Complex Pattern Matching in Complex Structures: the XSeq Approach

Kai Zeng #1, Mohan Yang #2, Barzan Mozafari *3, Carlo Zaniolo #4

#University of California, Los Angeles, CA
1,2,4{kzeng,yang,zaniolo}@cs.ucla.edu

*Massachusetts Institute of Technology, Cambridge, MA
3barzan@csail.mit.edu

***Abstract*—There is much current interest in applications of *complex event processing* over data streams and of *complex pattern matching* over stored sequences. While some applications use streams of flat records, XML and various semi-structured information formats are preferred by many others—in particular, applications that deal with domain science, social networks, RSS feeds, and finance. XSeq and its system improve complex pattern matching technology significantly, both in terms of expressive power and efficient implementation. XSeq achieves higher expressiveness through an extension of XPath based on Kleene-* pattern constructs, and achieves very efficient execution, on both stored and streaming data, using Visibly Pushdown Automata (VPA). In our demo, we will (i) show examples of XSeq in different application domains, (ii) explain its compilation/query optimization techniques and show the speed-ups they deliver, and (iii) demonstrate how powerful and efficient application-specific languages were implemented by superimposing simple 'skins' on XSeq and its system.**

I. Introduction

There has been much interest in querying massive collections of data in order to discover useful patterns in biological data, user behavior, social networks, financial data analysis, and many others. XML provides a very popular data exchange format that is often used in these applications, along with languages such XPath and XQuery. However, experience with processing XML streams has revealed that they present several limitations, both in terms of expressive power and amenability to efficient implementation, which can be effectively addressed by Kleene-closure constructs by use of Visibly Pushdown Automata (VPA) at the implementation level [1]. VPA [2] and Nested Words [3] provide a natural extension to the finite state automata (FSA) model over XML [4], which have proven to be very effective, inasmuch as they provide a scalable technology that achieves the right balance between expressiveness and tractability [1].

In this demo, we will (i) show examples of how XSeq system extends XPath with powerful Kleene-* constructs and how these new constructs are critical in various application domains, (ii) explain the compilation/query optimization techniques used in our system and show the speed-ups they deliver, and (iii) demonstrate how powerful and efficient application-specific languages were implemented by superimposing simple 'skins' on XSeq and its system.

In the next section, we briefly overview the XSeq language, and in section III, we present an outline of the architecture of our engine. In Section IV, we demonstrate several examples from a wide range of interesting application domains. We highlight the demonstration of ease-of-use and performance of XSeq system in Section V, and present our conclusions in Section VI.

II. The XSeq Query Language

As our first example, let us consider a stored XML document describing the clinical trial records of a particular drug, which consists of a bunch of testers, each with multiple prescriptions. Figure 1 shows a snippet of such an XML record.

```
<drugTests>
  <test @patientID='2012312'>
    <prescription @date='2008-07-14', @dosage='350' />
    <prescription @date='2008-07-20', @dosage='400' />
    <prescription @date='2008-07-27', @dosage='420' />
    ...
  </test>
  <test ...
</drugTests>
```

Fig. 1. The clinical trial records.

Example 1: Say that, for each patient participating the drug test, we must identify the longest consecutive period in which he/she took a dosage of higher than 400 milligram each time. Two trials are consecutive if they are no more than 7 days apart. This example is expressed by the following query:

Query 1:

```
return $T@ID, max(last($X)@date - $P@date)
from //$T /$P (\$X)*
partition by //$T @ID
where tag($T)='test' and tag($X)='prescription'
and tag($P)='prescription'
and prev($X)@date + 7 >= $X@date
and $P@date + 7 >= first($X)@date
and $P@dosage >= 400 and $X@dosage >= 400
```

Here, the path expression consists of named steps, where variables start with \$. `tag($X)` returns the XML tag name of variable `$X`.

As illustrated by this example and Table I, XSeq inherits most constructs and their similar semantics from the navigational fragments of XPath (e.g., axes and attributes), but also extends the syntax of XPath with a few but powerful constructs: (i) the Kleene-* and (ii) *immediate following* axes, i.e., *immediate_following_sibling* and *first_child*.

Kleene-*. XSeq supports Kleene-* expressions in path expressions, where a Kleene-* expression `A*` is defined as the infinite

Axis	Shorthand
child	/
descendant	//
following_sibling	λ (empty string)
immediate_following_sibling	\
first_child	/\

TABLE I
IN ADDITION TO XPATH AXES, XSEQ HAS '\'

union $\emptyset \cup \mathtt{A} \cup (\mathtt{AA}) \cup (\mathtt{AAA}) \cup \cdots$. Thus, in Query 1, $(\backslash\$\mathtt{X})^*$ means zero or more repetition of $\backslash\$\mathtt{X}$.

Order Semantics, Aggregates. XSeq is a sequence query language. Therefore, unlike XPath where the input and output are sets, the input and the output of an XSeq query are sequences. In XSeq, the input data is viewed as a pre-order traversal of the XML tree. For conciseness in expressing horizontal sequence patterns, the default axis of XSeq is the *following_sibling* axis. XSeq also introduces the *immediately following* notion, which brings XSeq a clear advantage over all previous extensions of XPath in terms of expressiveness, succinctness and optimizability. With the *immediate_following_sibling* axis, XSeq can easily express a consecutive sequence, e.g., $(\backslash\$\mathtt{X})^*$ in Query 1 represents consecutive prescriptions, while XPath needs to use double negation to express the same meaning. Similarly, the *first_child* axis navigates to the very first child node under the current one. Besides the traditional aggregates (e.g. sum, max), XSeq also supports sequential aggregates (i.e., first, last, prev) which are only applied to variables from the Kleene-* expression. For instance, `last($X)` returns the last `$X` in the $(\backslash\$\mathtt{X})^*$ sequence. Similarly, `first($X)` returns the first node of $(\backslash\$\mathtt{X})^*$, and `prev($X)` returns the node before the current node of the sequence. As an example, `prev($X)@date + 7 >= $X@date` ensures that two trials are no longer than 7 days apart.

Partition By. XSeq supports partitioning the input XML data by their keys. As shown in Query 1, the data is partitioned by patients' IDs, whereby the search for the longest period for each patient can be done in parallel. Although this construct does not add to the expressiveness, it provides a more concise syntax for complex queries and better opportunities for optimization. However, XSeq only allows partitioning by an attribute field, and requires that except this attribute, the rest of the partitioning clause is a prefix of the path expression in the from clause. This constraint is important for ensuring efficiency and also for avoiding queries with ill semantics. The formal syntax and semantics of XSeq can be found in [1].

III. SYSTEM OVERVIEW

The high-level architecture of XSeq system is depicted in Figure 2. The XSeq system is a general-purpose pattern matching/complex event process engine, where (i) users can submit their XSeq queries through a web-based general purpose client to query XML data in many domains, such as genomic data, financial data, temporal databases and software traces; (ii) for specific applications, in particular genomic data analysis, users can specify the patterns by regular expressions via specialized application interfaces, which helps automatically translate the patterns into XSeq queries. Specialized application interfaces

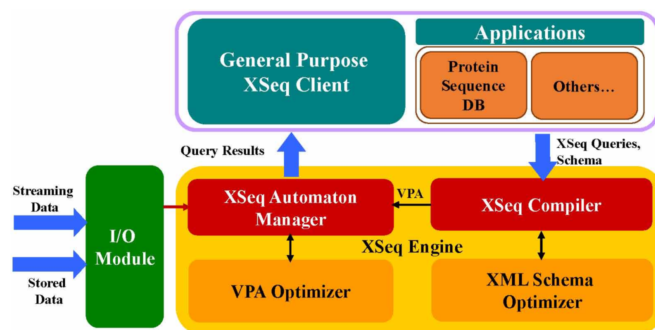


Fig. 2. System architecture

are also available to visualize the detected protein patterns, as shown in Figure 4.

Our XSeq engine comprises of four major modules. XSeq queries, once submitted, are first compiled by the XSeq compiler into a raw query plan, which is a VPA. During the compilation, the XSeq compiler consults with the XML schema optimizer module to generate better optimized VPA. This compiled VPA is handed to the XSeq automaton manager, which consults with the VPA optimizer and further optimizes the VPA using VPSearch algorithm. The final query plan is then executed by the automaton manager. The XSeq engine relies on the I/O module which provides access methods for both stored and streaming XML data. The I/O module serializes the stored data or hands over the streaming data, and presents a sequence of inputs to the automaton manager.

IV. ADVANCED APPLICATIONS

A. Genomic Data Analysis

XML is a standard data exchange format in Molecular Biology. Many biology databases provide data in XML format. Searching complex patterns in proteins, RNA and DNA sequences plays an important role in the study of genomics, pharmacy and so on. For instance, the *structural motifs* are important supersecondary structures in proteins, which have close relationships with the biological functions of the protein sequences. These motifs are of a large variety of structural patterns, usually very complex, e.g., the β-meander motif is composed of two or more consecutive antiparallel β-strands linked together, as depicted in Figure 3(a)[1].

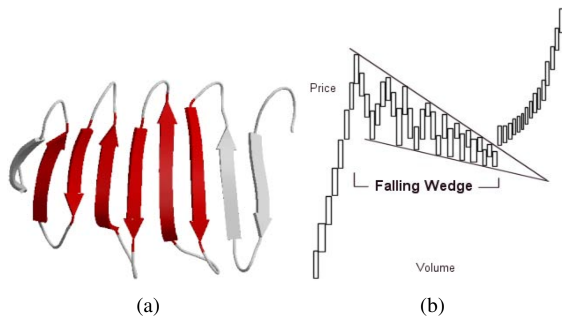
Consider now protein data with a simplified schema as below. Example 2 uses XSeq to detect such motifs.

```
<!DOCTYPE uniprot [
<!ELEMENT uniprot (protein)*>
<!ELEMENT protein (fullName, feature+)>
<!ELEMENT fullName (#PCDATA)>
<!ATTLIST feature type CDATA #REQUIRED> ]>
```

Example 2 (Detecting β-meander motifs):

```
return $N/text()
from //protein[$N] /$F \$G (\$H)*
where tag($N) = 'fullName' and tag($F) = 'feature'
and tag($G) = 'feature' and tag($H) = 'feature'
and $F@type = 'beta-strand'
and $G@type = 'beta-strand'
and $H@type = 'beta-strand'
```

[1] http://en.wikipedia.org/wiki/Beta_sheet



(a) (b)

Fig. 3. (a) The β-meander motif (b) The falling wedge pattern

In our demonstration, we will provide a protein dataset (from the famous uniprot database[2]) to allow our audience to interact with XSeq. For users who are not familiar with XSeq, we will let them to use our specialized web-based client to write regular expression style sequence patterns. Our system automatically translates arbitrary regular expression patterns into equivalent XSeq queries, and visualizes the detected motif sites on the protein sequences through graphical views, as shown in Figure 4.

B. Financial Data Analysis

We will use stock data from Yahoo! Finance[3] to allow the audience to issue queries to search for patterns of their interest. For instance, the falling wedge pattern, as shown in Figure 3(b)[4], is a bullish presage of an uptrend.

```
<!DOCTYPE prices [
<!ELEMENT prices (row)*>
<!ATTLIST row date CDATA #REQUIRED>
<!ATTLIST row open CDATA #REQUIRED>
<!ATTLIST row close CDATA #REQUIRED>
<!ATTLIST row volume CDATA #REQUIRED> ]>
```

Example 3 (Falling wedge pattern):

```
return $R@close, last($Y)@close
from //$R ((\$S)* \$X (\$T)* \$Y)*
where tag($R) = 'row' and tag($S) = 'row'
and tag($X) = 'row' and tag($T) = 'row'
and tag($Y) = 'row'
and $R@close > first($S)@close
and prev($S)@close > $S@close
and last($S)@close > $X@close
and $X@close < first($T)@close
and prev($T)@close < $T@close
and last($T)@close < $Y@close
and prev($X)@close < $X@close
and prev($Y)H@close > $Y@close
```

C. Temporal Queries

Traditional temporal databases use a state-oriented representation, where tuples of a database are time-stamped with their maximal period of validity. This state-based representation requires temporal coalescing and/or temporal joins even for basic query operations (e.g. projection), and are thus prone to inefficient execution. Some recent research work has proposed using XML-based event-oriented representation for transaction-time temporal database, where value updates in database history are recorded as events [5], [6], [7]. For example, below is the DTD of a temporal employee XML, where each employee has a sequence of *salary* and *dept* elements time-stamped by the *tstart*, *tend* attributes, representing the update events ordered by their start time in the database's evolution history.

[2] uniprot: http://www.uniprot.org

[3] Yahoo! Finance: http://finance.yahoo.com

[4] http://en.wikipedia.org/wiki/Wedge_pattern

Fig. 4. The web client for complex pattern matching in protein sequences

```
<!DOCTYPE employees [
<!ELEMENT employees (employee*)>
<!ELEMENT employee (name (salary | dept)+)>
<!ATTLIST employee id CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT salary (#PCDATA)>
<!ELEMENT dept (#PCDATA)>
<!ATTLIST salary tstart CDATA #REQUIRED>
<!ATTLIST salary tend CDATA #IMPLIED>
<!ATTLIST dept tstart CDATA #REQUIRED>
<!ATTLIST dept tend CDATA #IMPLIED> ]>
```

XSeq is a powerful event-oriented temporal language, which can express effectively the basic temporal operations (e.g., temporal joins and temporal coalescing), as well as very complex temporal sequence patterns. This can be illustrated by the following example.

Example 4: Find employees who have risen quickly without changing department. More precisely, we want to find employees who

1) once hired (with some salary and into some department),
2) have gone through one or more salary adjustments, followed by
3) a transfer to another department,
4) for a final salary that is 40% above the initial one.

This complex pattern can be expressed succinctly by Query 2.

Query 2:

```
return $X
from //employee[@$X] /$A \$B \$C (\$D)* \$E
where tag($A) = 'salary' and tag($B) = 'dept'
and tag($C) = 'salary' and tag($D) = 'salary'
and tag($E) = 'dept' and tag($X) = 'id'
and $E/text() <> $B/text()
and last($D)/text() > 1.4 * $A/text()
```

In this demo, users will experience the simplicity and power of XSeq by writing temporal queries over XML-based temporal dataset.

D. Software Analysis

Modern programming languages and software frameworks offer ample support for debugging and monitoring applications. For example, in the .NET framework, the *System.Diagnostics* namespace contains flexible classes which can be easily incorporated into applications to output runtime debug/trace information as XML files. The following XML snippet shows a software trace of a function `fibonacci` that recursively called itself but in the end threw out an exception.

```
<main>
  ...
  <fibonacci @input = '500'>
    <fibonacci @input = '499'>
      ...
      <exception @msg = 'overfolow' />
    </fibonacci>
  </fibanacci>
  ...
</main>
```

Searching and analyzing the patterns in software traces could help debugging. For example, we can easily identify the input to the last iteration of the function `fibonacci` and the depth of the recursive calls by

Query 3:

```
return last($F)@input, count($F)
from //$X (/$F)* /$E
where tag($X) != 'fibonacci'
and tag($F) = 'fibonacci'
and tag($E) = 'exception'
```

Our XSeq system can also help database administrators, given that many commercial databases can export query plans in XML format. Thus our administrators can write XSeq queries to detect patterns of interest, in order to locate performance bottlenecks, e.g., the most costly two-way join in a large block of multi-way joins.

During the demonstration, we will provide XML-based software traces and SQL query plans, and let the audience raise interesting queries against the XSeq system.

V. Ease-of-Use and Performance

Our audience will be allowed to write their own sequence queries using our user-friendly interfaces designed for various application domains. Thus, those users who are interested in genomic data analysis, but are unfamiliar with regular expressions, will be able to use our web-based client to translate genomic patterns into XSeq, and visualize the detected patterns in graphical views (see Figure 4). However, more experienced users will work with our general-purpose client to try different XSeq queries over various XML data sets, ranging from financial data and temporal databases to software debugging traces. Finally, we will have other native XPath engines available so that audience can compare the performance of XSeq on both traditional XML queries and sequence queries. Our audience will be able to watch individual XSeq optimization techniques in action and experience their effectiveness.

As described in [1], the XSeq engine has both static and run-time optimizations. The main static optimizations are as follows:

Cutting the inferrable prefix. We can always remove the longest prefix of the pattern as long as (i) the prefix has not been referenced in the return or the where clause, and (ii) the omitted prefix can be inferred from the remaining suffix. Due to the sequential nature of VPA, such simplifications can greatly improve efficiency by reducing a global pattern search to a more local one.

Reducing non-determinism. Our algorithm for translating XSeq queries produces VPAs that are typically non-deterministic. Reducing non-determinism speeds up execution by avoiding many unnecessary backtracking. For instance, if we know that an element referred in the query contains no attributes and no subelements, we can remove the non-deterministic states corresponding to those attributes and subelements. This technique decreases non-determinism without incurring in the exponential memory costs which are common with full determinization of VPAs.

At query time, a straightforward evaluation of a VPA requires matching the pattern starting from every input element. However, inspired by the KMP algorithm, we use a similar backtracking minimization technique for VPAs—a technique called VPSearch [8]. During the demo, we will provide a high level description of these optimizations and then demonstrate their effectiveness by simply testing running times after each optimization is turned on or off.

VI. Conclusion

The XSeq system supports a powerful language extension to XPath which is effective at expressing complex sequence patterns over both stored and streaming XML data. XSeq is highly amenable to optimization and efficient implementation, and thus the XSeq system achieves excellent performance on both traditional and sequence XML queries. We have devised user-friendly interfaces, including a specialized client for translating regular expression style protein sequence pattern into XSeq queries, so that the users can gain the first-hand experience at writing XSeq queries. We will present several real-world examples from various application domains, such as genomic data analysis, financial data analysis, temporal databases and software analysis.

References

[1] B. Mozafari, K. Zeng, and C. Zaniolo, “High-performance complex event processing over xml streams,” in *SIGMOD Conference*, 2012, pp. 253–264.

[2] R. Alur and P. Madhusudan, “Visibly pushdown languages,” in *STOC*, 2004, pp. 202–211.

[3] ——, “Adding nesting structure to words,” *J. ACM*, vol. 56, no. 3, 2009.

[4] C. Koch, “Xml stream processing,” in *Encyclopedia of Database Systems*, 2009, pp. 3634–3637.

[5] T. Amagasa, M. Yoshikawa, and S. Uemura, “A data model for temporal xml documents,” in *DEXA*, 2000, pp. 334–344.

[6] F. Wang, C. Zaniolo, and X. Zhou, “Archis: an xml-based approach to transaction-time temporal database systems,” *VLDB J.*, vol. 17, no. 6, pp. 1445–1463, 2008.

[7] C. Zaniolo, “Event-oriented data models and temporal queries in transaction-time databases,” in *TIME*, 2009, pp. 47–53.

[8] B. Mozafari, K. Zeng, and C. Zaniolo, “From regular expressions to nested words: Unifying languages and query execution for relational and xml sequences,” *PVLDB*, vol. 3, no. 1, pp. 150–161, 2010.