High-Performance Complex Event Processing over XML Streams

Barzan Mozafari
UCLA
420 Westwood Plaza
Los Angeles, CA, USA
barzan@cs.ucla.edu

Kai Zeng
UCLA
420 Westwood Plaza
Los Angeles, CA, USA
kzeng@cs.ucla.edu

Carlo Zaniolo
UCLA
420 Westwood Plaza
Los Angeles, CA, USA
zaniolo@cs.ucla.edu

ABSTRACT

Much research attention has been given to delivering high-performance systems that are capable of *complex event processing* (CEP) in a wide range of applications. However, many current CEP systems focus on efficiently processing data having a simple structure, and are otherwise limited in their ability to efficiently support complex continuous queries on structured or semi-structured information. However, XML streams represent a very popular form of data exchange, comprising large portions of social network and RSS feeds, financial records, configuration files, and similar applications requiring advanced CEP queries. In this paper, we present the XSeq language and system that support CEP on XML streams, via an extension of XPath that is both powerful and amenable to an efficient implementation. Specifically, the XSeq language extends XPath with natural operators to express sequential and Kleene-* patterns over XML streams, while remaining highly amenable to efficient implementation. XSeq is designed to take full advantage of recent advances in the field of automata on Visibly Pushdown Automata (VPA), where higher expressive power can be achieved without compromising efficiency (whereas the amenability to efficient implementation was not demonstrated in XPath extensions previously proposed).

We illustrate XSeq's power for CEP applications through examples from different domains, and provide formal results on its expressiveness and complexity. Finally, we present several optimization techniques for XSeq queries. Our extensive experiments indicate that XSeq brings outstanding performance to CEP applications: two orders of magnitude improvement are obtained over the same queries executed in general-purpose XML engines.

Categories and Subject Descriptors

H.2.3 [**Information Systems**]: DATABASE MANAGEMENT—*Languages, Query languages*

Keywords

Complex Event Processing, XML, Visibly Pushdown Automata

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

1. INTRODUCTION

XPath is an important query language on its own merits and because it provides the kernel of other languages used in diverse applications, including XQuery, several graph languages [29], and OXPath for web information extraction [12]. Much work has also focused on the efficient support for XPath in the diverse computational environments required by these applications. In particular, finite state automata (FSA) have proven to be very effective at supporting XPath queries over XML streams [16], and are also apt at providing superior scalability through the right mix of determinism versus non-determinism. In fact, numerous XML engines have been successfully built for efficient and continuous processing of XML streams [9, 25, 24, 5, 13, 11, 10]. All these systems support full or fragments of XPath or XQuery, and thus, naturally inherit the pros and cons of these languages. The simplicity of XPath and the generality of XQuery have made them very successful and effective for general-purpose applications. However, these languages lack explicit constructs for expressing Kleene-* and sequential patterns—a vital requirement in many CEP applications[1]. As a result, while the existing engines remain very effective in general-purpose applications over XML streams, their usability for CEP applications (that involve complex patterns) becomes highly limited as none of them provide any explicit sequencing/Kleene-* constructs over XML.

To better illustrate the difficulty of expressing sequence queries in existing XML engines (that mostly support fragments of XPath/ XQuery), in Figure 1 we have expressed a common query from stock analysis in XPath 2.0, where the user is interested in a sequence of stocks with falling prices[2]. As shown in this example, due to the lack of explicit constructs for sequencing and Kleene-* patterns, the query in XPath/ XQuery is very hard to write and understand for humans and is difficult to optimize. In fact, it is not a surprise that these general-purpose XML engines perform two orders of magnitude slower on these complex sequential queries than the same queries expressed and executed in XSeq (the language and system presented in this paper), whereby explicit constructs for Kleene-* patterns and effective VPA-based optimizations allow for high-performance execution of CEP queries.

[1]There are several definitions of CEP applications [7, 18, 36], but they commonly involve three requirements: (i) complex predicates (filtering, correlation), (ii) temporal/order/sequential patterns, and (iii) transforming the event(s) into more complex structures. In this paper we mainly focus on (i) and (ii) while achieving (iii) represents a direction for future research, e.g. by embedding our language (called XSeq) inside XSLT.

[2]In fact, in practice, stock queries tend to be much more complex, e.g. in a wedge pattern (`www.investopedia.com`), the user seeks an arbitrary number of falling and rising phases of a particular stock.

```
  <result>{
for $t1 in doc("auction.xml")//Stock[@stock_symbol='DAGM'] return
<head>{$t1/@close}{
  for $t4 in $t1/following-sibling::Stock[@stock_symbol='DAGM'] where $t4/@close<=$t1/@close
   and (every $t2 in for $x in $t1/following-sibling::Stock[@stock_symbol='DAGM']
                     where $x<<$t4 return $x satisfies $t2/@close<=$t1/@close and $t2/@close>=$t4/@close)
   and (every $t2 in for $x in $t1/following-sibling::Stock[@stock_symbol='DAGM']
                     where $x<<$t4 return $x, $t3 in for $x in $t2/following-sibling::Stock[@stock_symbol='DAGM']
                     where $x<<$t4 return $x satisfies $t2/@close>=$t3/@close and $t3/@close>=$t4/@close)
  return <bottom> {$t4/@close} </bottom>
}  </head>
}</result>
```

Figure 1: A query in XPath 2.0/XQuery for a sequence of 'falling price' in Nasdaq's XML.

These limitations of XPath are not new. In fact, several more powerful extensions of XPath, have been previously proposed in the literature [31, 33, 32]. However, the efficient implementation of these extensions remained an open research challenge, which the papers proposing said extensions did not tackle neither for stored data nor for data streams. In fact, the following was declared to be an important open problem since 2006 [31]: *Efficient algorithms for computing the transitive closure of XPath path expressions.*

Fortunately, significant advances have been recently achieved in automata theory with the introduction of Nested Words [3] and Visible Pushdown Automata [2]. In fact, Nested Words and VPA strive to achieve a balance between expressiveness and tractability: unlike pushdown automata (PDA), VPAs have all the appealing properties of FSA. For instance, VPAs enjoy higher expressiveness (than word automata) and more succinctness (than tree automata), while their decision complexity and closure properties are analogous to the corresponding word automata, e.g., VPAs are closed under union, intersection, complementation, concatenation, and Kleene-*; their deterministic versions are as expressive as their non-deterministic counterparts; and membership, emptiness, language inclusion and equivalence are all decidable [2, 3]. However unlike word automata, VPAs can model and query well-nested data such as XML, RNA sequences and software traces [3].

Although these new types of automata can bring major benefits in terms of expressive power, to the best of our knowledge, their efficient implementation in the context of XPath-based query languages has not been explored before. In this paper, we introduce the XSeq language which achieves new levels of expressive power supported by a very efficient implementation technology. XSeq extends XPath with powerful constructs that support (i) the specification of and search for complex sequential patterns over XML, and (ii) efficient implementation using the Kleene-* optimization technology and streaming Visibly Pushdown Automata (VPA).

Being able to compile complex pattern queries into equivalent VPAs has several key benefits. First, it allows for expressing complex queries that are common in CEP applications. Second, it allows for efficient stream processing algorithms. Finally, the closeness of VPAs under *union* operation creates the same opportunities for CEP systems (through combining their corresponding VPAs) that the closeness of NFAs (non-deterministic finite automata) created for publish-subscribe systems [10, 35, 17] where simultaneous processing of massive number of queries becomes possible through merging the corresponding automata of the individual queries.

Contributions. We make the following contributions:

1. The design of XSeq, a powerful and user-friendly query language for CEP over XML.

2. An efficient implementation for XSeq based on VPA-based query plans, and several compile-time and run-time optimizations.

3. Formal results on the expressiveness of XSeq, and the complexity of its query evaluation and query containment.

4. An extensive empirical evaluation of XSeq system, using several well-known queries, datasets and engines.

Paper Organization. We present the main constructs of our language in Section 2 using simple examples. The generality and versatility of XSeq for expressing CEP queries is illustrated in Section 3 where several well-known queries are discussed. Our query execution and optimization techniques are presented in Section 4, followed by our formal results in Section 5. Our results are empirically validated in Section 6. After an overview of the related work in Section 7, we conclude in Section 8.

XSeqQuery	←	[*return* Output *from*] PathExpr [*partition by* PathExpr][*where* Condition]
Output	←	OutTerm[, Output]
OutTerm	←	OutBase[@*attr*] | Aggr '(' OutBase@*attr* ')'
OutBase	←	'$' *variable* | SeqAggr '(' *variable* ')'
Aggr	←	*max* | *min* | *count* | *sum* | *avg*
SeqAggr	←	*first* | *last* | *prev*
Axis	←	*self* | *child* | *parent* | *descendant* | *ancestor* | *following_sibling* | *preceding_sibling* | *child* '\' | *following_sibling* '\'
NameTest	←	*ID* | '*' | '$' *variable*
Step	←	Axis '::' NameTest
PathExpr	←	Step | PathExpr '/' PathExpr | PathExpr *union* PathExpr | PathExpr *intersect* PathExpr | PathExpr '[' NodeExpr ']' | '(' PathExpr ')' '*'
NodeExpr	←	PathExpr | Condition | *not* NodeExpr | NodeExpr *and* NodeExpr | NodeExpr *or* NodeExpr
Condition	←	*{See Appendix A.}*

Figure 2: Simplified Syntax of XSeq.

2. XSEQ QUERY LANGUAGE

In this section, we briefly introduce the query language supported by our CEP system, called XSeq. The simplified syntax of XSeq is given in Figure 2 which suffices for the sake of this presentation. Below we explain the semantics of XSeq via simple examples and leave the formal semantics in our technical report [23].

Inherited Constructs from Core XPath. The navigational fragments of XPath 1.0 and 2.0 are called, respectively, Core XPath 1.0 [33] and 2.0 [32]. The semantics of these common constructs are similar to XPath (e.g., axes, attributes). Other syntactic constructs of XPath (e.g. the $following$) can be easily expressed in terms of these main constructs (see [32]). In XSeq there are two new axes to express the *immediately following* [3] notion, namely $child\backslash$ and $following\_sibling\backslash$, which are described later on. Some of the axes in XSeq have shorthands:

Axis	Shorthand
self	•
child	/
descendant	//
following_sibling	λ (empty string, i.e. default axis)
following_sibling \	\
child \	/ \

[3]XSeq does not have analogous operators for immediately preceding since backward axes of XPath are rarely used in practice.

EXAMPLE 1 (**A family tree.**). *Our XML document is a family tree where every node has several attributes: Cname (for name), Bdate (for birthdate), Bplace (for the city of birth) and each node can contain an arbitrary number of sub-entities Son and Daughter. Under each node, the siblings are ordered by their Bdate.*

In the following, we use this schema as our running example.

EXAMPLE 2. *Find the birthday of Mary's sons.*

QUERY 3. //daughter[@Cname='Mary']/son/@Bdate

Kleene-* and parentheses. Similar to Regular XPath [31] and its dialects [33, 34], XSeq supports path expressions such as $/\mathtt{a}(/\mathtt{b}/\mathtt{c})^*/\mathtt{d}$, where a Kleene-* expression A^* is defined as the infinite union $\cdot \cup A \cup (A/A) \cup (A/A/A) \cup \cdots$

EXAMPLE 4. *Find those sons born in 'New York', who had a chain of male descendants in which all the intermediary sons were born in 'Los Angeles' and the last one was again born in 'New York'. For all such chains, return the name of the last son.*[4]

QUERY 5.
```
//son[@Bplace='NY'](/son[@Bplace='LA'])*/son[@Bplace='NY'
  ]/@Cname
```

The parentheses in $()^*$ can be omitted when there is no ambiguity. Also, note the difference between the semantics of $(/\mathtt{son})*$ and $//\mathtt{son}$: the latter only requires a son in the last step rather than the entire path.

Syntactic Alternatives. In XSeq, the node selection conditions can be alternatively moved to an optional `where` clause, in favor of readability. When a condition is moved to the `where` clause, its step should be replaced with a variable (variables in XSeq start with $). Also, similarly to XPath 2.0 and XQuery, the query output in XSeq can be moved to an optional `return` clause. Query 6 below is an alternative way of writing Query 5 in XSeq. Here, $\mathtt{tag}(\mathtt{X})$ returns the tag name of variable $\$X$.

QUERY 6.
```
return $B@Cname
from //son[@Bplace='NY'] (/$A)* /$B[@Bplace='NY']
where tag(A)='son' and A@Bplace='LA' and tag(B)='son'
```

For clarity, in this paper we mainly use this alternative syntax.

Order Semantics, Aggregates. XSeq is a *sequence query language*. Therefore, unlike XPath where the input and output are a set (or binary relation), in XSeq the XML stream is viewed as a pre-order traversal of the XML tree. Thus, both the input and the output of an XSeq query are a *sequence*. The XML nodes are ordered according to[5] their relative position in the XML document.

As a result, besides the traditional aggregates (e.g. sum, max), XSeq also supports sequential aggregates (`SeqAggr` in Figure 2) which are only applied to variables under a Kleene-* [6]. As a result, the $ before the variable names (otherwise used for differentiation from XML tags) is omitted when variables are passed to these functions, since their arguments cannot be XML tags. For instance, the path expression $/\mathtt{son}(/\$\mathtt{X})^*, \mathtt{last}(\mathtt{X})@\mathtt{name}$ returns the name of the last `X` in the $(/\$\mathtt{X})^*$ sequence. Similarly, $\mathtt{first}(\mathtt{X})$ returns the first node of the $(/\$\mathtt{X})^*$ and $\mathtt{prev}(\mathtt{X})$ returns the node before the current node of the sequence. Finally, $\mathtt{X@Bdate} > \mathtt{prev}(\mathtt{X})@\mathtt{Bdate}$ ensures that the nodes that match $(/\$\mathtt{X})^*$ are in increasing order of their birth date.

Siblings. Since XSeq is designed for complex sequential queries, its default axis (i.e. when no explicit axis is given) is the 'following_sibling'. The omission of the 'following_sibling' allows for concise expression of complex horizontal patterns.

EXAMPLE 7. *Find all the younger brothers of 'Mary'.*

QUERY 8.
```
return $S@Cname
 from // $D[@Cname='Mary'] $S
 where tag(D)='daughter' and tag(S)='son'
```

Here, since no other axes appear between `D` and `S`, they are treated as siblings.

Immediately Following. This is the construct that gives XSeq a clear advantage over all the previous extensions of XPath in terms of expressiveness, succinctness and optimizability. We believe that one of the main shortcomings of the previous XML languages for CEP applications is their lack of explicit constructs for expressing the notion of '*immediately following*' (see Section 3). Thus, to overcome this, XSeq provides two explicit axes, $\backslash$ and $/\backslash$, for *immediately following* semantics. For example, $\mathtt{Y}\backslash\mathtt{X}$ will return the *immediately next sibling* of node `Y`, while $\mathtt{Y}/\backslash\mathtt{X}$ will return the *very first child* of node `Y`. Similarly to other constructs, these operators return an empty set if no such node can be found, e.g., when we are at the last sibling or a node with no children.

EXAMPLE 9. *Find the first two elder siblings of 'Mary'.*

QUERY 10.
```
return $X@Cname, $Y@Cname
 from  //daughter[@Cname='Mary'] \ $X \ $Y
```

EXAMPLE 11. *Find the second child of 'Mary'.*

QUERY 12.
```
return $Y@Cname
 from  //daughter[@Cname='Mary'] / \ $X \ $Y
```

Partition By. Inspired by relational Data Stream Management Systems (DSMS), XSeq supports a partitioning operator that is very essential for many CEP applications. Nodes can be partitioned by their key, so that different groups can be processed in parallel as the XML stream arrives. 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 path expression in the partitioning clause be 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.

EXAMPLE 13. *For each city, find the oldest person born there.*

By knowing the cities that are present in our XML, we could write several queries, one for each city e.g. $\mathtt{max}(//\mathtt{son}[\mathtt{Bplace} = 'LA'])$. However, in streaming applications such information is generally not provided a priori. Moreover, instead of running several queries over the same stream, an explicit partition by clause allows for simultaneous handling of different key values and is much easier to optimize. For instance:

QUERY 14.
```
return $X@Bplace, min($X@Bdate)
 from  //$X
 partition by //X@Bplace
```

[4]This is an example of a well-known class of XML queries which has been proven [31] as not expressible in Core XPath 1.0.

[5]When a WINDOW is defined over the XML stream, the input nodes can be re-ordered. For simplicity of the discussion, we do not discuss re-ordering.

[6]XSeq also supports sequential aggregates on compound Kleene-*, e.g. $(/\mathtt{son}/\$\mathtt{X})^*$. The full syntax and semantics are in [23].

```
<! DOCTYPE stocks [
<! ELEMENT stocks (transaction*)>
<! ATTLIST transaction company CDATA #REQUIRED> ]
<! ATTLIST transaction price CDATA #REQUIRED>
<! ATTLIST transaction buyer IDREF #REQUIRED>
<! ATTLIST transaction date CDATA #REQUIRED>>
```

Figure 3: The DTD for the stream of Nasdaq transactions.

If the user desires an XML output, he can embed the XSeq query in an XQuery or XSLT expression. [7] Here, we only covered the basic constructs of XSeq that are needed in the paper. More details on the syntax is provided in Appendix A. Next, we will use these basic constructs to express more advanced queries from a wide range of CEP applications.

3. ADVANCED QUERIES FROM COMPLEX EVENT PROCESSING

In this section we present more complex examples from several domains and show that XSeq can easily express such queries.

Stock Analysis. The 'V'-shape pattern is a well-known query in stock analysis. Consider an XML stream of stock quotes as defined in Figure 3.

EXAMPLE 15 (**'V'-shape pattern**). *Find those stocks whose prices have formed a 'V'-shape. That is, the price has been going down to a local minimum, then rising up to a local maximum which was higher than the starting price.*

The 'V'-shape query only exemplifies many important queries from stock analysis [8] that are provably impossible to express in Core XPath 1.0 and Regular XPath, simply both of these languages lack the notion of 'immediately following sibling' in their constructs. XPath 2.0, however, can express these queries through the use of its `for` and quantified variables: using these constructs, XPath 2.0 can 'simulate' the concept of 'immediately following sibling' in XPath 2.0 by double negation, i.e. ensuring that 'for each pair of nodes, there is nothing in between'. But this approach leads to very convoluted XPath expressions which are extremely hard to write/understand and almost impossible to optimize (See 1 and Section 6).

On the other hand, XSeq can express this queries with its simple constructs that can be easily translated and optimized as VPA:

QUERY 16 ('V'-PATTERN IN XSEQ).
```
return last($Y)@price
from /stocks/Z*(\ $X)*(\ $Y)*
where tag(Z) = 'transaction'
  and tag(X) = 'transaction' and tag(Y) = 'transaction'
  and X@price < prev(X)@price and Y@price < prev(Y)@price
partition by /stocks/transaction@company
```

Social Networks. Twitter provides an API[9] to automatically receive the stream of new tweets in several formats, including XML. Assume the tweets are ordered according to their date timestamp:

```
<! DOCTYPE twitter [
<! ELEMENT twitter ((tweet)*)>
<! ELEMENT tweet (message)>
<! ELEMENT message (#PCDATA)>
<! ATTLIST tweet tweetid CDATA #REQUIRED>
<! ATTLIST tweet userid CDATA #REQUIRED>
<! ATTLIST tweet date CDATA #REQUIRED> ]>
```

[7] Formatting the output is out of the scope of this paper. Instead, we only focus on the query expression and its efficient execution for CEP applications.

[8] http://www.chartpattern.com/

[9] http://dev.twitter.com/

EXAMPLE 17 (DETECTING ACTIVE USERS). *In a stream of tweets, report users who have been active over a month. A user is active if he posts at least a tweet every two days.*

This query, if not impossible, would be very difficult to express in XPath 2.0 or Regular XPath. The main reason is that, again due to their lack of 'immediate following', they cannot easily express the concept of "adjacent" tweets.

QUERY 18 (DETECTING ACTIVE USERS IN XSEQ).
```
return first(T)@userid
from /twitter/ Z* (\ $T)*
where tag(Z) = 'tweet' and tag(T) = 'tweet'
  and T@date-prev(T)@date < 2
  and last(T)@date-first(T)@date > 30
partition by /twitter/tweet@userid
```

Inventory Management. RFID has become a popular technology to track inventory as it arrives and leaves retail stores. Below is a sample schema of events, where events are ordered by their timestamp:

```
<! DOCTYPE events [
<! ELEMENT events (event*)>
<! ELEMENT event (message)>
<! ELEMENT message (#PCDATA)>
<! ATTLIST event ts CDATA #REQUIRED>
<! ATTLIST event itemid CDATA #REQUIRED>
<! ATTLIST event eventtype CDATA #REQUIRED> ]>
```

EXAMPLE 19 (DETECTING ITEM THEFT). *Detect when an item is removed from the shelf and then removed from the store without being paid for at a register.*

QUERY 20 (DETECTING ITEM THEFT IN XSEQ).
```
return first(R).T@itemid
from /events/$T \ $W* \ $X
where tag(T) = 'event' and tag(W) = 'event' and tag(X) = 'transaction'
  and T@eventtype = 'removed from shelf'
  and X@eventtype = 'removed from store'
  and W@eventtype != 'paid at register'
partition by /events/event@itemid
```

Directory Search. Consider the following first-order binary relation which is familiar from temporal logic [33]:

$$\phi(x,y) = \text{descendant}(x,y) \wedge q(y) \wedge \forall z(\text{descendant}(x,z) \wedge \text{descendant}(z,y) \rightarrow p(z))$$

For instance, for a directory structure that is represented as XML, by defining q and p predicates as $q(y)$: 'y is a file' and $p(z)$: 'z is a non-hidden folder', the ϕ relation becomes equivalent to the following query:

EXAMPLE 21. *Retrieve all reachable files from the current folder by repeatedly selecting non-hidden subfolders.*

According to the results from [33], such queries are not expressible in XPath 1.0. This query, however, is expressible in XPath 2.0 but not very efficiently. E.g.,

`//file except //folder[@hidden='true']//file`

Such queries can be expressed much more elegantly in XSeq (and also in Regular XPath):

QUERY 22 (ϕ QUERY IN XSEQ).
```
(/folder[@hidden = 'false'])*/file
```

Genetics. Haemophilia is one of the most common recessive X-chromosome disorders. In genetic testing and counseling, if the fetus has inherited the gene from an affected grandparent the risk to the fetus is 50% [1]. Therefore, the inheritance risk for a person can be estimated by tracing the history of haemophilia among its even-distance ancestors, i.e. its grandparents, its grand-parents' grand-parents, and so on.

EXAMPLE 23. *Given an ancestry XML which contains the history of haemophilia in the family, identify all family members who are at even-distance from an affected member, and hence, at risk.*

This query cannot be easily expressed without Kleene-* [8], but is expressible in XSeq:

QUERY 24 (DESCENDANTS OF EVEN-DISTANCE FROM A NODE).
```
return $Z@Cname
from //$X[@haemophilia = 'true'] (/$Y/$Z)*
```

Queries 22 and 24 are not expressible in XPath 1.0, are expressible in XPath 2.0 but not efficiently, and are easily expressible in Regular XPath and XSeq.

Temporal Queries.

Expressing temporal queries represents a long-standing research interest. A number of language extentions and ad-hoc solutions have been proposed. However, XSeq is able to express a large range of temporal queries. We will take the famous temporal aggregate named RISING, introduced by TSQL2 [28], as an example. Below is the DTD of a temporal employee XML. Each employee record is ordered by the start time of the record (tstart):

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

EXAMPLE 25 (RISING). *What is the maximum time range during which the salary of John is rising?*

QUERY 26.
```
return max(last(X)@tend-first(X)@tstart)
from //employee/Z*(\ $X)*
where tag(Z) = 'salary' and tag(X) = 'salary'
  and X/salary/text() > prev(X)/salary/text()
  and X@tstart <= prev(X)@tend
partition by /employees/employee@id
```

4. XSEQ OPTIMIZATION

The choice of operators in XSeq is heavily influenced by whether they can be efficiently evaluated or not. Our criterion for efficiency of an XSeq operator is whether it can be mapped to a Visibly Pushdown Automaton (VPA). The rationale behind choosing VPA as the underlying query execution model is two-fold. First, XSeq is mainly designed for complex patterns and patterns can be intuitively described as transitions in an automaton: fortunately, VPAs are expressive enough to capture all the complex patterns that can be expressed in XSeq. Secondly, VPAs retain many attractive computational properties of finite state automata on words [2]. In fact, by translation into VPAs, we can exploit several existing algorithms for streaming evaluation [19] and optimization of VPAs [21]. For unfamiliar readers, we have provided a brief background on VPAs in Appendix B.

In Section 4.1, we describe a simplified version of our translation from XSeq queries into equivalent VPAs[10] which can faithfully capture the same pattern in the input. Then, in Sections 4.2 and 4.3, we present several static (compile-time) and run-time optimizations of VPAs in our XSeq implementation. In Section 6 we study the effectiveness of these optimizations in practice.

[10]In this paper, we do not formally define 'equivalence'. Informally, when an XSeq query and a VPA are equivalent, every portion of the input XML that produces an output result in the fomer, will be accepted by the latter and vice versa.

4.1 Efficient Query Plans via VPA

As described above, compiling XSeq queries into efficient query plans starts by constructing an equivalent VPA for the given query. We construct this VPA by an iterative bottom-up process where we start from a single-state (trivial) VPA and at each Step of the XSeq query, we compose the original VPA with a new VPA that is equivalent with the current Step. Next, we show how different axes can be mapped into equivalent VPAs. Lastly, we show how other constructs of the XSeq query can be handled as well.

In the following, whenever connecting the accepting state(s) of a VPA to the starting state(s) of the previous VPA, note that VPAs are closed under concatenation, and thus, the resulting automaton is still a valid VPA.

Handling /: The /X axis is equivalent to a VPA with two states E and O where E is the starting state where we invoke the stack on open and closed tags accordingly (see Appendix B for the rules regarding stack manipulation in a VPA), and transition to the same state on all input symbols as long as the consumed input in E is well-nested. Upon seeing the appropriate open tag (e.g., $\langle$X$\rangle$) we non-deterministically transition to our accepting state O.

Handling @: In the presence of the attribute specifier, @, we add a new state A as the new accepting state which will be transitioned to from our previous accepting state upon seeing any attribute. We remain in state A as long as the input is another attribute, i.e. to account for multiple attributes of the same open tag.

Figure 4(a) demonstrates the VPA for /son@Bdate. Figure 5 shows the intuitive correspondence of this VPA with the navigation of the XML document, where:

- E matches zero or more (well-nested) subtrees in the pre-order traversal of the XML tree,
- O matches the open tag for son, i.e. $\langle$son$\rangle$,
- A matches the attribute list of $\langle$son$\rangle$, namely O.

To see the correspondence between this VPA and the XSeq query, note that to find all the direct sons of a daughter, we navigate through the pre-order traversal of the sub-tree under each daughter node, then *non-deterministically* skip an arbitrary number of her children (i.e., $\mathtt{E}^*$) until visiting one of her children who is a son (i.e., O), and then finally visit all the tokens that correspond to his son's attributes, i.e. $\mathtt{A}^*$. The non-determinism assures that we eventually visit all the sons under each daughter.

Handling ()*: Kleene-* expressions in XSeq, such as $(/\mathtt{son})^*$, are handled by first constructing a VPA for the part inside the parentheses, say V_1, then adding an ϵ-transition from the accepting state of V_1 back to its starting state. Since VPAs are closed under Kleene-*, the resulting automaton will still be a VPA.

Handling //: The // axis can also be easily defined as a Kleene-* of the / operator. For instance, the //daughter construct is equivalent to $(/\mathtt{X})*/\mathtt{daughter}$, where X is a wild card, i.e. matches any open tag. Figure 5 shows the correspondence between the VPA states for // and the familiar traversal of the XML document.

Handling siblings: Let V_1 be the VPA that recognizes the query up to node D. The VPA for recognizing the sibling of D, say node S, is constructed by adding four new states (E1, C, E2 and O) to V_1, where:

- We transition from the accepting state(s) of V_1 to E1. E1 invokes the stack on open and closed tags accordingly, and transitions to itself on all input symbols as long as the consumed input in E1 is well-nested.

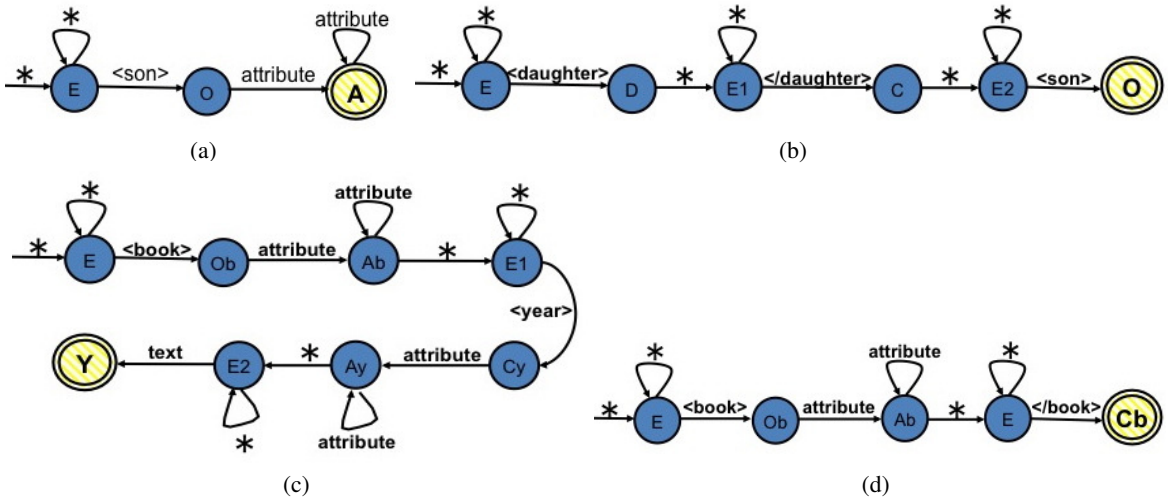


Figure 4: VPAs for (a) /son@Bdate**, (b)** /daughter son**, (c)** //book[year = 2000]**, and (d)** //book[@title = 'mytitle']

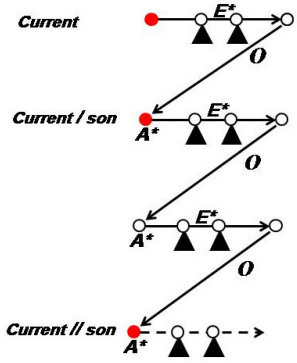


Figure 5: Visual correspondence of VPA states and XSeq axes.

- Upon seeing a close tag of D, we non-deterministically transition from E1 to C.
- We transition from C to E2 upon any input. Similar to E1, E2 invokes the stack on open and closed tags accordingly, and transitions to itself on all input symbols as long as the consumed input in E2 is well-nested.
- Upon seeing an open tag for the sibling, i.e. ⟨S⟩, we non-deterministically transition from E2 to state O which is marked as the accepting state of the new VPA.

Figure 4(b) shows the VPA for query "/daughter son". The intuition behind this construction is that E1 skips all possible subtrees of the last daughter non-deterministically, while E2 non-deterministically skips all other siblings of the current daughter until it reaches its sibling of type son.

Handling \ : The construct \X is handled according to the last axis that has appeared before it. Let V_1 be the VPA for the XSeq query up to \X. When the previous axis is vertical (e.g. / or //), then we only need to add one new state to the V_1, say O, where from all the accepting states of V_1 we transition to state O upon seeing any open tag of X. The new accepting state will be O.

When the axis before \X is horizontal (e.g. siblings), we add three new states to V_1, say E, C and O, where:

- We transition from the accepting state(s) of V_1 to E. At E, we invoke the stack upon open and closed tags accordingly, and transition to E on all input symbols as long as the consumed input in E is well-nested.
- We non-deterministically transition from E to C upon seeing a close tag of the last (horizontal) axis.
- We transition from C to O upon an open tag for X and fail otherwise. O will be the new accepting state of the VPA.

Handling backward axes. Backward axes are translated by using one variable per axis but expressing all (exponential) combinations in which they appear as predicates (handling predicates is described next). For instance, an XSeq query //son[@Cname = 'John']///son[@Cname = 'Bob']/ancestor::son[@$Cname$ = '$Alex$'] is equivalent to a VPA with three states X, Y and Z (as well as intermediary states to capture the well-nestedness, similar to forward axes above) with the condition that the name of Y is 'Bob' and either the name of X is 'John' and the name of Z is 'Alex' or X is 'Alex' and Z is 'John', i.e. the different orders in which they could appear in the pre-order traversal of the XML tree.

Handling predicates. Comparisons between the values of different nodes are deferred to the first state where both values have been seen. This requires that we have access to the value of a node even we are not at its corresponding state. However, previous input symbols in a VPA can only be remembered in one of the two ways. 1) Retrieving the top symbol on the stack. However, this operation in a VPA is only allowed when the current input is a close tag, and 2) Encoding a finite amount of history in the state itself, i.e. every state represents one out of finite number of cases in the past.

In our real implementation of XSeq, we simply use a few variables (a.k.a. registers) at each state, in order to remember the latest values of the operands in the predicate(s) that need to be evaluated at that state. However, in our complexity analysis in Section 5, we use the abstract form of a VPA, namely where a state is duplicated as many as there are unique values for its operands.

Handling Partition By Since the pattern in the 'partition by' clause is the prefix of the pattern in the 'from' clause, the partition by clause can be simply treated as a new predicate on the attribute which is partitioned by. For example, when translating Query 26 into a VPA, let $A_1, \cdots, A_n$ be all the states that represent the attribute(s) of any employee node. Then, we can implement the partition by clause by simply checking at every state A_i that the current value of the ID attribute is equal to the last value of the ID attribute seen at A_j state, for all $1 \leq j \leq n$.

Handling other constructs Union, intersection (equivalently, node tests) and negation can all be implemented with their corresponding

operations on the intermediary VPAs, as VPAs are closed under union, intersection and complementation. The translations are thus straightforward (omitted here for space constraints).

4.2 Static VPA Optimization

Cutting the inferrable prefix. When the schema (e.g. DTD) is available, 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 always inferred for the remaining suffix. For example, consider the following XSeq query, defined over the SigmodRecord dataset[11]:
`//issue/articles/authors/author[text()='Alan Turing']`
This XSeq query generates a VPA with many states, i.e. 3 states for every step. However, based on the DTD, we infer that `author` nodes always have the same prefix, i.e. `issue/articles/authors/`. Thus, we remove the part of the VPA that corresponds to this common prefix. Due to the sequential nature of VPAs, such simplifications can greatly improve the efficiency by reducing a global pattern search to a more local one.

Reducing non-determinism from the transition table. Our algorithm for translating XSeq queries produces VPAs that are typically non-deterministic. Reducing the degree of non-determinism always improves the execution efficiency by avoiding many unnecessary backtracks. In general, *full determinization* of a VPA is an expensive process, which can increase the number of states from $O(n)$ to $O(2^{n^2})$ [2].

However, there are special cases that the degree of non-determinism can be reduced without incurring an exponential cost in memory. Since self-loops in the transition table are the main source of non-determinism, the XSeq's compile-time optimizer removes such edges from the generated VPA, whenever possible. For instance, consider the XSeq query $//book[year = 2000]$ and its corresponding VPA in Figure 4(c). If according to the schema, `year` nodes cannot contain any subelements, the optimizer will remove the self-loop from `E2`'s transition table (we remove `E2` entirely, if it lacks any other transitions). Also, if a node, say `year`, does not have any attributes, the optimizer will remove its corresponding state, here Ay.

Finally, whenever self-loops can only occur a fixed number of times, they are removed by duplicating their corresponding states accordingly. For instance, if we know that `book` nodes only contain two subelements, say `title` followed by `year`, the optimizer replaces `E1` with 3 new states (without any self-loop) to explicitly skip the title's open, text and closed tags. The latter expression $(E1\hat{}3)$ is executed more efficiently as it will be deterministic.

Reducing non-determinism from the states. In order to skip all the intermediate subelements, the automatically generated VPAs contain several states with incoming and outgoing ϵ-transitions. In the presence of the XML schema, many of such states become unnecessary and can be safely removed before evaluating the VPA on the input. We have several rules for such safe omissions. Here, we only provide one example.

Consider the VPA in Figure 4(d) where the states `Ob`, `Ab` and `Cb` match with ⟨`book`⟩, its attributes and ⟨/`book`⟩, respectively. If we know that `book` nodes cannot contain another book, we can remove the state `E`.

4.3 Run-time VPA Optimization

In the previous sections, we demonstrated how XSeq queries can be translated into equivalent VPAs and presented several techniques for reducing the degree of non-determinism in our VPAs. One of the main advantages of using VPAs as the underlying execution model is that we can take advantage of the rich literature on efficient evaluation of VPAs. In particular we use the one-pass evaluation of the VPAs as described in [19] and use the pattern matching optimization of VPAs as described in [21].

In a straightforward evaluation of a VPA over a data stream, one would consider the prefix starting from every element of the stream as a new input to the VPA. In other words, upon acceptance or rejection of every input, the immediate next starting position would be considered. However, for word automata, it is well-known that this naive backtracking strategy can be easily avoided by applying pattern matching techniques such as the KMP [15] algorithm. Recently, a similar pattern matching technique was developed for VPAs, known as VPSearch [21]. Similar to word automata, VPSearch avoids many unnecessary backtracks and therefore, reduces the number of VPA evaluations. We have implemented VPSearch and its run-time caching techniques in our Java implementation of XSeq. Further details on streaming evaluation of VPAs and the VPSearch algorithm can be found in [19] and [21], respectively. Because of the excellent VPA execution performance achieved by K*SQL [21], we have used the same run-time engine for XSeq queries once they are compiled into a VPA (see Section 7).

5. EXPRESSIVENESS AND COMPLEXITY

Our main focus in this paper is to introduce XSeq through intuitive examples from important complex event processing domains. We have also provided the high-level idea of how XSeq queries can be optimized and translated into equivalent VPAs. For space limitations, we leave the formal treatment of XSeq to our technical report [23], including the formal semantics and rigorous details of the translation into VPAs. Therefore, in this section we briefly summarize our results on the expressiveness of XSeq, and its complexity for query evaluation and query containment—three fundamental questions for any query language.

The full language of XSeq is too rich for a rigorous logical analysis, and thus we focus on its navigational features by excluding arithmetics, string manipulations and aggregates. To allow for memory-efficient streaming algorithms we also disallow $\neq$ operator in our analysis. Thus, we obtain a more concise language, called Core XSeq[12].

In the following, Σ is the alphabet (i.e., set of unique tokens in the XML document), FO is the first order logic, FO* is the extension of FO with a transitive closure operator that applies to formulas with exactly two free variables, FO(MTC) is first-order logic extended with the monadic transitive closure operator [34], and MSO$_\mu$ is monadic second order logic over words augmented with a binary matching relation μ [2].

THEOREM 1 (EXPRESSIVENESS). *Core XSeq $\equiv$ MSO$_\mu$:*

1. *For every query in Core XSeq of size $O(m)$ there is an equivalent VPA with $O(m^2 \cdot |\Sigma|^{m^2} \cdot 2^m)$ states.*
2. *There are linear-time encodings of Visibly Pushdown Expressions (VPE) into Core XSeq queries.*

PROOF SKETCH. In Section 4.1, we have provided the linear-time mapping from XSeq to VPA with predicates. To get rid of the predicates, we need to replicate each state for every value that it needs to remember for evaluating those predicates. We have $O(m)$ predicates, for each we need to remember at most $|\Sigma|^2$ different

[11] `http://www.cs.washington.edu/research/xmldatasets/`

[12] Similar approaches in analyzing XPath 1.0 and 2.0, has led to sub-languages Core XPath 1.0[33] and Core XPath 2.0[32].

values, by remembering the min or max of each operand (e.g. when the predicate is $T@price < S@price$ we only need to remember the max of the first and the min of the second operand). A careful case-analysis leads to $O(m^2|\Sigma|^{m^2}2^m)$ states in the VPA without any predicates.

To prove that all MSO$_\mu$ formulas can be expressed in XSeq, we encode the VPEs [26] as XSeq queries, similar to the encoding used in [21], except that in XSeq, since negation in the path expression is not allowed, we negate the predicates. (Visibly Pushdown Expressions (VPE) [26] are generalizations of regular expressions that are equivalent to Visibly Pushdown Languages (VPLs) and thus, to VPAs and MSO$_\mu$ (MSO over nested words) [2].) □

The expressiveness of the previous languages are as follows:

Core XPath 1.0 $\subsetneq$FO$\equiv$ Core XPath 2.0 $\equiv$ Conditional XPath $\subsetneq$ Regular XPath $\subsetneq_?$ FO* $\equiv$ Regular XPath$^{\approx}$ $\subsetneq_?$ FO(MTC) $\equiv$ Regular XPath(W) $\subsetneq$ μRegular XPath$\equiv$Core K*SQL$\equiv$MSO$\equiv$Core XSeq

(for proofs, see their respective papers).

Thus, for every query written in any of the languages above there exists an equivalent Core XSeq query, and except for μRegular XPath and Core K*SQL, Core XSeq is strictly more expressive than the rest.

LEMMA 2 (QUERY EVALUATION). *Data and query complexities for Core XSeq's query evaluation are* PTIME *and* EXPTIME*, respectively.*

PROOF SKETCH. By mapping Core XSeq queries into VPAs, the query evaluation of the former corresponds to the language membership decision of the latter. Using the membership algorithm provided in [19], we only need space O($s^4 \cdot log\ s \cdot d + s^4 \cdot n \cdot log\ n$) where n is the length of the input, d is the depth of the XML document (thus, $d < n$), and s is the number of the states in the VPA. PTIME data complexity comes from n and the EXPTIME query complexity comes from s which is exponential in the query size (see Theorem 1). □

LEMMA 3 (QUERY CONTAINMENT). *Query containment for Core XSeq is decidable and is* 2-EXPTIME*-complete.*

PROOF SKETCH. Once two Core XSeq queries are translated into VPAs, their query containment problem corresponds to the language inclusion problem for their VPAs, say M_1 and M_2. To check $L(M_1) \subseteq L(M_2)$, we check if $L(M_1) \cap \overline{L(M_2)} = \emptyset$. Given M_1 with s_1 states and M_2 with s_2 states, we can determinize [30] and complement the latter to get a VPA for $\overline{L(M_2)}$ of size O($2^{{s_2}^2}$). $L(M_1) \cap \overline{L(M_2)}$ is then of size O($s_1 \cdot 2^{{s_2}^2}$), and emptiness check is polynomial (cubic) in the size of this automaton. Since, s_1 and s_2 are themselves exponential in the size of their Core XSeq queries, membership in 2-EXPTIME holds. For completeness of the 2-EXPTIME, note that XSeq syntactically subsumes Regular XPath$(*, \cap)$ for which the query containment has been shown to be 2-EXPTIME-complete [8]. □

6. EXPERIMENTS

In this section we study the amenability of XSeq language to efficient execution. Our implementation of the XSeq language consists of a parser, VPA generator, a compile-time optimizer, and the VPA evaluation and optimization run-time, all coded in Java. We first evaluate the effectiveness of our different compile-time optimization heuristics in isolation. We then compare our XSeq system with the state-of-the-art XML engines for (i) complex sequence queries, (ii) Regular XPath queries, and (iii) simple XPath queries. While these systems are designed for general XML applications, we show that XSeq is far more suited for CEP applications. In fact, XSeq achieves up to two orders of magnitude out-performance on (i) and (ii), and competitive performance on (iii). Finally, we study the overall performance, throughput and memory usage of our system under different classes of patterns and queries.

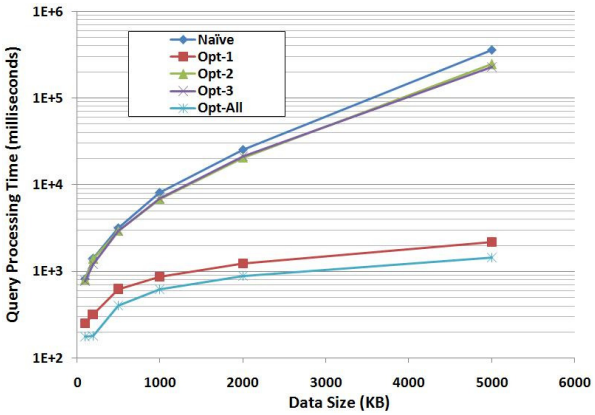


Figure 6: Contribution of different optimization techniques.

All the experiments were conducted on a 1.6GHz Intel Quad-Core Xeon E5310 Processor running Ubuntu 6.06, with 4GB of RAM. We have used several real-world datasets including NASDAQ stocks that contains more than $7.6M$ records[13] since 1970, and also the Treebank dataset[14] that contains English sentences from Wall Street Journal and has with a deep recursive structure (max-depth of 36 and avg-depth of 8). We have also used XMark [27] which is well-known benchmark for XML systems and provides both data and queries. Due to lack of space, for each experiment we only report the results on one dataset. The results and main observations, however, were similar across different datasets.

6.1 Effectiveness of Different Optimizations

In this section, we evaluate the effectiveness of the different compile-time optimizations from Section 4.2, by measuring their individual contribution to the overall performance[15]. For this purpose, we executed the X2 query from XMark [27] over a wide range of input sizes (generated by XMark, from 50KB to 5MB). The results of this experiment are reported in Figure 6, where we use the following acronyms to refer to different optimization heuristics (see Section 4.2):

Opt-1	Cutting the inferrable prefix
Opt-2	Reducing non-determinism from the pattern clause
Opt-3	Reducing non-determinism from the where clause

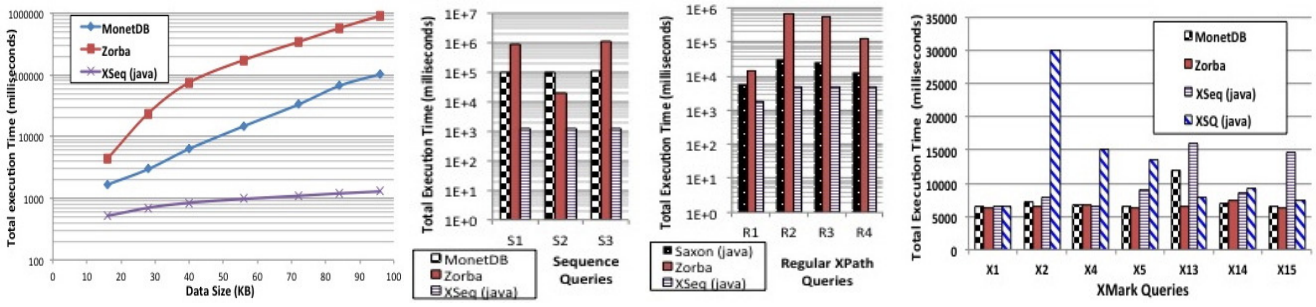
In this graph, we have also included the naive and combined (Opt-All) versions, namely when, respectively, none and all of the compile-time optimizations are applied. The first observation is that combining all the optimization techniques delivers a dramatic improvement in performance (1-2 orders of magnitude, over the naive one).

Cutting the inferable prefix, Opt-1, leads to fewer states in the final VPA. Like other types of automata, fewer states can significantly reduce the overall degree of non-determinism. The second

[13] http://infochimps.org/dataset/stocks_yahoo_NASDAQ

[14] http://www.cs.washington.edu/research/xmldatasets/www/repository.html

[15] The effectiveness of the VPA evaluation and optimization techniques have been previously validated in their respective papers [19, 21].



(a) (b) (c) (d)

Figure 7: XSeq vs. XPath/XQuery engines: (a) 'V'-pattern query over Nasdaq stocks, (b) Sequence queries over Nasdaq stocks, (c) Regular XPath queries over XMark data, and (d) conventional XPath queries from XMark.

reason behind the key role of Opt-1 in the overall performance is that it reduces non-determinism from the *beginning* of the pattern: this is particularly important because non-determinism in the starting states of a VPA is usually disastrous as it prevents the VPA from the early detection of unpromising traces of the input. In contrary, reducing non-determinism in the pattern and the `where` clause (Opt-2, Opt-3) has a much more local effect. In other words, the latter techniques only remove the non-determinism from a single state or edge in the automata, while the rest of the automata may still suffer from non-determinism. However local, Opt-2 and Opt-3 can still improve the overall performance when combined with Opt-1. This is because of the extra information that they learn from the DTD file.

6.2 Sequence Queries vs. XPath Engines

We compare our system against two[16] of the fastest academic and industrial engines: MonetDB/XQuery[6] and Zorba [4]. First, we used several sequence queries on Nasdaq transactions (embedded in XML tags), including the 'V'-shape pattern (defined in Example 15 and Query 16). By searching for half of a 'V' pattern, we defined another query to find 'decreasing stocks'. Also, by defining two occurrences of a 'V' pattern, we defined what is known as the 'W'-shape pattern [17]. We refer to these queries as S1, S2 and S3. We also defined several Regular XPath queries over the treebank dataset, named R1, R2, R3 and R4 where,

R1: /FILE/EMPTY(/VP)*/NP,
R2: /FILE(/EMPTY)*/S,
R3: /FILE(/EMPTY)*(/S)*/VP,
R4: /FILE(/EMPTY)*/S(/VP)*/NP

Sequence queries. For expressing these queries (namely S1, S2 and S3) in XQuery, we had to mimic the notion of 'immediately following sibling', i.e. by checking that for each pair of siblings in the sequence, there are no other nodes in between. The XQuery versions of S2 has been given in Figure 1. Due to the similarity of S1 and S3 to S2 here we omit their XQuery version (roughly speaking, S1 and S3 consist of, respectively, two and four repetitions of S2).

Not only were sequence queries difficult to express in XPath/ XQuery but were also extremely inefficient to run. For instance, for the queries at hand, neither of Zorba or MonetDB could handle any input data larger than 7KB. The processing times of these sequence queries, over an input size of 7KB, are reported in Figure 7(b). Note that the Y-axis is in log-scale: *the same sequence queries written in XSeq run between 1-3 orders of magnitude faster than their XPath/XQuery counterparts do on two of the fastest XML engines.* Figure 7(a) shows that gap between XSeq and the other two engines grows with the input size. This is due to the linear-time query processing of XSeq which, in turn, is due to the linear-time algorithm for evaluation of VPAs along with the backtracking optimizations when the VPA rejects an input [21]. Zorba and MonetDB's processing time for these sequence queries are at least quadratic, due to the nested nature of the queries.

In summary, the optimized XSeq queries run significantly (1-3 orders of magnitude) faster than their equivalent counterparts that are expressed in XQuery. This result indicates that traditional XML languages such as XPath and XQuery (although theoretically expressive enough), due to their lack of explicit constructs for sequencing, are not amenable to effective optimization of complex queries that involve repetition, sequencing, Kleene-*, etc.

Regular XPath queries. As mentioned in Section 1, despite the many benefits and applications of Regular XPath, currently there are no implementations for this language (to our best knowledge). One of the advantages of XSeq is that it can be also seen as the first implementation of Regular XPath, as the latter is a subset of the former. In order to study the performance of XSeq for Regular XPath queries (e.g., R1, $\cdots$, R4) we compared our system with the only other alternative, namely implementing the Kleene-* operator as a higher-order user-defined functions (UDF) in XQuery. Since MonetDB does not support such UDFs, we used another engine, namely Saxon [14]. The results for 464KB of treebank dataset are presented in Figure 7(c) as Zorba, again, could not handle larger input size. Thus, for Regular XPath queries, similarly to sequence queries, XSeq proves to be 1-2 orders of magnitude faster than Zorba, and between 2-6 times faster than Saxon. Also, note that the relative advantage of Saxon over Zorba is only due to the fact that Saxon loads the entire input file in memory and then performs an in-memory processing of the query [14]. However, this approach is not feasible for streaming or large XML documents[18].

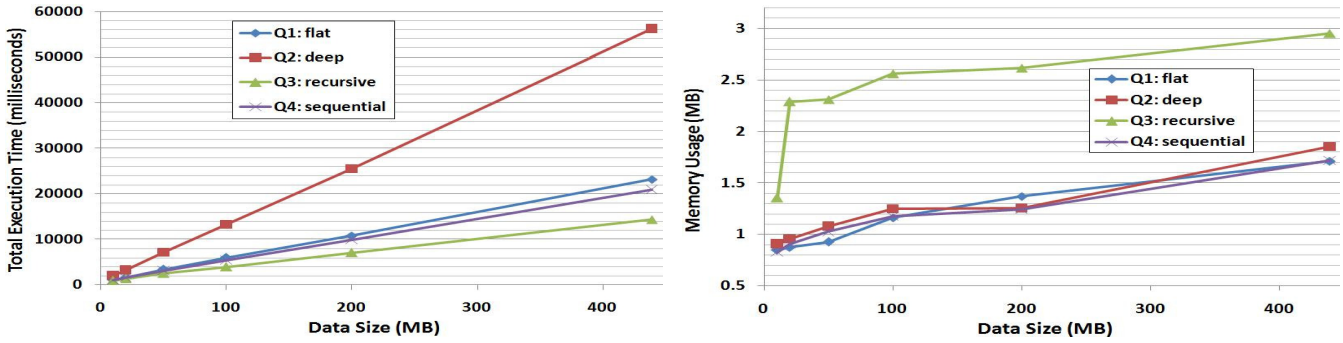
6.3 Conventional Queries vs. XPath Engines

As shown in the previous section, complex sequence queries written in XSeq can be executed dramatically faster (from 0.5 to 3 orders of magnitude) than even the fastest of XPath/ XQuery engines. In this section, we continue our comparison of XSeq and native XPath engines by considering simpler XPath queries, i.e. queries without sequencing and Kleene-*. For this purpose, we used the XMark queries which in Figure 7(d) are referred to as X1,

[16] Since the sequence queries of this experiment are not expressible in XPath, we could not use the XSQ [25] engine as it does not supports XQuery.

[17] 'W'-pattern (a.k.a. double-bottom) is a well-known query in stock analysis.

[18] Due to lack of space, we omit the results for the case when the input size cannot fit in the memory. Briefly, unlike XSeq, Saxon results in using the disk swap, and thus, suffers from a poor performance.



(a) (b)

Figure 8: Effect of different types of XSeq queries on total execution time (a) and memory usage (b).

X2, and so on[19]. Once again, we executed these queries on MonetDB, Zorba (as state-of-the-art XPath/XQuery engines) and XSQ (as state-of-the-art streaming XPath engine) as well as on our XSeq engine. In this experiment, the XMark data size was 57MB. Note that both Zorba and MonetDB are implemented in C/C++ while XSeq is coded in Java, which generally accounts for an overhead factor of 2X in a fair comparison with C/C++ implementations. The results are summarized in Figure 7(d). The XSeq queries were consistently competitive compared to all the three state-of-the-art XPath/XQuery engines. XSeq is faster than XSQ for most of the tested queries. For some queries, e.g. X2 and X4, XSeq is even 2-4 times faster. Even compared with MonetDB and Zorba, XSeq is giving surprisingly competitive performance, and for some queries, e.g. X4, were even faster. Given that XSeq is coded in Java, this is an outstanding result for XSeq. For instance, once the java factor is taken into account, the only XMark query that runs slower on the XSeq engine is X15, while the rest of the queries will be considered about 2X faster than both MonetDB and Zorba.

In summary, once the maturity of the research on XPath/ XQuery optimization is taken into account, our natural extension of XPath that relies on a simple VPA-based optimization seems very promising: XSeq achieves better or comparable performance on simple queries, and is dramatically faster for more involved queries.

6.4 Throughput for Different Types of Queries

To study the performance of different types of queries in XSeq, we selected four representative queries with different characteristics which, based on our experiments, covered a wide range of different classes of XML queries. To facilitate the discussion, below we label the XML patterns as 'flat', 'deep', 'recursive' and 'monotone':

Q1: flat	/site/people/person[@id = 'person0']/name/text()
Q2: deep	/site/closed_auctions/closed_auction/annotation/ description/parlist/listitem/parlist/listitem/text/ emph/keyword/text()
Q3: recursive	(parlist/listitem)*
Q4: monotonic	//closed_auctions/ (\X[tag(X)='closed_auction' and X@price < prev(X)@price])*

We executed all these queries on XMark's dataset. Also, the first two queries (Q1 and Q2) are directly from XMark benchmark (referred to as Q1 and Q15 in [27]). We refer to them as 'flat' and 'deep' queries, respectively, due to their few and many axes. In XMark's dataset, the `parlist` and `listitem` nodes can contain one another, which when combined with the Kleene-*, is the reason why we have named Q3 'recursive'. The Q4 query, called 'monotonic', searches for all sequences of consecutive closed auctions where the price is strictly decreasing. These queries reveal interesting facts about the nature of XSeq language and provide insight on the types of XSeq queries that are more amenable to efficient execution under the VPA optimizations.

The query processing time is reported in Figure 8(a). The first important observation is that XSeq has allowed for linear scalability in terms of processing time, regardless of the query type. This has enabled our XSeq engine to steadily maintain an impressive throughput of 200,000-700,000 tuples/sec, or equivalently, 8-31 MB/sec even when facing an input size of 450MB. This is shown in Figures 9(a) and 9(b) in which the X-axes are drawn in log-scale. Interestingly, the throughput gradually improves when the window size grows from 200K to 1.1M tuples. This is mainly due to the amortized cost of VPA construction and compilation, and other run-time optimizations such as backtrack matrices [21] that need to be calculated only once.

Among these queries, the best performance is delivered for Q3 and Q4. This is because they consist of only two XPath steps, and therefore, once translated into VPA, result in fewer states. Q1 comes next, as it contains more steps and thus, a longer pattern clause. Q2 achieves the worst performance. This is again expected, because Q2's deep structure contains many tag names which lead to more states in the final VPA. In summary, this experiment shows that with the help of the compile-time and run-time optimizations, XSeq queries enjoy a linear-time processing. Moreover, the fewer axes (i.e. steps) involved in the query, the better the performance.

7. PREVIOUS WORK

XML Engines. Given the large amount of previous work on supporting XPath/XQuery on stored and streaming data, we only provide a short and incomplete overview, focusing on the streaming ones. Several XPath streaming engines have been proposed over the years, including TwigM [9], XSQ [25], and SPEX [24]; also the processing of regular expressions, which are similar to the XPath queries of XSQ, is discussed in [24] and [5]. XAOS [5] is an XPath processor for XML streams that also supports reverse axes (parent and ancestor), while support for predicates and wildcards is discussed in [13]. Finally, support for XQuery queries on very small XML messages (<100KB) is discussed in [11].

Language extensions. Extending the expressive power of XPath has been the focus of much research [31, 33, 32, 34, 20]. For instance, *Core XPath 2.0* [32], extended Core XPath 1.0 with path intersection, complementation, and quantified variables. *Conditional XPath* [20], extended XPath with 'until' operators, while the inclusion of a least fixed point operator was proposed in [31]. More modest extensions, that better preserved the intuitive clarity and simplicity of Core XPath 1.0, included *Regular XPath* [31], *Regular XPath*$^{\approx}$ [33] and *Regular XPath(W)* [34]. These allowed ex-

[19]Due to space limit and similarity of the result , here we only report 7 out of the 20 XMark queries.

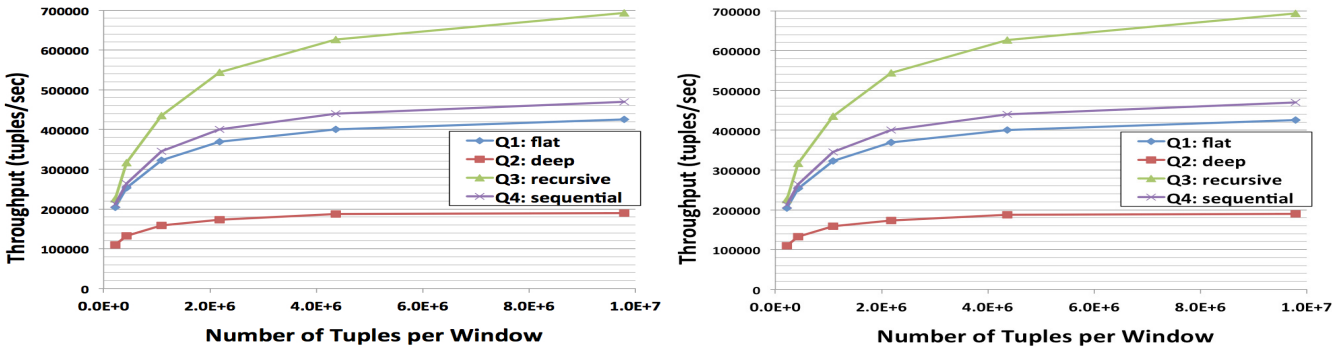


Figure 9: The effect of different types of queries on (a) Total query execution time, (b) Throughput in terms of tuple processing, and (c) Throughput in terms of datasize.

pressions such as $/\mathtt{a}(/\mathtt{b}/\mathtt{c})^*/\mathtt{d}$, where a Kleene-* expression A^*, was defined as the infinite union $\cdot \cup A \cup (A/A) \cup (A/A/A) \cup \cdots$ Even for these more modest extensions, however, efficient implementation remained an issue: in 2006, the following open problem was declared as a challenge for the field [31]: *Efficient algorithms for computing the transitive closure of XPath path expressions.*

VPA. Visibly Pushdown Automata (VPA) have been recently proposed for checking Monadic Second Order (MSO) formulas over dual-structured data such as XML [2, 3], and have led to new streaming algorithms for XML processing [19, 26]. The recently proposed query language K*SQL [22, 21] used VPAs to achieve good performance and expressivity levels needed to query both relational and XML streams. However, while very natural for relational data, K*SQL is quite procedural and verbose for XML, whereby the equivalents of simple XPath queries are long and complex K*SQL statements. At the VPA implementation level, however, the same VPA optimization techniques support both XSeq and K*SQL.

8. CONCLUSION AND FUTURE WORK

We have described the design and implementation of XSeq, a query language for XML streams that adds powerful extensions to XPath while remaining very amenable to optimization and efficient implementation. We studied the power and efficiency of XSeq both in theory and in practice, and proved that XSeq subsumes Regular XPath and its dialects, and hence, provides the first implementation of these languages as well. Then, we showed that well-known complex queries from diverse applications, can be easily expressed in XSeq, whereas they are difficult or impossible to express in XPath and its dialects. The design and implementation of XSeq leveraged recent advances in VPAs and their online evaluation and optimization techniques.

Inasmuch as XPath provides the kernel of several query languages, such as XQuery, we expect that these languages will also benefit from the extensions and implementation techniques described in this paper. In analogy to YFilter [10], where thousands of XPath expressions were merged into one NFA, the fact that VPAs are closed under union creates important opportunities for concurrent execution of numerous number of XSeq queries. Another line of future research is to use XSeq in applications with other examples of visibly pushdown words, such as software analysis, JSON files, and RNA sequences.

9. ACKNOWLEDGMENTS

This work was supported in part by NSF (Grant No. IIS 1118107).

We would like to thank the reviewers, Balder ten Cate, Alexander Shkapsky, Nikolay Laptev and Shi Gao for their comments.

10. REFERENCES

[1] M. Alexander, J. Fawcett, and P. Runciman. *Nursing practice: hospital and home : the adult*. Churchill Livingstone; 2nd edition, 2000.
[2] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, 2004.
[3] R. Alur and P. Madhusudan. Adding nesting structure to words. In *Developments in Language Theory*, 2006.
[4] R. Bamford and et. al. Xquery reloaded. *VLDB*, 2009.
[5] C. Barton and et. al. Streaming xpath processing with forward and backward axes. In *ICDE*, 2003.
[6] P. Boncz and et. al. Monetdb/xquery: a fast xquery processor powered by a relational engine. In *SIGMOD*, 2006.
[7] L. Brenna, J. Gehrke, M. Hong, and D. Johansen. Distributed event stream processing with non-deterministic finite automata. In *DEBS*, 2009.
[8] B. t. Cate and C. Lutz. The complexity of query containment in expressive fragments of xpath 2.0. *J. ACM*, 56(6), 2009.
[9] Y. Chen, S. B. Davidson, and Y. Zheng. An efficient xpath query processor for xml streams. In *ICDE*, 2006.
[10] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *TODS*, 28(4), 2003.
[11] D. Florescu and et. al. The bea/xqrl streaming xquery processor. In *VLDB*, 2003.
[12] T. Furche, G. Gottlob, G. Grasso, C. Schallhart, and A. J. Sellers. Oxpath: A language for scalable, memory-efficient data extraction from web applications. *PVLDB*, 4(11), 2011.
[13] V. Josifovski, M. Fontoura, and A. Barta. Querying xml streams. *VLDB Journal*, 14(2), 2005.
[14] M. Kay. Ten reasons why saxon xquery is fast. *IEEE Data Eng. Bull.*, 31(4), 2008.
[15] D. E. Knuth, J. H. M. Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2), 1977.
[16] C. Koch. Xml stream processing. In *Encyclopedia of Database Systems*. 2009.
[17] N. Laptev, H. Mousavi, A. Shkapsky, and C. Zaniolo. Optimizing Regular Expression Clustering for Massive Pattern Search. Technical report, UCLA, 2012.
[18] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2001.
[19] P. Madhusudan and M. Viswanathan. Query automata for nested words. In *MFCS*, 2009.
[20] M. Marx. Conditional xpath. *TODS*, 30(4), 2005.
[21] B. Mozafari, K. Zeng, and C. Zaniolo. From regular expressions to nested words: Unifying languages and query

execution for relational and xml sequences. *PVLDB*, 3(1), 2010.
[22] B. Mozafari, K. Zeng, and C. Zaniolo. K*sql: A unifying engine for sequence patterns and xml. In *SIGMOD*, 2010.
[23] B. Mozafari, K. Zeng, and C. Zaniolo. High-performance complex event processing over xml streams. Technical report, UCLA, 2012.
[24] D. Olteanu, T. Kiesling, and F. Bry. An evaluation of regular path expressions with qualifiers against xml streams. In *ICDE*, 2003.
[25] F. Peng and S. S. Chawathe. Xpath queries on streaming data. In *SIGMOD*, 2003.
[26] C. Pitcher. Visibly pushdown expression effects for xml stream processing. In *PLAN-X*, 2005.
[27] A. Schmidt and et. al. Xmark: a benchmark for xml data management. In *VLDB*, 2002.
[28] R. T. Snodgrass. Tsql2. In *Encyclopedia of Database Systems*. 2009.
[29] L. Strömbäck and S. Schmidt. An extension of xquery for graph analysis of biological pathways. In *DBKDA*, 2009.
[30] N. V. Tang. A tighter bound for the determinization of visibly pushdown automata. In *INFINITY*, 2009.
[31] B. ten Cate. The expressivity of xpath with transitive closure. In *PODS*, 2006.
[32] B. ten Cate and M. Marx. Axiomatizing the logical core of xpath 2.0. In *ICDT*, 2007.
[33] B. ten Cate and M. Marx. Navigational xpath: calculus and algebra. *SIGMOD Record*, 36(2), 2007.
[34] B. ten Cate and L. Segoufin. Xpath, transitive closure logic, and nested tree walking automata. In *PODS*, 2008.
[35] Z. Vagena, M. M. Moro, and V. J. Tsotras. Roxsum: Leveraging data aggregation and batch processing for xml routing. In *ICDE*, 2007.
[36] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, 2006.

APPENDIX

A. MORE ON XSEQ'S SYNTAX

Conditions. In XSeq, a `Condition` can be any predicate which is a boolean combination of *atomic formulas*. An atomic formula is a *binary operator* applied to two *operand*s. A *binary operator* is one of $=, \neq, <, >, \leq, \geq$. An *operand* is any algebraic combination (using +, -, etc.) of string or numerical constants and terms of the form $X@attr$ where $attr$ is an attribute and $\$X$ is either an `OutBase` or a scalar function applied to a `$variable`

Negation. XSeq does not provide explicit constructs for path negation (e.g. `except` in XPath 2.0). The reason is that by forcing the programmer to simulate the negation with other constructs the resulting query is often more amenable to optimization. For instance, the query of Example 4 could be expressed in XPath 2.0 using their `except` operator as:

```
//son[@Bplace='NY']//son[@Bplace='NY']@Cname     except
//son[@Bplace='NY']//son[@Bplace != 'LA']//son[@Bplace='NY'
  ]@Cname
```

However, as shown in Query 5, this query can be expressed in XSeq without using the negation.

B. BACKGROUND ON VPA

Informally, visibly pushdown words and their closely related models, namely nested words [3], model a sequence of letters (i.e., a "normal" word) together with hierarchical edges connecting certain positions along the word. The edges are properly nested (i.e., edges do not cross), but some edges can be pending. Visibly pushdown words generalize normal words (all positions are internal-data) and ordered trees. Also, natural operations (such as concatenation, prefix, suffix) on words are easily generalized to nested words. Visibly pushdown words have found applications in many areas, ranging from program analysis to XML, and even representations of genomic data [3].

Visibly Pushdown Automata (VPA) are a natural generalization of finite state automata to visibly pushdown words. Visibly pushdown languages (VPLs) consist of languages accepted by VPAs. While VPLs enjoy higher expressiveness and succinctness compared to word and tree automata, their decision complexity and closure properties are analogous to the corresponding word and tree special cases. For example, VPLs are closed under union, intersection, complementation, concatenation, and Kleene-* [2]; deterministic VPAs are as expressive as their non-deterministic counterparts; and membership, emptiness, language inclusion and equivalence are all decidable [2, 3]. Next, we briefly recall the formal definition of a VPA. Readers are referred to the seminal paper [2] for more details.

Let Σ be the finite input alphabet, and let $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i$ be a partition of Σ. The intuition behind the partition is: Σ_c is the finite set of *call* (push) symbols, Σ_r is the finite set of *return* (pop) symbols, and Σ_i is the finite set of *internal* symbols. Visibly pushdown automata are formally defined as follows:

DEFINITION 1. *A visibly pushdown automaton (VPA) M over S is a tuple $(Q, Q_0, \Gamma, \Delta; F)$ where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, Γ is a finite stack alphabet with a special symbol $\perp$ (representing the bottom-of-stack), and $\Delta = \Delta_c \cup \Delta_r \cup \Delta_i$ is the transition relation, where $\Delta_c \subseteq Q \times \Sigma_c \times Q \times (\Gamma \backslash \{\perp\}), \Delta_r \subseteq Q \times \Sigma_r \times \Gamma \times Q$, and $\Delta_i \subseteq Q \times \Sigma_i \times Q$.*

If $(q, c, q', \gamma) \in \Delta_c$, where $c \in \Sigma_c$ and $\gamma \neq \perp$, there is a *push-transition* from q on input c where on reading c, γ is pushed onto the stack and the control changes from state q to q'; we denote such a transition by $q \xrightarrow{c/+\gamma} q'$. Similarly, if (q, r, γ, q'), there is a *pop-transition* from q on input r where γ is read from the top of the stack and popped (if the top of the stack is $\perp$, then it is read but not popped), and the control state changes from q to q'; we denote such a transition $q \xrightarrow{r/-\gamma} q'$. If $(q, i, q') \in \Delta_i$, there is an *internal-transition* from q on input i where on reading i, the state changes from q to q'; we denote such a transition by $q \xrightarrow{i} q'$. Note that there are no stack operations on internal transitions. We write St for the set of *stacks* $\{w\perp | w \in (\Gamma \backslash \{\perp\})*\}$. A *configuration* is a pair (q, σ) of $q \in Q$ and $\sigma \in St$. The transition function of a VPA can be used to define how the configuration of the machine changes in a single step: we say $(q, \sigma) \xrightarrow{a} (q', \sigma')$ if one of the following conditions holds:

1. If $a \in \Sigma_c$ then there exists $\gamma \in \Gamma$ such that $q \xrightarrow{a/+\gamma} q'$ and $\sigma' = \gamma \cdot \sigma$
2. If $a \in \Sigma_r$, then there exists $\gamma \in \Gamma$ such that $q \xrightarrow{a/-\gamma} q'$ and either $\sigma = \gamma \cdot \sigma'$, or $\gamma = \perp$ and $\sigma = \sigma' = \perp$
3. If $a \in \Sigma_i$, then $\gamma \in \gamma'$ and $\sigma = \sigma'$.

A (q_0, w_0)-*run* on a word $u = a_1 \cdots a_n$ is a sequence of configurations $(q_0, w_0) \xrightarrow{a_1} (q_1, w_1) \cdots \xrightarrow{a_n} (q_n, w_n)$, and is denoted by $(q_0, w_0) \xrightarrow{u} (q_n, w_n)$. A word u is accepted by M if there is a run $(q_0, w_0) \xrightarrow{u} (q_n, w_n)$ with $q_0 \in Q_0$, $w_0 = \perp$, and $q_n \in Q_F$. The language $L(M)$ is the set of words accepted by M. The language $L \subseteq \Sigma^*$ is a visibly pushdown language (VPL) if there exists a VPA M with $L = L(M)$.