used for defining a pattern). In our proposed extension we have \( \text{PatternVars} \) for expressing the predicates over a pattern. We propose five variables whose role and semantic can be expressed in the following way:-

- **before**: This variable represents the element in the underlying physical sequence just before the start of the sub-pattern.
- **pprev**: Running variable representing the previous item position in the sequence.
- **pcur**: Running variable representing the current item in the sequence.
- **all**: Denotes all of the elements of matched sub-pattern.
- **after**: This variable represents the element just after the end of the sequence.

Following points are to be observed for these variables:

1. **before** denotes the element just before the start of the subsequence. Clearly it is outside the boundary of the pattern to be matched. Same can be applied to **after** variable, **after** represents the element past the sub-sequence. So, it is also outside the sub-sequence.

2. In above explanation, *Running variable* means these variables iterate through whole pattern and logical expression represented is evaluated for every element.

3. Further we will evaluate two potential possibilities of the **pprev** variable in the section 3.2.4. In first approach, **pprev** denotes the previous element in the underlying tuple stream (and this element could be part of another subsequence too). In another approach, **pprev** would strictly denote the element only inside the pattern to be matched.

4. In a sequence when **pcur** points to it’s first element, then **pprev** points to the element which is physically one element before it. If there is no previous subsequence then the previous element is empty sequence. So this comparison always results in false condition and **pcur** moves to the next i.e. second element of the sequence, if it exists.

5. **pcur** denotes the element that is part of the pattern. So **pcur** denotes the element that is inside the pattern.
6. all denotes every element of the required pattern. So any condition that is applicable on every element of the pattern can be specified on all.

7. In many of the usecases pcur and all can be used interchangeably.

8. Here it is evident that our proposed PatternVars is quite different to that of nine bound variables for a window in [1]. The variables defined in the start clause for the window in [1] are initially bounded when the predicate is satisfied and the variables defined in the end clause are also bounded when end predicates are satisfied or stream ends. These variables never change over this particular window. Due to this fact there is a less control over the contents of a window except start and end conditions in [1].

More elaboration on PatternVars

In this section we will evaluate the different semantic approaches for the pprev. In the end we will also evaluate both of the approaches.

Previous variable strictly inside the pattern: Suppose we have the pattern as $A^+B^+$ where $A$ denotes the increment and $B$ denotes the decrement. We have the following sequence:

\[
\{10, 12, 24, 9, 4\}
\]

The query for this pattern can be given in the following way:

\[
\text{pattern } \$t \text{ tumbling maximal in } \$seq \\
\$p \text{ as element()}+ \text{ pprev } \$a, \text{ pcur } \$b, \text{ all } \$u \text{ when } \$a < \$b \\
\$q \text{ as element()}+ \text{ pprev } \$c, \text{ pcur } \$d, \text{ all } \$v \text{ when } \$c > \$d
\]

Here $\$p$ and $\$q$ represent the pattern variables $A$ and $B$ respectively. We can represent the variable bindings for $\$p$ as in Table 3.1. In this way we can see that the comparison between $\$a$ and $\$b$ always results in false and hence the subsequence cannot be bound.

<table>
<thead>
<tr>
<th>pprev($$a$)</th>
<th>pcur($$b$)</th>
<th>all($$u$)</th>
<th>ExprSingle</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>10</td>
<td>${10}$</td>
<td>false</td>
</tr>
<tr>
<td>{}</td>
<td>12</td>
<td>${12}$</td>
<td>false</td>
</tr>
<tr>
<td>{}</td>
<td>24</td>
<td>${24}$</td>
<td>false</td>
</tr>
<tr>
<td>{}</td>
<td>9</td>
<td>${9}$</td>
<td>false</td>
</tr>
<tr>
<td>{}</td>
<td>4</td>
<td>${4}$</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 3.1: Binding for $\$p$ in $A^+B^+$ for sequence 3.1

Taking another example from the string matching patterns, suppose we want to match the pattern $A^+B^+$ with the following element stream containing elements $a_1, a_2, a_3, \ldots$ and $b_1, b_2, b_3, \ldots$ in which $a_1, a_2, a_3, \ldots$ represent the value ‘$A$’ and $b_1, b_2, b_3, \ldots$ represent the value ‘$B$’. Suppose the sequence of elements for this case is as:-

\[
\{a_1, a_2, b_1, b_2\}
\]

The query for this pattern can be given in the following way:
Table 3.2: Binding of $p$ in $A^+B^+$ for sequence 3.2

<table>
<thead>
<tr>
<th>pprev($a$)</th>
<th>pcur($b$)</th>
<th>all($u$)</th>
<th>ExprSingle</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>$a_1$</td>
<td>${a_1}$</td>
<td>true</td>
</tr>
<tr>
<td>{}</td>
<td>$a_2$</td>
<td>${a_1,a_2}$</td>
<td>true</td>
</tr>
<tr>
<td>{}</td>
<td>$b_1$</td>
<td>${a_1,a_2,b_1}$</td>
<td>false</td>
</tr>
</tbody>
</table>

Pattern $t$ tumbling maximal in $seq$
$p$ as element() pprev $a$, pcur $b$, all $u$ when $u/val$ eq ‘‘$A’’
$q$ as element() pprev $c$, pcur $d$, all $v$ when $v/val$ eq ‘‘$B’’

Table 3.3: Binding of $q$ in $A^+B^+$ for sequence 3.2

<table>
<thead>
<tr>
<th>pprev($c$)</th>
<th>pcur($c$)</th>
<th>all($v$)</th>
<th>ExprSingle</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>$b_1$</td>
<td>${b_1}$</td>
<td>true</td>
</tr>
<tr>
<td>$b_1$</td>
<td>$b_2$</td>
<td>${b_1,b_2}$</td>
<td>true</td>
</tr>
<tr>
<td>$b_2$</td>
<td>{}</td>
<td>${b_1,b_2}$</td>
<td>false</td>
</tr>
</tbody>
</table>

Binding of $p$ is represented as in Table 3.2, in the same way binding of $q$ is shown in Table 3.3. In this way, the whole pattern, binding of $t$ would be $a_1,a_2,b_1,b_2$.

**Previous in tuple stream**: This means that we take previous element from the underlying physical stream. Taking the same sequence from 3.1 and pattern as $A^+B^+$, the variable bindings for $p$ can be represented as in the Table 3.4 and binding for $q$ can be represented as in the Table 3.5. Summarily, the whole pattern comprises of $p$ and $q$ and can be represented as $\{12,24,9,4\}$.

Table 3.4: Binding for $p$ after taking previous for $A^+B^+$ for sequence 3.1

<table>
<thead>
<tr>
<th>pprev($a$)</th>
<th>pcur($b$)</th>
<th>all($u$)</th>
<th>ExprSingle</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>10</td>
<td>${10}$</td>
<td>false</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>${12}$</td>
<td>true</td>
</tr>
<tr>
<td>12</td>
<td>24</td>
<td>${12,24}$</td>
<td>true</td>
</tr>
<tr>
<td>24</td>
<td>9</td>
<td>${12,24,9}$</td>
<td>false</td>
</tr>
</tbody>
</table>

Now we take another example of the pattern $A^+$ and representing all the variables in the sequence. Here $A$ represents the increasing value in the sequence. Let’s take the sequence of elements as:

$\{10,15,20,25,30,5\}$

The related XQuery can given as:
3.2.5 Alternation

Suppose we need to express the pattern defined by regular expression, $AB|C$ using the so far proposed syntax. The PatternDefClause is unable to specify it alone. But we can also represent the alternation based regular expressions. For this, we extend the grammar proposed in section 3.2 in the following way:

PatternClause := ‘pattern’ "$" VarName (WindowTypeClause)
                (SelectionClause)? ‘in’ ExprSingle Pattern
Pattern := Term | ‘(‘Pattern’)’ | ‘or’ ‘(‘Term’)’
Term := Primary | Term Primary
Primary := PatternDefClause | ‘(‘Pattern’)’

Now the pattern defined by regular expression $AB|C$ can be expressed in the following manner:

$pattern \$t tumbling maximal in $seq
\$p as element() + pprev $b, pcur $c when $b lt $c
return $p$

The step by step binding of variables can be seen in the Table 3.6. The sequence represented by $\$p$ would be 15, 20, 25, 30. We can again see the fact though the first element {10} should have been the part of the sequence but due to comparison of {10} and empty sequence it was not taken in the subsequence.
3.2 Overview of the Proposed Extension

3.2.6 Specifying Direct Regular Expressions

We represent the pattern using regular expressions. In the extension proposed in section 3.2, regular expression is needed to be translated to define the PatternDefClause. In our proposal, there is flexibility to express the patterns by the way as we discussed in the section 3.2 or directly using regular expressions as we describe below. The slight change in grammar of 3.2 can be presented as:

\[
\text{PatternClause} := \text{''pattern'' (WindowTypeClause) (SelectionClause)?} \text{''in'' ExprSingle RegExpClause \text{''when''} (PatternDefClause)*}
\]

\[
\text{RegExpClause} := \text{Term | ''(''' RegExpClause'''' or ''('''Term''''))''}
\]

\[
\text{Term} := \text{Primary | Term Primary}
\]

\[
\text{Primary} := \text{VarRef | RegExpClause}
\]

For example, if we take the case of a simple pattern $AB$, we can represent now this pattern query as:

\[
\text{pattern } t \text{ tumbling maximal in } seq
\]
\[
(a \ b)
\]
\[
\text{when}
\]
\[
(a \text{ as element}() \text{ pcur } q1 \text{ when } q1/name eq 'A'
\]
\[
(b \text{ as element}() \text{ pcur } q2 \text{ when } q2/name eq 'B'
\]
\[
\text{return } \{a, b\}
\]

The equivalent query for above would also be:

\[
\text{pattern } t \text{ tumbling maximal in } seq
\]
\[
(a \ b)
\]
\[
\text{or}
\]
\[
(c)
\]

\[
\text{when}
\]
\[
(a \text{ as element}() \text{ pcur } q1 \text{ when } q1/name eq 'A'
\]
\[
(b \text{ as element}() \text{ pcur } q2 \text{ when } q2/name eq 'B'
\]
\[
(c \text{ as element}() \text{ pcur } q3 \text{ when } q3/name eq 'C'
\]
\[
\text{return } \{a, b, c\}
\]

Taking another example for the pattern $(AB)C$:

\[
\text{pattern } t \text{ tumbling maximal in } seq
\]
\[
(a \ b)\text{or}(c)
\]
\[
\text{when}
\]
\[
(a \text{ as element}() \text{ pcur } q1 \text{ when } q1/name eq 'A'
\]
\[
(b \text{ as element}() \text{ pcur } q2 \text{ when } q2/name eq 'B'
\]
\[
(c \text{ as element}() \text{ pcur } q3 \text{ when } q3/name eq 'C'
\]
\[
\text{return } \{a, b, c\}
\]

In this way, user has the choice whether to express the pattern query in the above manner or not.

**OpenIssues**

Suppose we have the pattern corresponding to the regular expression $A^*B^* | B^+A^+$, but as shown above the direct regular expression in XQuery can not be given.
36 Extending XQuery with Pattern Matching Facility

as ($a \, $b)$ or ($b \, $a)$ as the quantifier is different in $A^*$ and $A^+$. Either we have to specify the direct regular expression as ($a \, $b)$ or ($d \, $c)$ where $d$ would be same as $b$ for $B$, having just quantifier different and predicates same. This would be correct but would result in unnecessary redundancy. On the other hand, expressibility also gets affected. So there is a trade off between the expressibility and redundancy.

Also for the regular expression like $A^+ B^+ | B^+ C^+$ the direct regular expression could be specified as ($a \, $b)$ or ($b \, $c)$, but in the output patterns $b$ could be different for both of the constituent expressions making the issue of variable visibility. To express clearly, the direct regular expression should be ($a \, $b)$ or ($p \, $c)$, here we are using $b$ and $p$ for $B^+$.

3.3 An Explanatory Example

We also further present another example to explain the semantics involved in WindowTypeClause, SelectionClause. In this example, we present the all the pattern matches related to different combinations of WindowTypeClause and SelectionClause. Suppose we need to match the pattern $A^* B^+ | B^+ C^+$. For simplicity lets assume that the elements of the XML are represented in the form of a table as shown in Table 3.7. Figure 3.1 shows the patterns generated using the combination of SelectionClause and WindowTypeClause. The underline in figure 3.1 shows the difference horizontally i.e. between the WindowTypeClause’s and the color difference shows the difference vertically i.e. between the SelectionClauses.

### Table 3.7: Sample XML Data

<table>
<thead>
<tr>
<th>SeqNumber</th>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>a3</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>b1</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>b2</td>
<td>B</td>
</tr>
<tr>
<td>6</td>
<td>b3</td>
<td>B</td>
</tr>
<tr>
<td>7</td>
<td>c1</td>
<td>C</td>
</tr>
<tr>
<td>8</td>
<td>c2</td>
<td>C</td>
</tr>
<tr>
<td>9</td>
<td>a4</td>
<td>A</td>
</tr>
<tr>
<td>10</td>
<td>a5</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>b4</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>b5</td>
<td>B</td>
</tr>
<tr>
<td>13</td>
<td>c3</td>
<td>C</td>
</tr>
<tr>
<td>Match Mode</td>
<td>Tumbling</td>
<td>Sliding</td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td>Maximal</td>
<td>{a_1, a_2, a_3, b_1, b_2, b_3}, {a_4, a_5, b_4, b_5}</td>
<td>{a_1, a_2, a_3, b_1, b_2, b_3}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{a_2, a_3, b_1, b_2, b_3}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{a_3, b_1, b_2, b_3}, {b_1, b_2, b_3, c_1, c_2}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{b_2, b_3, c_1, c_2}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{b_3, c_1, c_2}, {a_4, a_5, b_4, b_5}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{a_5, b_4, b_5}, {b_4, b_5, c_3}, {b_5, c_3}</td>
</tr>
<tr>
<td>Incremental</td>
<td>{a_1, a_2, a_3, b_1}</td>
<td>{a_1, a_2, a_3, b_1}</td>
</tr>
<tr>
<td></td>
<td>{a_1, a_2, a_3, b_1, b_2}</td>
<td>{a_1, a_2, a_3, b_1, b_2}</td>
</tr>
<tr>
<td></td>
<td>{a_1, a_2, a_3, b_1, b_2, b_3}</td>
<td>{a_1, a_2, a_3, b_1, b_2, b_3}</td>
</tr>
<tr>
<td></td>
<td>{a_4, a_5, b_4}, {a_4, a_5, b_4, b_5}</td>
<td>{a_1, a_2, a_3, b_1}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{a_4, a_5, b_4}, {a_4, a_5, b_4, b_5}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{b_4, b_5, c_3}</td>
</tr>
<tr>
<td>AllMatch</td>
<td>Same as Incremental</td>
<td>{a_1, a_2, a_3, b_1}, {a_1, a_2, a_3, b_1, b_2, b_3}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{a_2, a_3, b_1, b_2}, {a_2, a_3, b_1, b_2, b_3}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{a_3, b_1}, {a_3, b_1, b_2}, {a_3, b_1, b_2, b_3}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{b_1, b_2, b_3, c_1}, {b_1, b_2, b_3, c_1, c_2}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{b_2, b_3, c_1}, {b_2, b_3, c_1, c_2}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{b_3, c_1, c_2}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{a_4, a_5, b_4}, {a_4, a_5, b_4, b_5}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{a_5, b_4}, {a_5, b_4, b_5}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{b_4, b_5, c_3}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{b_5, c_3}</td>
</tr>
</tbody>
</table>

Figure 3.1: Pattern Matching Results for $A^+B^+|B^+C^+$
3.4 Impacts of Pattern Clause On The Tuple Stream

A pattern clause generates zero or more patterns, each of which is composed of the bounded variables of the PatternDefClause clauses. Primarily we can have two cases, one in which pattern clause is the initial clause in the FLWOR expression and the other in which pattern clause is the intermediate clause in the FLWOR expression. If the pattern clause is the initial clause in the FLWOR expression, the bound variables that describe each subsequence inside the pattern is part of the output tuple. These tuples form the initial tuple stream that serves as input to the next clause of the FLWOR expression. The cardinality of the tuple stream is equal to the number of pattern matches in the binding sequence and each tuple contains the bound variables resulting from the PatternDefClause.

In the following example we will illustrate a pattern clause that is the initial clause in the FLWOR expression. Suppose we want to match the pattern \( AB^*C \).

Let’s consider the sample element sequence as, \((A, C, A, B, B, B, B, C)\).

The pattern clause for the above match would be:

```
pattern $p tumbling maximal in $seq
  $a as element() pcur $q1 when $q1/name eq 'A'
  $b as element()* pcur $q2 when $q2/name eq 'B'
  $c as element() pcur $q3 when $q3/name eq 'C'
```

Here the output tuple stream can be represented as:

\((a = \{A\}, b = \{\}, c = \{C\})\)
\((a = \{A\}, b = \{B, B, B, B\}, c = \{C\})\)

As we can see there is tuple stream containing two tuples. In one of the tuple, \(b\) is bound to the empty sequence as it was required due to the \(B^*\) in the pattern to be matched. This can very well occur in this extension in contrast to the Window based XQuery [1].

3.5 State Diagram

Pattern matching semantics can be expressed as the state diagram in the Figure 3.2. This diagram shows various states while pattern matching. These states can be expressed as:-

**Closed State**: This is the state when a subsequence match is not detected. End of stream causes transition in end of state.

**Active Matching**: A pattern is composed of one or more variables. If any of the variables match, it goes in active match state. In this state it is expecting more matches for the pattern variables. Transition from Active state to Closed state can occur if the Effective Boolean value (EBV) is evaluated to be false.

**Pattern Match**: In this state a successful pattern match is established and with subsequent EBV false or end of stream it results in end state.
3.6 Patten Extension versus Window Extension

In this section we will compare the design of our proposed pattern extension and the current model of the XQuery[1].

As we earlier discussed, there are start and end expressions in the current model of XQuery for defining starting conditions and end conditions of a window respectively. There are four window variables in the current model for each of the expression `prevItem, curItem, nextItem, position` and one binding variable representing the window. Predicates over the starting and end condition of a window can be semantically described using these variables except the binding variable representing the window. Comparing to our pattern extension, we have five `PatternVars` namely `before, pprev, pcurl, all` and `after` and one binding variable representing the sub-pattern for expressing any sub-pattern. Though there is no direct comparison but semantically, `prevItem` in current model is comparable to the `before` in proposed extension, similarly `curItem` specifies the first element of the `all` variable in the pattern clause.

Contrary to current model, `PatternVars` in pattern extension are running variables, i.e. the predicates over these variables are evaluated over every element of the sub-pattern while in former, these variables just define the starting and end condition over a window. That’s why as we seen earlier, there are limited patterns that can be defined using the current model of XQuery and hence there was the need of pattern matching extension.

Also in current model of XQuery, we have earlier shown cases in section 2.2.2 that we need to perform the join for any predicate over the two windows. This forces user essentially to think about the join conditions and joins have higher cost. Alternatively, in pattern matching extension, this is straight forward to access the binding variable of the previous sub-pattern and writing the predicates over it for the required sub-pattern.

In case of star operator based queries in the current model as described in the section 2.2.4, we had transformed it using the ‘+’ operator and then took the
union of all the window formed by transformations. In our proposed extension, this can be handled very easily as occurrence indicator of elements can be specified in the PatternDefClause itself. Taking an example below to write the query for the pattern $A^*B^*C$:

```
pattern $p$ sliding maximal in $seq/stream/event$
  $a$ as element()* pcur $a1$ when $a1/person eq "A"
  $b$ as element()* pcur $b1$ when $b1/person eq "B"
  $c$ as element() pcur $c1$ when $c1/person eq "C"
return <pattern>{$p}</pattern>
```

In the above example, we can see that just element()* was used to specify for the $A^*$ and $B^*$. In fact, we will use this same pattern for the benchmarking against the current model in the section 6.2.
Chapter 4

Pattern Matching Extension In Action

In this chapter, we will provide few examples based on our proposal of pattern matching extension in the last chapter. We have grouped our queries based on real use cases and simple syntactical pattern matching in the sequences.

4.1 Syntactic Examples

4.1.1 Simple Patterns

In this section, we will formulate the simple pattern matching queries in the proposed extension. These simple patterns are on the same line as we introduced in the section 1.2.2. In the following patterns, the variable in a regular expression denotes the occurrence of that element in the sequence of elements. These patterns are based on elements that are non-correlated.

\[AB\]

Suppose we need to match the person occurrences of \( A \) and \( B \) and it must occur within 1 hour time window. This can be formulated as:

\[
\text{pattern } \$t \text{ tumbling maximal in } \$seq \\
\$a \text{ as element()} \ pprev \$p1, pcur \$q1 \text{ when } \$q1/name eq "A" \\
\$b \text{ as element()} \ pcur \$q2 \text{ when } \$q2/name eq "B" \\
\text{where } |\$b[last()] /@time - \$a[first()]/@time| \lt 1 \text{ hour} \\
\text{return } \{\$a, \$b\}
\]

By the current syntax \( a \) and \( b \) are matched to their respective boundaries.

\[A^*B^+C\]

Here \( A \) can have zero or more occurrences.

\[
\text{pattern } \$t \text{ tumbling maximal in } \$seq \\
\$a \text{ as element()}* \ pprev \$p1, pcur \$q1 \text{ when } \$q1/name eq "A"
\]
$b$ as element()+ pcur $q2$ when $q2$/name eq "B"
$c$ as element() pcur $q3$ when $q3$/name eq "C"
return {$a, b, c}$

\((AB)\mid C\)

pattern $t$ tumbling maximal in $seq$
($a$ as element() pcur $q1$ when $q1$/name eq "A"
$b$ as element() pcur $q2$ when $q2$/name eq "B")
or
$c$ as element() pcur $q3$ when $q3$/name eq "C"
return {$a, b, c}$

\(A^+\mid !B\)

pattern $t$ tumbling maximal in $seq$
$a$ as element()+ pprev $p1$, pcur $q1$ when $p1$/price lt $q1$/price
$b$ as element() pprev $p2$, pcur $q2$ when $p2$/price gt $q2$/price and $q2$/name eq "B" and $q2$/price gt $a[/last()]$/price + 5
return {$a, b}$

As we can see that $a$ is in scope for the $b$ but not pprev $p1$ or pcur $q1$.

\(A^+\mid B^+\)

Suppose A is defined as the event in which A's value increases and B's value decreases. So here we have to match the occurrence of event A as well as the movement of its value.

pattern $t$ tumbling maximal in $seq$
$a$ as element()+ pprev $p1$, pcur $q1$ when $p1$/price lt $q1$/price and $q1$/name eq "A"
$b$ as element() pprev $p2$, pcur $q2$ when $p2$/price gt $q2$/price and $q2$/name eq "B" and $q2$/price gt $a[/last()]$/price + 5
return {$a, b}$

In this section, we introduce the patterns in which variables in the regular expression imply predicates for the selection of elements from the sequence. Since these patterns are based on correlated elements, these include the conditional expression comprising PatternVars.

Suppose A is defined as the event in which A's value increases and B's value decreases. B's value should also be greater than by 5 of last A's value in the pattern.

pattern $t$ tumbling maximal in $seq$
$a$ as element()+ pprev $p1$, pcur $q1$ when $p1$/price lt $q1$/price and $q1$/name eq "A"
$b$ as element() pprev $p2$, pcur $q2$ when $p2$/price gt $q2$/price and $q2$/name eq "B" and $q2$/price gt $a[/last()]$/price + 5
return {$a, b}$
4.2 Usecase Examples

4.2.1 Stock Prices - ‘Jump’ Pattern

We need to find a pattern in which stock price increase from a price level to five times of previous price level. Here in this pattern A represents the initial price level and B represents the price increase and C represents the price which is five times of the initial price.

The corresponding query StockTimesTumbling.xq and StockTimesSliding.xq can be found in the MXQuery repository at PatternTests/Queries/Pattern location.

pattern $t tumbling maximal in $seq
$a as element() when fn:true()
$b as element()* pprev $p, pcur $q, all $r, after $s when $p/val lt $q/val and $q/val lt (5*($a/val))
$c as element() pprev $t, pcur $u, all $v when $t/val lt $u/val and $u/val ge (5*($a/val))

Lets evaluate the above XQuery for the various input element sequences:-

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Output Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>{10, 50}</td>
<td>(1) $a = {10}, $c = {50}</td>
</tr>
<tr>
<td>{10, 14, 50}</td>
<td>(1) $a = {10}, $b = {14}, $c = {50}</td>
</tr>
<tr>
<td>{10, 14, 15, 50}</td>
<td>(1) $a = {10}, $b = {14, 15}, $c = {50}</td>
</tr>
</tbody>
</table>
| {10, 14, 15, 50, 70}   | (1) $a = {10}, $b = {14, 15}, $c = {50}  
                        (2) $a = {14}, $b = {15, 50}, $c = {70} (in case of sliding) |

4.2.2 Stock Prices - ‘Tick’ Pattern

Suppose we need to find a pattern in which prices of a stock first falls, followed by a rise in the price that eventually goes higher than the price was when the fall began. This can be shown by $AB^*CD$ where PatternDefClause’s corresponding to $A, B, C, D$ are as $a, b, c, d$ in the XQuery and they are as follows:-

- $A$ matches for any item in the sequence.
- $B$ denotes the falling down of the prices.
- $C$ denotes the price rise uptill it is less than or equal to that of $A$’s.
- $D$ denotes the further price rise higher than that of $A$.

The XML Data and XQuery that we consider is as follows:-

```xml
<stock>
  <event> <date>2008-01-01</date> <price>99</price> </event>
```
<stock>
<event> <date>2008-01-02</date> <price>100</price> </event>
<event> <date>2008-01-03</date> <price>80</price> </event>
<event> <date>2008-01-04</date> <price>70</price> </event>
<event> <date>2008-01-05</date> <price>85</price> </event>
<event> <date>2008-01-06</date> <price>88</price> </event>
<event> <date>2008-01-07</date> <price>102</price> </event>
<event> <date>2008-01-08</date> <price>106</price> </event>
<event> <date>2008-01-09</date> <price>50</price> </event>
</stock>

pattern $p$ tumbling maximal in $seq$
$a$ as element() when fn:true()
$b$ as element()+ pprev $p1$, pcur $q1$ when $p1/price$ gt $q1/price$
and $q1/price$ lt $a/price$
$c$ as element()* pprev $p2$, pcur $q2$ when $q2/price$ gt $p2/price$
and $q2/price$ le $a/price$
$d$ as element() pprev $p3$, pcur $q3$ when $q3/price$ gt $p3/price$
and $q3/price$ gt $a/price$
return
{ $a$ $b$ $c$ $d$ }

The output of the XQuery would be:-

<pattern>
<event> <date>2008-01-02</date> <price>100</price> </event>
<event> <date>2008-01-03</date> <price>80</price> </event>
<event> <date>2008-01-04</date> <price>70</price> </event>
<event> <date>2008-01-05</date> <price>85</price> </event>
<event> <date>2008-01-06</date> <price>88</price> </event>
<event> <date>2008-01-07</date> <price>102</price> </event>
</pattern>

Now we will show the patterns corresponding to the variables:-

- **Pattern $A$**: As it is shown in the XQuery, $A$ matches for any element in the binding sequence. The pattern corresponding to $A$ would be:-

  <event> <date>2008-01-02</date> <price>100</price> </event>

- **Pattern $B^+$**: In construction of the subsequence for $B$, firstly element corresponding to 80(pcur) was compared to 100(pprev) then 70(changed pcur) was compared to 80(changed pprev) which passes the condition. Subsequently, 85 was compared to the 70 but it does not pass the condition. Here we can also observe that matching for $b$ starts right after where $A$ ends. $b$ was bound to as following in the tuple stream:-

  $b$ = {<event> <date>2008-01-03</date> <price>80</price> </event>,
  <event> <date>2008-01-04</date> <price>70</price> </event>
4.2 Usecase Examples

- **Pattern C**: In construction of the subsequence $pprev$ and $pcur$ were compared in the same manner as explained above. $c$ is bound to as following in the tuple stream:

  $c = \{<\text{event}> <\text{date}>2008-01-05</\text{date}> <\text{price}>85</\text{price}> </\text{event}>,
  <\text{event}> <\text{date}>2008-01-06</\text{date}> <\text{price}>88</\text{price}> </\text{event}>\}$

- **Pattern D**: $D$ is the non-grouping variable and the $pprev$ value for it is 88 and $pcur$ is 102. This condition passes and hence $d$ is bound to as following in the tuple stream:

  $d = \{<\text{event}> <\text{date}>2008-01-07</\text{date}> <\text{price}>102</\text{price}> </\text{event}>\}$

For the sliding, maximal parameters the output would be:

<pre>
<pattern>
  <event> <date>2008-01-02</date> <price>100</price> </event>
  <event> <date>2008-01-03</date> <price>80</price> </event>
  <event> <date>2008-01-04</date> <price>70</price> </event>
  <event> <date>2008-01-05</date> <price>85</price> </event>
  <event> <date>2008-01-06</date> <price>88</price> </event>
  <event> <date>2008-01-07</date> <price>102</price> </event>
</pattern>
</pre>

<pre>
<pattern>
  <event> <date>2008-01-03</date> <price>80</price> </event>
  <event> <date>2008-01-04</date> <price>70</price> </event>
  <event> <date>2008-01-05</date> <price>85</price> </event>
</pattern>
</pre>

Same arguments can be presented for this case as were in the previous case.

Let’s consider the sample XML below that shows the price for a stock of the company.

<pre>
<stock>
  <event> <date>2008-01-01</date> <price>101</price> </event>
  <event> <date>2008-01-02</date> <price>100</price> </event>
  <event> <date>2008-01-03</date> <price>104</price> </event>
  <event> <date>2008-01-04</date> <price>106</price> </event>
  <event> <date>2008-01-05</date> <price>99</price> </event>
  <event> <date>2008-01-06</date> <price>90</price> </event>
  <event> <date>2008-01-07</date> <price>93</price> </event>
  <event> <date>2008-01-08</date> <price>95</price> </event>
</stock>
</pre>

Suppose we need to find the increasing trend of the price for the above stock pricings. The PatternDefClause would be written as:

```xml
pattern $q$ tumbling maximal match in $seq/stock/event$
  $a$ as element()+ $pprev$ $p$, $pcur$ $c$ when $p/price < c/price$
return
<pattern>
  $\{a\}$
</pattern>
```
For our sample query, having tumbling and maximal match, it generates a tuple stream consisting of two tuples, each representing a pattern and containing $a$ bound variable from PatternDefClause. The return clause is evaluated for each of these tuples, generating the following query result:

```xml
<pattern>
  <event> <date>2008-01-02</date> <price>100</price> </event>
  <event> <date>2008-01-03</date> <price>104</price> </event>
  <event> <date>2008-01-04</date> <price>106</price> </event>
</pattern>
<pattern>
  <event> <date>2008-01-06</date> <price>90</price> </event>
  <event> <date>2008-01-07</date> <price>93</price> </event>
  <event> <date>2008-01-08</date> <price>95</price> </event>
</pattern>
```

The testcases shown in this example can be found in the repository by the name of StockPricesTumbling.xq and StockPricesSliding.xq at the location PatternTests/Queries/Pattern.

### 4.2.3 HTML Document Restructuring

We present here the UseCase related to convert a document structure into the user defined structure. Suppose the input document is:

```xml
<body>
  <h2>heading1</h2>
  <p>para1</p>
  <p>para2</p>
  <h2>heading2</h2>
  <p>para3</p>
  <p>para4</p>
  <p>para5</p>
</body>
```

The Output expected for this input is:

```xml
<chapter>
  <section title="heading1">
    <para>para1</para>
    <para>para2</para>
  </section>
  <section title="heading2">
    <para>para3</para>
    <para>para4</para>
    <para>para5</para>
  </section>
</chapter>
```

Here it is needed to match h2 tags and contain all the p tags in the section tags. Thus it can be taken as a pattern to match h2 and p tags. The XQuery for this would be as:-
4.2 Usecase Examples

4.2.4 Adjacent Bullet Point Detection

In this case the problem is to identify a sequence of bullet elements (among a sequence) and wrap them. The input document can be given as:

```xml
<p/>
<q/>
<bullet>one</bullet>
<bullet>two</bullet>
<x/>
<y/>
```

The output should be:

```xml
<p/>
<q/>
<list>
<bullet>one</bullet>
<bullet>two</bullet>
</list>
<x/>
<y/>
```

The corresponding XQuery using pattern extension can be given as:

```xml
pattern $t tumbling maximal in $seq/doc/*
$a as element() pcur $q1 when string(node-name($q1)) ne 'bullet'
$b as element() pcur $q2 when string(node-name($q2)) eq 'bullet'
return
{a}
</list>
```

4.2.5 Outlier Detection

Sensors are periodically reporting temperature values. Outlier is detected and reported, if current value of the sensor is two times higher (lower) than a previous
one. Each event has attributes: ID, timestamp, sensor ID and temperature value. The stream input for this use case can be shown as:-

<stream>
<event id="1" time="1" sensor="1" temp='10'/>
<event id="2" time="2" sensor="1" temp='9'/>
<event id="3" time="2" sensor="2" temp='10'/>
<event id="4" time="3" sensor="2" temp='35'/>
<event id="5" time="4" sensor="2" temp='25'/>
<event id="6" time="5" sensor="2" temp='22'/>
<event id="7" time="6" sensor="2" temp='5'/>
<event id="8" time="7" sensor="2" temp='7'/>
<event id="9" time="8" sensor="2" temp='10'/>
<event id="10" time="9" sensor="2" temp='11'/>
<event id="11" time="10" sensor="1" temp='20'/>
<event id="12" time="11" sensor="1" temp='25'/>
</stream>

The XQuery using pattern extension can be written as:-

```
pattern $t tumbling incremental in $seq/stream/event
$a as element() pprev $p, pcur $q when ($q/@temp gt ($p/@temp *2))
or ($q/@temp *2 lt ($p/@temp))
return
<Outliers>
{$a}
</Outliers>
```

Here we have used the incremental match mode as the data from the sensor nodes is streaming. It is needed to detect outliers with every incoming event. The output for the above XQuery can be given as:-

```
<Outliers><event id="4" time="3" sensor="2" temp='35'/><Outliers>
<Outliers><event id="7" time="6" sensor="2" temp='5'/></Outliers>
<Outliers><event id="11" time="10" sensor="1" temp='20'/></Outliers>
```

Chapter 5

Implementation

We extended the Micro XQuery Engine (MXQuery) and implemented our proposed pattern matching extension on top of it. In this chapter we will first describe the general architecture of MXQuery engine relevant for our details of implementation, followed by the implementation overview of pattern extension.

5.1 MXQuery

5.1.1 Overview

The MXQuery engine was developed as a lightweight XQuery engine primary for the usage on small devices like mobile phones etc. Due to its specialized nature of development for small devices, it becomes important for those devices as in today’s world many more applications use SOAP exchanges and XML messages for other applications.

The basic architecture of MXQuery engine is similar to the BEA XQuery engine[7] and it consists of pull based iterator model. This approach yields the advantages like low memory requirements, avoidance of materialization of intermediate results and lazy evaluation which is important for infinite streams and is already tested on the engine[6]. MXQuery engine is based on XMLPullAPI[11]. XMLPull is cursor based interface where events are directly available as properties from the parser and no object is created to represent an XML event whereas in BEA Engine each XML event is represented as new object.

XMLPull interface was originally designed to parse only XML files, so every sequence of event starts with START DOCUMENT and ends with END DOCUMENT event. For example, if there is a XML document like <a>2</a>, then a XMLPull parser would return the events: START DOCUMENT; START_TAG; INT; END_TAG; END DOCUMENT. In contrast to it, XQuery Data Model(XDM) which works on sequence of items where an item could also be a document. Hence it needs to be changed and several new events were introduced like START_SEQUENCE and END_SEQUENCE. So as we can see by representing the XML events as objects, BEA Engine comes with a high memory cost too.
5.1.2 **Iterator Model in MXQuery**

In MXQuery engine these new events are wrapped in as Tokens. Every Token represents event type, type annotation and the event identifier. During parsing an XQuery is translated in the iterator tree. Each iterator represents an operator of the XQuery library or basic operators. In MXQuery engine every iterator implements the Iterator interface which in turn extends the XDMIterator interface. The root iterator is directly returned to the user from which user can pull the result[12]. All iterators implement the `next()` method of XDMIterator interface, that returns the next XDM token.

Suppose we have the following simple XQuery:-

```
pattern $p tumbling maximal in (1,2)
$p as element() pprev $p1, pcur $c1 when $p1 lt $c1
return $p
```

The resulting iterator tree shown in Figure 5.1 shows the query transformation.

![Figure 5.1: Iterator Diagram](image-url)

In Figure 5.1 resulting iterator tree is shown for the above query. The parser translates the query into a top level `GFLWORIterator` which has two sub-iterators: `XMLContent` for `return` clause and `PatternNaiveIterator` for `pattern` clause. `XMLContent` has sub-iterators; two `TokenIterator` and one `VariableIterator`. `PatternNaiveIterator` represents the `pattern` clause and it operates over the sequence. Hence there is `SequenceIterator` having two `TokenIterator` as sub-iterators. The condition in `PatternDefClause` evaluates to true or false. Hence effective boolean value is calculated of this expression. `BooleanIterator` calculates this effective boolean value. Our expression in the `PatternDefClause` is Comparison Expression and thus yields a `CompareIterator`. This comparison
is performed upon two variables and \texttt{fn:data()} is implemented as DataValuesIterator. These DataValuesIterator have the VariableIterator as sub-iterators. The query user gets the top-level \texttt{GFLWORIterator} as the outcome of the parsing and the respective sub-iterators are invoked by the \texttt{next()} call. Here, when the \texttt{next} method of PatternNaiveIterator is called for the first time, it tries to bind the underlying window iterator from WindowFactory as we earlier mentioned this first time and initializes the VariableHolder of the pattern variable. If this is not successful, it straight away returns the Token END_SEQUENCE_TOKEN otherwise it binds the window iterator to the main pattern variable and returns START_SEQUENCE_TOKEN.

After calling the \texttt{next()} events from the \texttt{return} clause it resets the pattern iterator to start the new binding. The current position of the item in the underlying window remains updated and resumes from the next item. In further sections we will discuss more about the implementation details of the pattern extension clause.

### 5.1.3 Materialization in MXQuery

It is evident that a streaming engine needs materialization facility in order to serve different types of windows that we earlier mentioned in our proposal. MXQuery offers materialization facility by the way of \textit{Store}. All Stores save the materialized data in a list of Tokens. A Token is the materialization of one event. MXQuery contains Stores which save these lists which are optimized for fast, lazy materialization and fast read access[8]. MXQuery engine contains a WindowFactory with static methods to materialize iterators. The parameters of the methods are an iterator and some additional parameters to tune the process. WindowFactory in turn calls the WindowBuffer which is a factory for creating window iterators. Window Iterators are only created by a WindowBuffer or by a existing WindowIterator. All window iterators operate on window buffer and have a defined range on this buffer. In MXQuery WindowIterator is an abstract class containing the underlying WindowBuffer and other basic members. WindowItemIterator, WindowTokenIterator and WindowSequenceIterator extend this class. Window Iterators have in common that they provide a view on the original sequence. In our implementation, we create the window iterator over the underlying window buffer by giving the start and end positions, so it returns WindowSequenceIterator. At first call, when we pass the VariableIterator to the WindowBuffer then underlying window iterator(as we mentioned WindowSequenceIterator) is returned. In subsequent calls to this window iterator with start and end boundaries, the WindowSequenceIterator returns the window iterator for the same underlying window buffer.

### 5.1.4 Development Environment

The whole project was developed within the MXQuery project and the programming language used was Java. Also as MXQuery is built on Sun Wireless Toolkit 2.5, the Java packages were used accordingly. As MXQuery project has downstream MXQuery-Testing project for extensive testing using the various testcases, relevant testcases were also written in the MXQuery-Testing project.
for the implemented extension. The Pattern iterator classes mentioned here are in the ch.ethz.mxquery.iterators.pattern package.

5.2 Pattern Matching Extension In MXQuery

5.2.1 Parser

MXQuery parser was extended for the new pattern matching extension clause. We have added patternClause() function that parses the XQueries beginning with the keyword “pattern”. The iterator tree model that was generated is already described in the section 5.1.2.

5.2.2 Finite State Machine

As we earlier discussed in our language proposal, a pattern is semantically described in PatternDefClause. The PatternDefClause has also the quantifier field for the particular sub-pattern. This can be seen as equivalent to a variable in regular expression. Now these set of PatternDefClauses are needed to be evaluated corresponding to the stream of sequence of items.

We first parse the quantifier fields corresponding to all of PatternDefClause. We construct a Non-deterministic Finite State Machine in which each state corresponds to PatternDefClause. This FSM helps in matching the satisfiability of incoming data item corresponding to the current state of the machine. The FSM constructed has the edges. Edges simply show the transition possibility of a particular state. A state is linked to other states via edges. If any incoming item satisfies the predicates corresponding to the other states linked via edges to the current state then the machine makes the transition to that linked state.

Here it is to be noted that some states can have edges directly to the end state. For example, if the pattern to be matched as $A^+ B^* C^* D^*$ then $B, C, D$ have edges directly linked to the end state.

Let’s say the regular expression corresponding to the set of PatternDefClause is $A^* B^* C^* D^+$ then if current accepting state of FSM is $B$, then in the next stage, it can accept data items corresponding to both $C$ and $D$. If the incoming data item corresponds to the PatternDefClause as $D$, then FSM would enter in the end state.

5.2.3 Pattern Iterator

We extend the existing FFLWOR expression for the pattern matching extension. As the top level FFLWOR iterator composes of all the FFLWOR expression iterators, the pattern matching iterator was also added as a part of it. For this, existing parser was modified in order to parse the pattern matching extension and add it accordingly to the iterator tree. Our pattern matching extension implemented as pattern iterator and composed of the classes PatternNaiveIterator, PatternIterator and PatternBaseIterator.

PatternNaiveIterator: This class extends PatternIterator. PatternBaseIterator extends TokenBasedIterator and implements the basic iterator function next().
5.2 Pattern Matching Extension In MXQuery

of XDMIterator. The next function does the following:-

- It assigns the window to the binding variable of the pattern, if the input sequence ends or cannot return any more pattern match, this function returns END_SEQUENCE_TOKEN else it returns START_SEQUENCE_TOKEN.
- On the first call, it assigns the underlying window buffer that we earlier mentioned.

PatternIterator: In earlier section 3.2.4, we discussed about the PatternVars for a particular sub-pattern and this comprises of before, pprev, pcur, after and all. Also as we said in our specifications 3.2.4, pprev and pcur are running variables and conditional expression comprising them should be satisfied by every consecutive element in the required sub-sequence. In PatternIterator, these PatternVars are assigned the iterator for the item in the underlying sequence. This is used when ExprSingle boolean iterators are called for evaluating the condition in the when clause of every PatternDefClause.

PatternNaiveIterator: This is the main class that constructs the NFA mentioned in the section 5.2.2. Calling point of this class is the function assignWindow() which returns the main window for the binding variable of the pattern. If it cannot bind any window, it simply returns null and the PatternBaseIterator then returns the ENDSEQUENCE_TOKEN.

5.2.4 Pattern Variable Bindings

As we earlier described in section 3.2.4, the PatternVars are running variables i.e. the predicates should hold over the whole sequence. This works as follows:

- PatternVars are assigned the item over the sequence. This means, window iterator is set for the single item, over the same WindowBuffer as we mentioned that we use the same WindowBuffer.
- Above point is valid in case of before, pprev, pcur and after and for all, until the ExprSingle for PatternDefClause becomes true, all is assigned same item as of pcur.
- When ExprSingle becomes true for the first time, the current item position in the window iterator, lets call it initialPointer is set for all and at same time, all holds the same item as of pcur.
- Now pcur is advanced and for every pcur, pcur is appended in the all i.e. WindowIterator is assigned to the VariableHolder of all having boundaries as initialPointer and current item position of pcur.
- ExprSingle is computed, if ExprSingle is true above point is repeated.
- If ExprSingle becomes false or end of sequence occurs, all is set to the window iterator from the initialPointer up to the index of pcur at which ExprSingle was last true.
In the above procedure, in every iteration over the sequences of items, `destroyWindow()` is called over each of the `PatternVars` if the iterator corresponding to the `PatternVars` is not null. `before`, `pprev`, `pcur` and `after` are assigned the items each time in `assignVars()` of `PatternNaiveIterator` and all is assigned in `assignallVars()` of `PatternNaiveIterator`.

### 5.2.5 Matching Modes

Implementation of matching modes according to the `SelectionClause` in the XQuery is linked to the matching states described below.

Matching States

The machine makes the transition to the different running states according to the incoming items fed into the machine. The main purpose of defining running states was also to distinguish between the behavior of window types and selection modes and take the decision accordingly. We use `startPosition` as a pointer to the first item match of the machine and `currentPosition` is the the position of the current item fed into the machine. Our state machine has following possible running states:

- **NON-MATCH**: The machine goes into this running state whenever a item does not match according to the transition states in the edges. In this state `startPosition` is moved to that item, for which no match occurs. In broader sense, a pattern match needs to be re-started, if possible, from the beginning if this running state occurs.

- **PARTIAL-MATCH**: The machine is in this running state when it makes a partial match of the given pattern and needs to consume more input in order to determine the total match. So in this running state matching process is continued.

- **CONTINUOUS-MATCH**: This running state occurs when machine makes a total match of the pattern. The match would be from `startPosition` to the `currentPosition`. Also, the machine needs to consume more inputs in order to `extend` this match(such as match for `A*B*` and match for `B` can be extended further).

- **SLIDEPARTITION-MATCH**: This running state occurs whenever the machine cannot consume input according to the transition states of edges or end of underlying sequence occurs and the previous state of the machine was **CONTINUOUS-MATCH**. The name for this running state comes from the fact that in this running state, machine slides in the partition according to the `WindowTypeClause` defined in the XQuery. If the `WindowTypeClause` is **tumbling** then `startPosition` becomes equal to the `currentPosition` and if it is **sliding** then `startPosition` slides to next item of the previous pattern match.

`SelectionClause` plays role in deciding when pattern matches are produced i.e. `WindowIterator` is linked to the start position and current item position. We can classify it according to `SelectionClause` in following manner:
5.2 Pattern Matching Extension In MXQuery

- If SelectionClause is specified as maximal, the WindowIterator is only linked when the machine goes in running state as SLIDEPARTITION-MATCH.

- If SelectionClause is specified as incremental, WindowIterator is linked to all running states as CONTINUOUS-MATCH.

- For allmatch as SelectionClause, WindowIterator is linked to all running states as CONTINUOUS-MATCH and according to allmatch specifications as we earlier specified in section 3.2.1, machine is reset to NON-MATCH and startPosition is set to the next item of the last pattern match (if WindowTypeClause is sliding) or currentPosition is set to the currentPosition (if WindowTypeClause is tumbling).

5.2.6 No MatchBuffer

We earlier described that the WindowIterator and using the same WindowBuffer if the underlying iterator passed is WindowIterator for the materialization. Implementations based on matchbuffer simply store the matches so far generated[5] and generate the huge cost of storing, here we use the already implemented WindowIterator so we are able to save the storage cost of unnecessary materialization. In other sense, our virtual matchbuffer exists here in the form of WindowIterator.
Chapter 6

Experiments

In this chapter we will describe our benchmarking experiments.

6.1 Setup

Following are the important points about our set up:

- The machine for the experiment used was Intel Core2 Duo, 2.2 GHz running Windows 7, 32 bit version and total RAM of 2.0 GB.
- We increased size of data items for each measurement.

ch.ethz.mxquery.xdmio.XDMSerializer class serializes the XQuery data model(XDM) instance primarily using the entry function `eventsToXML(XDMIterator resIter)`. This function uses Java api `ByteArrayOutputStream` to put the data in the stream. For large set of data items, this results in bottleneck and large java heap space is needed. We replaced this `ByteArrayOutputStream` with a dummy stream for our experiments by which we effectively do not run into heap issues. By this, we are not just saving the serialized data in the buffer, all other computations remains the same.

6.2 Effectiveness Of The Extension

Here we compare the performance of our newly added extension versus current pattern matching features of XQuery. We performed benchmarking with two types of queries, one of the queries was having star operator selection and the other has plus operator selection. The objective was to analyze the performance difference for these two types of operators with the current model of XQuery and proposed pattern matching extension. Here, current model of XQuery is important because as we mentioned earlier, current model performs join even for simple queries while our proposed pattern matching extension does not.

Our first experiment is for the pattern $A^*B^*C$ and we measured the time difference with respect to current XQuery model and pattern matching extension clause. We used `sliding maximal` in the pattern matching clause and `sliding window` in the `FORSEQ` clause of current XQuery model. We keep the total
number of pattern matches constant so when the total number of items varies in the dataset we vary the length of the any single pattern matched. So number of pattern matches remain constant in this way.

For example, if we have total number of items as 2000, and our dataset is such that we have total 100 patterns of length 20. In the same way, for the next reading, if number of items is increased to 3000, then for 100 pattern matches (this was kept same) we will increase the pattern length from 20 to 30. Similarly, for 4000 data items, pattern length would be 40. The graph for this experiment is shown in the figure 6.1.

We can analyze that as we increase the number of data items, the total running time difference between the FORSEQ clause and pattern matching extension is getting huge. For total 200 data items, the running time for FORSEQ clause is 4 times more than that of pattern matching clause while for 800 data items, the running time for FORSEQ clause is 19 times more than that of pattern matching clause. This shows the expensive cost of joins that by four times increment of number of data items, total running time goes up by 19 times in comparison to pattern matching extension. In the figure 6.1 we have shown the measurements for maximum 800 number of times, we also took measurements for 2000 data items, there the difference in running time for FORSEQ clause went up to by 50 times.

The second experiment was for the pattern $A^+B^+C$, with the same data set and environment as described above. We analyzed the performance for sliding as well as tumbling in this case. Figure 6.2 shows the graph of tumbling, maximal

![Pattern A*B*C, sliding](image)

Figure 6.1: Pattern $A^*B^*C$ sliding
6.2 Effectiveness Of The Extension

in pattern matching clause and tumbling window in FORSEQ clause. Figure 6.3 shows the graph taking sliding, maximal in pattern matching clause and sliding window in FORSEQ clause.

For sliding case as shown in the figure 6.2, we can see that the running time difference between FORSEQ clause and proposed pattern clause is very high. For 2000 data items, the running time for FORSEQ clause is 39 times higher than that of pattern clause. We earlier mentioned that for the pattern $A^*B^*C$, the running time for same number of data items was 50 times higher than of pattern clause. This can be verified from the fact that star operators in general would be more costly for the random data as number of matches would be high. Also, figure 6.2 clearly shows the huge running time difference between pattern clause and FORSEQ clause.

As we mentioned earlier, figure 6.3 shows the measurements for tumbling case and can be well compared to that of sliding from figure 6.2 as number of data items remains the same for individual cases. The running time in the case of FORSEQ clause for 4000 items in sliding case is approximately 147 seconds while in tumbling it is only 4 seconds. This also goes in line with the fact that sliding windows are expensive and associated joins make overall matching more costly. The sliding version in this case on average takes 28 times more time to that of tumbling. Comparing the tumbling and sliding versions for pattern clause, the sliding one on average takes 10 times more time to that of tumbling.

We can conclude that sliding and tumbling version in pattern clause (as SelectionClause is MAXIMAL in our measurements) is far cheap to that of counterparts in current XQuery model.

![Figure 6.2: Pattern $A^+B^+C$ sliding](image)

Figure 6.2: Pattern $A^+B^+C$ sliding
6.3 Impact Analysis Of Main Elements

6.3.1 Window Type And Matching Mode

In this experiment, we analyzed the performance with different combinations of SelectionClause and WindowTypeClause in our proposed extension. Figure 6.4 shows the graph for this experiment. The different combinations measured are:

- Tumbling, Maximal
- Tumbling, Incremental
- Sliding, Maximal
- Sliding, Incremental
- Sliding, AllMatch

In this experiment, we detect a increasing price pattern for a stock tick. The length of total data set was varied for each group of reading same as was done in the earlier experiments. The pattern matching query that are running in this case is:

```
pattern $p <<window, selection>> in $seq
  $a as element()+ pprev $p1, pcur $c1 when xs:integer($p1/@price) lt xs:integer($c1/@price)
  return $p
```
6.3 Impact Analysis Of Main Elements

The window, selection is replaced by the combinations described above and the patterns of the prices matched are of kind (10, 20, 33, 44).

We can observe in the figure 6.4 that the running time for all combinations is in ascending order of tumbling maximal, tumbling incremental, sliding maximal, sliding incremental, sliding allmatch. This is expected by the definition earlier mentioned, on general data distribution.

However, there is a good difference in running times for sliding allmatch and sliding incremental versus less difference in running times for sliding incremental and sliding maximal. The latter can be easily understood for this particular pattern case as sliding incremental case would just additionally produce windows compared to sliding maximal in incremental phase of the particular pattern i.e. until it can grow in a complete pattern match.

Also, sliding allmatch for this example, would produce all the possible matches i.e if we have a particular pattern match of maximum length 5, then it would report all possible 10 match combinations. This is the reason of having huge running time for the sliding allmatch.

![Figure 6.4: Parametric comparison](image)

Since the total number of pattern matches in each case of window, selection varies, it would be interesting to compare the running time per item in the above experiment. Figure 6.5 shows the normalized running time comparison for the same data set and query set up as we had for Figure 6.4. The graph in figure 6.5 clearly shows the highest cost per item for tumbling maximal case. We have lowest number of matches (in 100's) for tumbling...
maximal case by its inherent property of skipping the last pattern match but this comes with a high cost of per item. Also, sliding, allmatch which has highest number of pattern matches (in 10000’s) has lowest cost per item in terms of running time. This analysis is useful in the scenarios where pattern matching decisions are needed to be taken for high number of data items.

6.3.2 Pattern Variable

Here we focus on the most interesting one which is 'all'. In this experiment, we used the same setup as we had in section 6.3.1. We changed the query to include one of the PatternVars, all in it and kept the rest as same. The modified query is:

```
pattern $p <<window, selection>> in $seq
$a as element()+ pprev $p1, pcur $c1, all $d1
when xs:integer($p1/@price) lt xs:integer($c1/@price)
return $p
```

We are not using $d1 in the ExprSingle of our query but from the implementation point of view, the binding of $d1 is performed every time while evaluation. It would be interesting to analyze the effect of this over the running time. We performed experiments using window, selection as tumbling, maximal and sliding, maximal. The results are shown in Figure 6.6 and Figure 6.7. These graphs clearly show that all variable makes considerable effect on the running
time of the query to that of without using it. This can be used as a future optimization, if any PatternVars is not present in the ExprSingle of the query then it could be left out for the binding.

Figure 6.6: Effect of all in Tumbling Maximal
Figure 6.7: Effect of all in Sliding Maximal
In this thesis, we proposed our pattern matching extension for XQuery that is seamlessly integrated with existing FFLWOR expression.

At first, we evaluated the pattern matching possibilities with the existing FORSEQ clause. We discussed some patterns that can be matched with FORSEQ clause using alone or combinations of tumbling, sliding or landmark windows. But at the one end pattern matching was expensive due to associated cost of making joins (that we showed in our benchmarking results) and also large set of queries were either complex or cannot be written using current XQuery model.

Our proposed pattern matching extension comes handy for such queries and far less expensive to that of FORSEQ clause, again was shown in the benchmarking results. Pattern matching is a complex topic and there can be close to infinite possibilities but semantic expressiveness of our proposal cover large number of cases out of it. Our proposal covers properties of a pattern by making use of SelectionClause, WindowTypeClause, TypeDecl and PatternVars. All these parameters are powerful enough to choose a pattern out of different possibilities. As we mentioned in section 1.3.1 about related works in SQL, one of the important difference with our proposed pattern extension is that “type of elements to be matched” can be specified where as in MATCH_RECOGNIZE[9] nothing as such could be mentioned.

Pattern matching clause was implemented in the MXQuery project and uses many existing implemented features of it. One such feature is window iterators that helps in materialization. Our implementation of pattern matching clause is based on finite state machine which makes transitions based on running states. We ran benchmarking experiments using our implementation and experiments showed a fairly high difference in running time with the current XQuery model as we expected.

In this work, we were able to prove our concept that separate pattern matching clause is needed as an extension to the current XQuery model with the arguments presented and benchmarking results. Although our finite machine implemented is not purely naive but there is still room for further optimizations as future work. This pattern matching extension works on contiguous patterns
and in the same way it could be extended for non-contiguous patterns. Our current implementation does not support alternation based queries, “group by” clause, and grouping such as \((A^+B^+)^+\). The current implementation can be extended further to add these features.

Further optimizations such as based on advance parsing of the pattern to be matched would help in alternation based expressions. Also, parallel execution approach for pattern matching can be fairly explored.
Appendix A

Grammar for the extension

```plaintext
FLWORExpr := InitialClause IntermediateClause* ReturnClause
InitialClause := ForClause| LetClause| WindowClause| PatternClause
IntermediateClause := InitialClause| WhereClause| GroupByClause| OrderByClause| CountClause
PatternClause := "pattern" "$" VarName (WindowTypeClause) (SelectionClause) "in" ExprSingle (Pattern | RegExp)
RegExp := RegExpClause "when" (PatternDefClause)+
RegExpClause := Term | "(" RegExpClause ")" "or" "(" Term ")"
Term := Primary | Term Primary
Primary := VarRef | RegExpClause
Pattern := Term | "(" Pattern ")" "or" "(" Term ")"
Term := Primary | Term Primary
Primary := PatternDefClause | "(" Pattern ")"
PatternDefClause := "$"VarName TypeDecl (PatternVars)? "when" ExprSingle
ExprSingle

TypeDecl := "as" SeqType
SeqType := "element" "(" ")" OccurenceIndicator?
OccurrenceIndicator := "*" | "+" | "?"
WindowTypeClause := Tumbling| Sliding
SelectionClause := Maximal| Incremental| AllMatch
PatternVars := ("before"|"pprev"|"pcur"|"after"|"all") "$"Varname ("," ("before"|"pprev"|"pcur"|"after"|"all") "$"VarName)*
```
Grammar for the extension
Bibliography