Extending XQuery With A Pattern Matching Facility

Peter M. Fischer, Aayush Garg, and Kyumars Sheykh Esmaili

Systems Group
ETH Zurich
8092 Zurich, Switzerland
`petfisch@inf.ethz.ch,garga@student.ethz.ch,kyumarss@inf.ethz.ch`

Abstract. Considering the growing usage of XML for communication and data representation, the need for more advanced analytical capabilities on top of XQuery is emerging. In this regard, a pattern matching facility can be considered as a natural extension to empower XQuery. In this paper we first provide some use cases for XML pattern matching. After showing that current XQuery falls short in meeting basic requirements, we propose an extension to XQuery which imposes no changes into current model, while covering a wide range of important use cases. We also implemented our proposal into the MXQuery prototype and show through experiments that, compared to the existing pattern matching means, our extension is not only more expressive, but also more efficient.

Keywords: XQuery, Pattern Matching

1 Introduction

There is a growing interest in the area of applications that deal with finding patterns in data items. These applications include:

- Security applications to detect unusual behavior
- Financial applications to detect market trends
- RFID and Sensor data processing
- Complex document analysis and formatting

While expressing tree patterns is fairly natural in XPath (and XSLT for more expressiveness), there is no comprehensive approach for sequence pattern matching on XML that takes into account its properties, its data model(s), and the query languages. As the *FORSEQ* window extension [4] shows, XQuery fits nicely with sequence processing. The existing language features are, however, not sufficient for expressive and effective pattern matching.

As a running example, we define the sequence S:

$$S = (B, A, B, C, A, B, B, B, B, C) \qquad (1)$$

A simple pattern which specifies the occurrence of three consecutive $< B/ >$ elements (written as B for brevity) would see possible instances (B, B, B) at the positions (5, 6, 7), (6, 7, 8).

1.1   Example Use Case

MASTER [1] is a project which tries to solve the problem of compliance in Service-Oriented Architecture (SOA) systems. An important part of this project is SOA Monitoring. This is realized by observing the message flow between services and then checking particular properties on this message sequence.

For example, upon invocation of operations of a browsing/shopping service, the following events are captured from a service wrapper or message bus:

```
<event operation="Login" uid="511" time="10:00"/>
<event operation="Search" uid="101" time="10:01"/>
<event operation="Logout" uid="511" time="10:03"/>
```

The monitoring system is given a set of expected or not-expected patterns, which it detects in this event sequences, (e.g. the pattern `SearchNoBuy`) and report them. Since the messages are exchanged between services in XML format, relational pattern matching systems cannot be applied.

1.2   Requirements for a Pattern Matching Language

A language to describe patterns needs to cover several aspects:

Pattern Structure: Patterns can be represented in a number of ways. The most common approaches are *regular expressions*, and *temporal logic*; we chose regular expressions since they are more widely used. Regular expressions are composed of a set of variables that uses quantification and grouping. Supposing a regular expression is A^*B^+ applied on S, the following pattern match instances are generated: {B}, {A, B}, {A, B, B, B, B},{B, B, B, B}.

Variable Binding: In contrast to classical regular expressions, in which the variable symbols directly correspond to the symbols in in the sequence, general pattern matching requires more expressiveness: Let's assume that we want to detect a pattern in which there are several increasing values, followed by at least one decreasing value. The pattern structure is A^*B^+, where A is a sequence representing increasing values (e.g. as a predicate comparing consecutive values), and B accordingly, with a correlation to A.

Instance Relationship and Selection: In order to properly identify a pattern instance, a user needs to specify 1) how far a pattern should extend if there are fitting elements, and 2) where the next pattern matching instance should be started in relation to the curent one. For example, the well known '*' quantifier expresses zero or more matches. For a pattern A^* and a sequence (A,A,A), always returning (A,A,A) might not be the desired result. In certain scenarios, (A) and (A,A) would also be valid results.

1.3   Paper Structure

The paper is structured as follows: Section 2 and Section 3 describe pattern matching capabilites in the relational world and existing features of XQuery supporting patterns, respectively. Section 4 describes our XQuery pattern matching extension proposal. Section 5 describes our prototype implementation and a summary of the evaluation. Then we conclude in Section 6.

2   Related Work in Relational and Streaming Databases

A lot of previous work has been done in the field of pattern matching. For this work, the areas of relational database systems and data streams are most relevant. While most of these focus on optimizations, there is a number of language-oriented proposals. The ANSI 2007 proposal defines the MATCH_RECOGNIZE clause for SQL[3]: Patterns are defined as (restricted) regular expressions over sequences of rows, with an extensive discussion on how to bind the variables to columns, and how to relate match instances. SASE+[8] supports Kleene closure over event streams, and provides a formal analysis of the expressibility of this language. A particular emphasis has been given to event selection strategies, in particular non-contiguous matches. Cayuga[5] presents a query language based on Cayuga Algebra for naturally expressing complex event patterns. This query language uses many SQL-like constructs.

Compared to these approaches, our language proposal does not re-invent the wheel: We aim for similar pattern structure and match selection, so that implementation and optimization strategies can be utilized. A significant difference is in the area of variable binding expressions, which are richer in the XML world.

3   Existing Pattern Matching Features In XQuery

3.1   String Matching

XPath and XQuery provide textual pattern matching based on regular expression matching, such `fn:matches(string, pattern)`. The XQuery Fulltext facility also provides textual matching facilities, taking into account text or XML structure. While in particular the pattern structure part of `fn:matches` is quite rich, text functions are not really suitable for general pattern matching.

3.2   Tree Pattern Matching

The notion of pattern matching is often associated with tree pattern matching in an XML environment. There is a tremendous amount of theoretical work (e.g [9]) in this area; from a practical point of view XPath expressions (for rigid tree patterns) and XSLT (for unknown, complex structures) provide the necessary facilities. A recent work [6] applies the new higher-order functions in XQuery 1.1 to achieve XSLT-like recursive matching. XPath or XSLT tree pattern matching is by design limited to a single document; the expressiveness of pattern structure, variable binding, and match selection is quite restricted.

3.3   Window Clause

The window clause of XQuery 1.1 gives the ability of selecting subsequences over a possibly infinite stream. It added a new language construct for XQuery called *FORSEQ* that integrates seamlessly into the *FLWOR* expression. A detailed

description of the *FORSEQ* clause can be found in [4], here we will describe its basic features. Windows boundaries are determined using predicates, stated as `start` and `end` expressions, which can refer to "previous", "current", or "next" items at the current position as well, as the explicit position. Window instance relationships can be specified as *tumbling*, *sliding*, and *landmark*

Pattern Structure Simple patterns like A^+ can be expressed directly as windows which open on the occurrence of an A element and close whenever the next item is not an A element. More complex patterns need to be expressed differenly: Pattern structure needs to be expressed as joins over windows for the individual variables, with the window boundary positions as join criteria. Consider the following regular expression: A^+B^+C. A possible XQuery 1.1 query can be given as:

```
forseq $w in $seq/stream/event sliding 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 tumbling window
        start curItem $bx, position $bp when $bx/person eq "B"
        end nextItem $by, position $bq when not($by/person eq B")
        where $bp eq ($aq + 1)
        return
         ...
```

This query uses a FORSEQ to create a sequences of A+ variable bindings, with their end positions expressed in `$aq`. Nested into this expression is another FORSEQ expression to create a B+ sequence, which is joined on its start position `$bp`. This approach can then be repeated for other variables. While it is possible to express complex patterns this way, they are cumbersome to write and, as we will see in the experiment section, difficult to optimize. Additional problems arise when variables use the "*" quantifier, since FORSEQ cannot create bindings to empty sequences, thus necessitating a refactoring of the pattern. For details of this refactoring and more examples, we refer to the technical report[7].

Variable Bindings While the predicate-based window boundaries of FORSEQ are quite expressive, there is no support in the current XQuery model to close a window based on its contents. For example, assume that a trader has a daily limit of 50 million dollars. Expressing this specification using a tumbling window, the windows close at the close of the day, and then an aggregation of the trades will be performed. Clearly for this case, we will get the results only after the day has closed, so execeeding the limit is easily possible. When using a landmark window, it is opened for the change of the day and closed at every trade; then aggregation is performed. Since landmark windows can almost never be discarded, the cost for memory management and computations will be extremely high. In addition, the underlying tumbling nature of the windows is no longer visible in the query.

4   Extending XQuery with a Pattern Matching Facility

4.1   Overview

Since a simple extension of XQuery is not really feasible, we propose to integrate a *PatternClause* into the FLWORExpr, thus representing pattern instances (and possibly also the parts out of which they were composed) as variables bound to sequences, contributing to the "'tuple stream"' as defined in XQuery 1.1.

```
    FLWORExpr ::= InitialClause IntermediateClause* ReturnClause
InitialClause ::= ForClause|LetClause|WindowClause|PatternClause
PatternClause ::= "pattern" "$" VarName (WindowTypeClause)
                  (SelectionClause) "in" ExprSingle RegExpClause
                  "using" (PatternVarClause)+
```

Covering the requirements stated in Section 1.2, *WindowTypeClause* and *SelectionClause* are introduced to cover Match Selection, *RegExpClause* for the pattern structure, and *PatternVarClause*es to determine the variables used in the RegExpClause. A pattern like A^+B^+C would be expressed follows:

```
pattern $p tumbling maximal in $seq $a $b $c using
    $a as item()+ pcur $q1 when $q1 eq ''A''
    $b as item()+ pcur $q2 when $q2 eq ''B''
    $c as item() pcur $q3 when $q3 eq ''C''
```

The output on S can be written (in "tuple" notation) as:

```
($p = {A,B,C}, $a = {A}, $b = {B}, $c = {C})
($p = {A,B,B,B,B,C}, $a = {A}, $b = {B,B,B,B}, $c = {C})
```

4.2   Pattern Structure as Regular Expression

Similar to the relational approaches, we represent the structure as regular expressions, where the alphabet is formed by the variables later defined in the *PatternVarClauses*. Repetition of groups (aka nested repetitions, such as $((AB)*C)+)$ has not gained much support in the relation world due to the possibly high evaluation cost and the lack of use cases [10, 8, 5]. We therefore exclude nested groups, which brings an additional benefit, as we can use normal XQuery variables as regular expression variables, simplifying the integration. In Sections 4.3 and 4.5, we show how to express nesting using the composability of XQuery and the `Pattern` clause. The proposed grammar thus looks as follows:

```
RegExpClause ::= RegTerm | "(" RegExpClause ")" "or" "(" RegTerm ")"
    RegTerm ::= RegPrimary | RegTerm RegPrimary
  RegPrimary ::= VarRef | RegExpClause
```

Patterns are contiguous, i.e., there must be no gap in the bindings of the underlying stream to pattern variables. A common case of such gaps are partitions; these are naturally supported (see Section 4.5). For other cases, these gaps can be simulated using dummy variables.

4.3   Pattern Variable Clauses

Since we use regular XQuery variables in the pattern structure, we define a binding expression, in which a predicate is evaluated over a candidate sequence:

```
PatternVarClause ::= "$"VarName SequenceType (PatternVars)? "when"
                     ExprSingle
```

SequenceType contains an XQuery type, which acts as a type selector, while its occurrence indicator becomes the quantifier. As an example, when declaring a variable a as part of a^+, containing at least one $<a/>$ element, we can write:

```
$a as element(a)+ when fn:true()
```

We propose five variables to express predicates over the (candidate) sequence:

- `before` Represents the element directly before the sequence
- `after` Represents the element directly after the sequence
- `all` Denotes all of the elements of the sequence so far
- `pprev` Running variable representing the previous item in the sequence
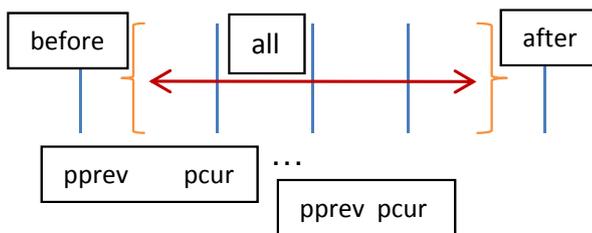- `pcur` Running variable representing the current item in the sequence

Fig. 1. Pattern Variables

The set of variables form a superset of the variables in the window clause in [4], since `all` now provides access to the contents of the sequence observed after `before`, not just the boundary elements. `pprev` and `pcur` are essentially syntactic sugar that simplify writing monotonic relationships over the sequence. Instead of using these five variables, we also provide the possibility to expose an unrestricted sequence **nested** to a nested *Window* or *Pattern Expression*, on which the first produced sequence is bound to the variable defined by nested.

To summarize the semantics: items become part of the pattern variable as long as the type matches, the occurrence indicator allows adding more elements, and computing the effective booleans value (EBV) of ExprSingle gives *true*.

Table 1 gives an example of the variable bindings for the pattern A^+, where A is an increasing value in the sequence, applied to the sequence $\{10, 15, 20, 25, 30, 5\}$. The pattern variable clause can be written as:

```
$p as xs:integer+  pprev $b, pcur $c when $b lt $c
```

The result includes 15 to 30, since comparing () and $\{10\}$ yields *false*.

before($a)	pprev($b)	pcur($c)	all($u)	after($d)	ExprSingle
()	()	10	(10)	15	false
10	10	15	(15)	20	true
10	15	20	(15, 20)	25	true
10	20	25	(15, 20, 25)	30	true
10	25	30	(15, 20, 25, 30)	5	true
10	30	5	(15, 20, 25, 30, 5)	()	false

Table 1. Evaluation Stages and Variable Bindings for A^+, increasing values

4.4 Match Selection

The relationship of the individual pattern instances and the selection of the variable parts play an important role in the semantics of the pattern clause. Table 2 gives an example of the interaction of the individual modifiers, which we will new explain now step by step.

Selection Policy: The *SelectionClause* provides three options, which determine how much of a sequence should be matched for greedy operators such as * or $^+$. This is similar to MATCH_RECOGNIZE; FORSEQ does not have such issues, since fulfilling a predicate is precise.

```
SelectionClause ::= AllMatch | Maximal| Incremental
```

- `AllMatch` gives the all possible variants of patterns in the sequence
- `Maximal` mode finds only the longest matching pattern
- `Incremental` mode increases the match candidates incrementally by a single item (starting from the empty sequence) and applies the `Maximal` match to every such partition. The union of all such matches then forms the output

Restart Policy The *WindowTypeClause* specifies where to start a new pattern:

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

- `Tumbling` specifies that the next pattern will be searched after the last element of the previous pattern, ensuring non-overlapping patterns.
- `Sliding` allows overlapping patterns. It matches the next pattern past the first element of the previous pattern which is not part of an existing result.

The semantics of both `Tumbling` and `Sliding` clauses correspond closely to both the clauses of the same name in the Window Extension [4], besides suppressing already contained pattern instances, as well as the SKIP PAST LAST ROW and SKIP TO NEXT ROW clauses in MATCH_RECOGNIZE [3].

Order of Matches Results are produced by the order of the first element of the pattern, with shorter pattern instances (generated by `INCREMENTAL` or `ALLMATCH`) before longer matches. This is again similar to what MATCH_RECOGNIZE and the Window Clause in XQuery 1.1 do. If ordering by the end of a pattern instance is needed, this can be achieved by using an `order by` clause.

	Maximal	Incremental	AllMatch
Tumbling	$\{a_1,a_2,b_1,b_2\}$ $\{a_3,b_3\}$	$\{a_1\}$ $\{a_1,a_2\}$ $\{a_1,a_2,b_1\}$ $\{a_1,a_2,b_1,b_2\}$ {\color{red}$\{a_3\}$} $\{a_3,b_3\}$	$\{a_1\}$ $\{a_1,a_2\}$ $\{a_1,a_2,b_1\}$ $\{a_1,a_2,b_1,b_2\}$ $\{a_2\}$ $\{a_2,b_1\}$ $\{a_2,b_1,b_2\}$ $\{b_1\}$ $\{b_1,b_2\}$ $\{b_1,b_2,a_3\}$ $\{b_2\}$ $\{b_2,a_3\}$ $\{a_3\}$ $\{a_3,b_3\}$ $\{b_3\}$
Sliding	$\{a_1,a_2,b_1,b_2\}$ $\mathbf{\{b_1,b_2,a_3\}}$ $\{a_3,b_3\}$	$\{a_1\}$ $\{a_1,a_2\}$ $\{a_1,a_2,b_1\}$ $\{a_1,a_2,b_1,b_2\}$ {\color{red}$\{b_1,b_2,a_3\}$} $\{a_3,b_3\}$	

Table 2. Selection for Pattern: (A*B*) or (B*A*); Input: $a_1 a_2 b_1 b_2 a_3 b_3$

4.5 Wrapping Up

We have compiled a list of use cases that shows the expressiveness and usefulness of our proposed pattern extension [7]. Given the space constraints, we are only showing a few interesting aspects here:

Partitioning: A typical use case is the detection of patterns on partitions of the original sequence, e.g., the shopping habits of an individual user in the general clickstream of a web store. Given the integration of the pattern clause into the FLWOR expression, this can be expressed very naturally:

```
for $a in $events
    group by $a/user
    pattern $p ... in $a
    ...
    return  <p> $p </p>
```

`Group by` splits the variable binding/tuple stream into n substreams according to the values generated by $a/user, assigning all relevant variables, in this case $a. The nested pattern clauses then work on each of these partitions individually. It is also possible to apply the partitioning after detecting a pattern.

Repeating Groups and Nested Variables: As stated above, we do not allow patterns like $(A^+B^*)^*$ to be expressed directly. Given the composable nature of XQuery, we can express this by using a nested pattern clause as input for the full pattern clauses (similar to [5, 8]) or by variables using the `nested` keyword:

```
pattern $p  in $events $v using
$v item ()* nested $n when pattern $pn in $n ($a $b) using $a
              element(a)+ when fn:true() $b element (b) when
                                    fn:true() return $pn[1]
    return <p> $p </p>
```

The sequence of the values considered for $v (named $n) is consumed by the nested pattern expression, yielding matches, which are then bound to to $v.

5   Implementation and Evaluation

We extended the Micro XQuery Engine (MXQuery) [2] and integrated an initial implementation of our proposed pattern matching extension into it. MXQuery lends itself well for the pattern extension, since it pioneered the XQuery Window extension [4] and the extension of XDM to infinite data [4]. Its architecture follows a classic parser/optimizer/runtime approach: It uses an iterator-based runtime, and a token-based representation of XDM. Given the existing MX-Query infrastructure, the initial implementation of the pattern matching extension could be done using roughly 1500 LOC. The bulk of the extension went to the runtime, in which the logic of a pattern structure matching, variable binding, and match selection were encapsulated into an iterator. This iterator follows the same implementation as the FOR, LET, and FORSEQ iterators in terms of variable binding "mechanics". The predicate evaluation for pattern variable clauses is then mapped on existing iterators, using the variables bound by the pattern iterator. Many optimizations are possible in the predicate evaluation, but they are not implemented yet; in particular, incremental evaluation on predicates over `all` would be promising. The *RegExpClause*, and the corresponding *PatternVarClause*s can be translated in a fairly straightforward way into a Non-deterministic Finite State Machine (NFA), similar to the approaches taken in relational pattern matching [5, 8]. It should be noted that expressing the pattern clause as a single iterator encapsulating a NFA is just one possibility to implement it. Other approaches might express the pattern as a combination of relational or XML iterators, similar to recent approaches in event processing [10].

In order to validate the quality and usefulness of our implementation, we carried out a number of performance experiments. The main goals were to compare the cost relative to the FORSEQ-based translations of patterns, and to establish an understanding of the impact of particular parameters on the pattern matching implementations. Due to space restrictions, we are only summarizing the results of two experiments here, more results can be found in the technical report [7]. All experiments were run on an Intel Xeon 3050, 2.13 GHz with 8GB RAM, running Windows Vista, 64 bit version and the latest Sun JVM.

We evaluated the pattern A^+B^+C, which can be translated to FORSEQ fairly directly using joins, as shown in Section 3. Table 3 shows the timing result of a (`sliding, maximal`) pattern matching clause and a `sliding` window in the *FORSEQ* clause on a flat sequence of elements *A,B* and *C*. The pattern extension is significantly more efficient than the *FORSEQ* translation, creating only a linear overhead over the parsing and FLWOR execution cost, since multi-sequence joins can be avoided. In contrast, the cost of FORSEQ increases in a $O(n^2)$ fashion because of the two-way join for the B and C elements. We performed similar experiments for different patterns expressed using FORSEQ, and the pattern clause; they showed the same trends and an even greater benefit for more complex patterns. For trivial patterns (like $A+$ for a simple window condition), the costs for FORSEQ and the pattern clause were comparable, as nearly the same work needed to be performed.

Sequence Length (items)	50	100	500	1000	5000	10000
Parse+FLWOR	0.006	0.01	0.02	0.027	0.092	0.172
Pattern	0.035	0.031	0.177	0.312	1.191	2.312
FORSEQ	0.14	0.051	1.429	5.906	113.671	438.73

Table 3. Processing time (s): Pattern A^+B^+C sliding maximal

We also performed a brief sensitivity analysis for the individual pattern structure and match selection parameters. Since these are very similar to the related relational approaches, we saw similar tradeoffs there. For example, the different combinations of *SelectionClause* and *WindowClause* on the same pattern structure/variables behaved as expected: The cost of running each variant had an almost linear relationship to the size of the output sets produced by each of the variants, showing that even our straightforward implementation provided a good scaleup to more complex requirements.

6   Conclusions

In this paper, we proposed our pattern matching extension for XQuery that is seamlessly integrated with existing *FLWOR*. We found that pattern matching is a needed and useful feature in XQuery and, since the current support for it is not sufficient, a separate pattern matching clause is needed. The proposal provides rich match selection and variable binding semantics, while balancing pattern structure expressiveness and cost in a similar way as relational approaches. Implementing this proposal on an open-source XQuery engine, and relevant benchmarking results prove that our extension is practical.

References

1. MASTER: Managing Assurance, Security, and Trust in sERvices. http://www.master-fp7.eu/.
2. MXQuery. http://mxquery.org/.
3. Anonymous. Pattern matching in sequences of rows. Technical report, 2007.
4. I. Botan et al. Extending XQuery with Window Functions. In *VLDB*, 2007.
5. L. Brenna et al. Cayuga: a High-Performance Event Processing Engine. In *SIGMOD*, 2007.
6. W. Candillon, M. Brantner, and D. Knochenwefel. XQuery Design Patterns. In *Balisage*, 2010.
7. A. Garg. Pattern Matching In XQuey. In *Master's Thesis, ETH Zurich*, 2010.
8. D. Gyllstrom et al. SASE: Complex Event Processing over Streams. In *CIDR*, 2007.
9. J. Lu, T. W. Ling, Z. Bao, and C. Wang. Extended XML Tree Pattern Matching: Theories and Algorithms. In *TKDE*, 2010.
10. Y. Mei and S. Madden. ZStream: a Cost-Based Query Processor for Adaptively Detecting Composite Events. In *SIGMOD*, 2009.