THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

EFFICIENT XML STREAM PROCESSING AND SEARCHING

By

WEI ZHANG

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Spring Semester, 2012

Wei Zhang defended this dissertation on March 22, 2012.

The members of the supervisory committee were:

Robert A. van Engelen
Professor Directing Dissertation

Erlebacher Gordon
University Representative

Xiuwen Liu
Committee Member

Xin Yuan
Committee Member

Zhenhai Duan
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the dissertation has been approved in accordance with the university requirements.

To Xiaobo, my love, whose wholehearted support helped me complete this dissertation.

ACKNOWLEDGMENTS

This dissertation would not have been completed without the helps of several people. I would like to express my gratitude to my advisor, Dr. Robert A. van Engelen, for his support, patience, and encouragement throughout my graduate studies. He has been instrumental in the successful completion of this work. He provided an environment that nurtured my growth during the early stages of the research process, while giving sufficient freedom and patience during the later stages of my dissertation research. His technical and editorial advice was essential to the completion of this dissertation and has taught me innumerable lessons and insights on the workings of academic research in general.

I would give my thanks to my committee members, Dr. Xin Yuan, Dr. Zhenhai Duan, Dr. Xiuwen Liu and Dr. Gordon Erlebacher for reading my prospectus and previous drafts of this dissertation and providing many valuable comments that improved the presentation and contents of this dissertation. Special thanks would also be given to Dr. David Whalley, my previous comittee member, for his valuable advices on my research and publications, comments and text edits on my prospectus and dissertation drfats.

Last, I would like to thank my wife Xiaobo for her understanding and for her love during the past few years. Her support, encouragement, and dedication to our children was in the end what made this dissertation possible. My parents, Binfan and fanglan, receive my deepest gratitude and love for their dedication and the years of support during my graduate studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF LISTINGS

ABSTRACT

In this dissertation, I present a table-driven streaming XML (Extensible Markup Language) parsing and searching technique, called TDX, and investigate related techniques. TDX expedites XML parsing, validation and searching by pre-recording the states of an XML parser in tabular forms and by utilizing an efficient runtime streaming parsing engine based on a two-stack push-down automaton. The parsing tables are automatically produced from the XML schemas or from the WSDL (Web Services Description Language) service descriptions. Because the schema constraints and XPath expressions are pre-encoded in a parsing table, the approach effectively implements a schema-specific XML parser and/or query processor that combines parsing, validation and search into a single pass. Moreover, the runtime parsing engine is independent of XML schemas and XPath query expressions, parsing can be populated on-the-fly to the runtime engine, thus TDX efficiently eliminates the recompilation and redeployment requirements of schema-specific parsers to address the schema changes. Similarly, different XPath queries can also be preprocessed at compile time and populated on-the-fly to the TDX searching engine without runtime overhead.

To construct the parsing tables, we developed a set of mapping rules that translate XML schemas to augmented grammars. The augmented grammars support the full expressive power of the W3C XML Schema by introducing *permutation phrase* grammars and *multi-occurrence phrase* grammars. The augmented grammars are suitable to construct a predicative parsing table. The predictive parsing table constructed from the augmented grammars can be integrated into the parser at any time to maximize the performance or be populated on-the-fly at runtime and address schema changes efficiently.

Because parsing tables or searching tables are pre-processed at compile time, and looking up the tables at runtime is deterministic and takes constant time, TDX efficiently implements a single pass, predictive validating parser without backtracking or function calling overheads. Our experimental results show a significant performance improvement compared to widely used XML parsers, either validating and non-validating, and to XML query processors.

CHAPTER 1

INTRODUCTION

Extensible Markup Language (XML) [205] has been enormously successful as a markup language for documents and data, and has been acknowledged as the *de facto* standard for data representation and exchange over the Web and in other software systems due to its key advantages in interoperability and flexible expressiveness. By explicitly tagging information with named *elements* and *attributes*, XML enables the creation of documents that are to a significant degree self-describing, offering the promise of more robust information sharing between loosely coupled organizations and systems. An application processing an XML document can use such element and attribute markup to identify particular information items and to detect some classes of errors in document content. However, the interoperability and flexibility of XML often come at the price of reduced efficiency of message composition, transfer, and parsing compared to compact binary protocols. Several studies have evaluated the performance of XML-based protocols such as SOAP and concluded that XML incurs a substantial performance penalty compared to binary protocols [35, 56, 76]. Parsing and validation of XML instances against a schema is expensive [117, 179], as well as the cost of deserialization into usable in-memory objects for applications [51, 76].

In the performance-critical setting of business computing, however, the interoperability and flexibility of XML became a liability due to its potentially significant performance penalty. Traditionally, XML processing is conceptually a multi-tiered task, an attribute it inherits from the multiple layers of specifications that govern its use: XML [205], XML Namespaces [195], XML Information Set (Infoset) [194], and XML Schema [195]. Traditional XML processor implementations reflect these specification layers directly. Bytes, read off the "wire" or from disk, are converted to some known form (often UTF-8 characters) and tokenized (UTF stands for Universal Text Format). Attribute values and end-of-line sequences are normalized. Namespace declarations and prefixes are resolved, and the tokens are then transformed into some representation of the document Infoset; at the same time, checking for well-formed syntax. The Infoset is optionally checked against an XML schema grammar like XML Schema, document type definition(DTD) [60] or Relax NG [54] for validity and rendered to the user through some interface, such as Simple API (application programming interface) for XML (SAX) or Document Object Model (DOM).

A DOM based parser offers many advantages including capabilities of nodes manipulations like insert, delete or update. Such a parser requires the in-memory representation

of the entire document, thus resulting in lower efficiency of memory usage. Furthermore, not only is a DOM expensive to build an in-memory tree representation, but it also incurs long latencies. It may not be preferred by applications that are response-time critical. For example, long latency in a stock exchange service might mean lost opportunities.

A SAX based parser treats the XML data as a byte sequence. It starts to parse and validate the data as soon as it can determine the tags and content data, thus decreasing the latency. Because it does not require the construction of the entire XML message, memory space efficiency is improved. However, streaming data are available for reading only once and are provided in a fixed order determined by the data source. Streaming parsers cannot seek forward or backward in the stream and cannot revisit a data item seen earlier unless these items are buffered. This streaming features and XML features like unordered attributes as well as namespaces impose challenges for streaming parsers. Examples of streaming data include real-time news feeds, stock market data, sensor data, surveillance feeds, and data from network monitoring equipment. Some data are only available in streaming forms because they have a limited lifetime of interest to most consumers. For instance, articles in a topical news feed are not likely to retain their value for very long. Moreover, the data source may lack resources to provide non-streaming access. A network router, for example, which provides real-time packet counts, message forwards, error reports, and security alerts is typically unable to fulfill the processing or storage requirements of providing non-streaming access to such data.

In practice, these tasks are combined to some extent. Typically a generic parser handles scanning, XML normalization, namespaces, and well-formedness checking, as required by the XML specification. Validation is typically performed in a separate phase that sits on top of the generic XML parser module, operating as a filter on the output of the generic parser. Because validation is an add-on in such a design, it has a strictly detrimental effect on parser performance. Validation is, therefore, typically used exclusively for debugging, if at all, and is disabled during production. However, validation is important, because applications that use the XML data typically require the data to be modeled according to a schema. Applications should also not be burdened by having to verify assertions on the data content that can otherwise be done by a validating parser. However, the current separation of parsing and validation in XML parsers incurs significant overhead and requires frequent access to the schema. This overhead can be eliminated by integrating parsing and structural validation into a schema-specific parser.

To address the performance problem, different XML parsing techniques for boosting parsing and/or validating performance have been studied. Among these approaches, schema-specific parsers outweigh in terms of performance. Schema-specific parsers pre-compile an available schema in some manners at compile time so that documents conforming to the schema are parsed and validated efficiently at runtime. The product of the compilation can be classified into specialized APIs [97, 151, 183], data structures [179, 155, 184], or intermediate representations [142]. These schema-specific parsers in this area are limited to DTDs or a limited subset of XML schema and do not include namespace support which makes these approaches useless for applications that require namespace binding, for example, XML Web services.

Moreover, each generated schema-specific parser must be appropriate to the operating system, compiler, supporting libraries, and hardware on which applications will run. The parser must be regenerated and redeployed when an XML schema from which it is constructed is updated. This is a significant challenge and a drawback for developing applications or services in the environments where schemas are subject to change such as developing an extensible Web service. A Web service is usually a long term agreement that allows consumers to interact with a web service. To address schema updates, service designers typically add new elements to their schema by changing the source code, adding the required business logic and rebuilding the service. However, this approach only works for simple services. Consider for example a large business application requires customizations to fit specific industries, countries, and customers. Exposing such business applications as Web services is difficult because they have to be able to be customized over time and these customizations must work for all consumers, even consumers that have made changes to the application. This issue requires that the services have to be designed to be extensible, that is, extensible Web services that typically require extensible XML schemas.

As XML has become the *de facto* standard of messages transformation and storage, it is crucial to efficiently extract nodes and contents from XML messages. XPath [193] was introduced by the W3C as a standard language for specifying node selection, matching conditions, and for computing values from an XML document. It is used in many XML standards such as XSLT [53] and lies at the core of XQuery [197] database access language for XML messages. A general-purpose implementation of an XPath query reflecting the XPath specification for accessing portions of XML documents, for example, an XPath-based API is provided in DOM 3 [201] for traversing DOM trees, typically requires time exponential in the size of queries in the worst case. Since efficient XML content querying is crucial for the performance of almost all XML processing architectures, a growing need for studying high performance XPath-based querying has emerged. The performance of implementations of these languages depends on the efficiency of the underlying XPath query engine. Many techniques for efficiently evaluating XPath expressions over XML messages have been proposed in the past decade. Based on the XML processing modes, XML query processing are largely classified into two models, query processing over a parse tree or over an XML stream.

The parse tree based approaches (for example, [13, 18, 40, 41, 93]) concentrate on finding effective mechanisms for matching query strings to indexed data strings or indexing data paths and data elements to efficiently check structural relationships. Such approaches require that an entire document be constructed in memory before an XPath expression is evaluated. For large documents, such approaches may result in unacceptable overhead because building a DOM or parse tree and indexes over the tree are computing extensive and may consume a large amount of memory. Furthermore, such an XPath query engine may perform unnecessary traversals of the input document. For example, consider an expression such as `/descendant::x/ancestor::y`, which selects all `y` ancestors of `x` elements in the document. The Xalan [13] XPath query engine evaluates this expression by using one traversal over the entire document to find all the `x` elements, and for each x element, a visit to each of its ancestors to find appropriate `y` elements. As a result, some elements in the document may be visited more than

once.

The premise of streaming XPath engines, also known as XML filters, is that in many instances XPath expressions can be evaluated in a single depth-first, document-order traversal of an XML document. A streaming XPath query engine converts a simple XPath expression or a set of XPath expressions into intermediate representations and finds the matched patterns against the streaming XML messages at runtime. The state of the art of such XPath query techniques includes automata-based approaches [34, 58, 91, 102, 138, 149], push-down automata approaches [87], stack based approaches [38, 41] and directed acyclic graph (DAG) based approaches [20]. Such approaches present two major advantages. First, rather than storing the entire document in memory, only the portion of the document relevant to the evaluation of the XPath is stored. Second, the algorithm visits each node in the document exactly once, avoiding unnecessary traversals. However, these XML streaming query techniques typically exhibit the following properties.

- **Separation parsing and query:** All the state-of-the-art streaming XPath query engines typically implement query evaluation on top of a streaming XML parser such as SAX [128]. Query evaluation is performed on the events generated from the underlying parser. This mechanism presents performance constraints on the query engine. On the one hand, the separation introduces an additional layer for the engine, thus introducing an extra overhead that can be eliminated by integrating parsing and query evaluation. On the other hand, because the query engine relies on a separate parser, the underlying parser may become the performance the bottleneck.

- **Nonvalidating query:** Underlying parsers where the streaming query engine sits on top are often nonvalidating to boost the overall query performance. However, validation is important, because applications that use the XML data typically require the data to be modeled according to a schema for the security reason. Applications such as high-performance Web services and content-based publish-subscription systems may require valid XML messages.

- **Runtime expression transformation:** Current streaming XPath query techniques transform the XPath expressions into different intermediate forms on the fly. However, converting the expressions and storing these intermediate representations at runtime is typically expensive.

- **Full XPath features limitation:** Current automata based XML filters are limited to handling location path expressions that only contain forward axes and do not support backward axes handling. Moreover, some of the XPath features cannot be efficiently represented as automaton without exponentially increasing the number of machine states. For instance, the XPath feature `position()`=n selects the n^{th} appearance of an element where n may be an arbitrary positive number cannot be presented using automata effectively.

- **Memory space inefficiency:** In automata-based approaches, XPath expressions are transformed into deterministic or nondeterministic finite automata. Each data node causes a state transition in the underlying finite state automata representation of the filters. The active states of the machine usually correspond to the prefix matches identified in the data. For deep and recursive XML data, the number of active states can be exponentially large [18, 19, 57, 77].

In this dissertation, I present a novel approach to efficiently parse, validate an XML stream and query the stream against a single or multiple XPath query expressions. The general idea of the approach is to encode the parsing states into a compact parsing table associated with semantic actions

for data type validation by pre-processing an available schema at compile time, a schema-independent streaming driver efficiently parses the XML tokens and content value data generated by a scanner at runtime by consulting the parsing table and invoking the semantic actions for validating data type constraints imposed by the schema. A single or multiple XPath expressions can also be pre-processed at compile time and integrated into the table for selected nodes and texts, thus well-formedness parsing, structural and type-check validating, and searching are all integrated. Table construction is only performed at compile time, thus eliminating schema access at runtime. Lookup operations in the table is deterministic and takes constant time. As a result, single pass processing is achieved without backtracking. In addition, validating actions are generated from the schema, thus they are specialized and optimized to improve performance. Moreover, in our approach, scanning, tokenization, parsing and validating together with searching are pipelined and executed simultaneously, to further boost the performance. Because the parsing or the query driver (or engine) is independent of the schema or XPath expressions, it only requires the parsing table that can be populated at runtime. As a result, the requirements of recompilation and redeployment of schema-specific parsers are effectively resolved; thus this approach achieves higher degree of flexibility than other schema-specific parsers.

1.1 A Motivating Example

Consider for example, the XML Schema fragment (Figure 1.1), which describes an element `Items` (line 1) of type `ItemsType`. The `ItemType` defines an element `item` that may have arbitrary occurrences of element `item` (line 3 - line 7). Each element `item` consists of a required attribute `partNum` (line 16) and four child elements `productName` (line 11), `quantity` (line 12), `price` (line 13) and `shipDate` (line 14). These four child elements must appear in that order and the last child element `shipDate` is optional, that is, it may not be present in an instance of the schema. The content value of the attribute `partNum` is restricted to a string pattern. The content constraint of the child element `quantity` must be a positive integer less than 100 (line 19 - line 23). The content constraints of other three child elements are also constrained to built-in data types (string, decimal and date) specified in the XML Schema specification [195]. The attribute `partNum` if of type `SKU` that is a string pattern (line 25 - line 29). A valid instance of the schema fragment is shown in Figure 1.2. The element `shipDate` is not present in the first element `item` (line 2 line 6) while it is present in the second element `item` (line 11).

It is observed that an element type defined by the XML schema must have a start element name and a closing element name. A start element name may be optionally followed by one or more attributes. The element content my be either a simple value of type plain text (built-in type or derived type defined by XML Schema component `xsd:simpleType`) or a set of child elements defined by XML Schema component `xsd:complexType`. By representing the tags defined in the schema as tokens of start-tags, end-tags, and attribute tags as well as a special token for the plain text, the structural constraints imposed by the schema fragment can be equivalently described using the context-free grammar 1.1. As a result, the context-free grammar can be used to parse the instances of the schema using any compiler techniques for context-free grammars. A parse tree for parsing the instance (Figure 1.2) of a schema fragment is shown in Figure 1.4. Because context-free grammar encodes the structural constraints, the validation of the instance structure is implied by the parse tree. However, a valid XML instance of an XML schema must comply to the both structural constraints and be of the valid data type imposed by the schema. The type-checking can be accomplished by semantic actions associated with grammar productions.

```
<xsd:element name="Items" type="ItemsType"/>

<xsd:complexType name="ItemsType">
  <xsd:sequence>
    <xsd:element name="item" type="itemType" minOccurs="0" maxOccurs="
       unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="itemType">
  <xsd:sequence>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="quantity" type="qauntityType"/>
      <xsd:element name="price"  type="xsd:decimal"/>
      <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="partNum" type="SKU" use="required"/>
</xsd:complexType>

<xsd:simpleType name=quantityType>
  <xsd:restriction base="xsd:positiveInteger">
      <xsd:maxExclusive value="100"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
</xsd:simpleType>
```

Figure 1.1: An XML Schema fragment.

```
<items>
  <item partNum="872-AA">
    <name>Lawnmower</name>
    <quantity>1</quantity>
    <price>148.95</price>
  </item>
  <item partNum="926-AA">
    <producName>Baby Monitor</productName>
    <quantity>5</quantity>
    <price>39.98</price>
    <shipDate>1999-05-21</shipDate>
  </item>
</items>
```

Figure 1.2: An XML instance of the schema fragment in Figure 1.1.

A specialized function is generated for each data type in the schema by exploiting the data type definition and is used for semantic actions with the same data types. For example, the `SKU` type, which is defined as a string pattern specified by the regular expression `d{3}-[A-Z]{2}`, can be validated by a DFA (Figure 1.5). This function specialized to the schema definition is much more efficient than generic functions for regular expression matches. Other data types in the schema fragment (`string`, `date`, `decimal` and `quantityType`) are validated in a similar way.

$$
\begin{aligned}
S &\rightarrow \texttt{<Items>}\, I_s \,\texttt{</Items>} \\
I_s &\rightarrow I'_s\, I_s \\
I_s &\rightarrow \epsilon \\
I'_s &\rightarrow \texttt{<item>}\, A\, I \,\texttt{</item>} \\
A &\rightarrow \texttt{[partNum]}\, \{\texttt{text}\} \\
I &\rightarrow N\, Q\, P\, D \\
N &\rightarrow \texttt{<productName>}\, \{\texttt{text}\} \,\texttt{</productName>} \\
Q &\rightarrow \texttt{<quantity>}\, \{\texttt{text}\} \,\texttt{</quantity>} \\
P &\rightarrow \texttt{<price>}\, \{\texttt{text}\} \,\texttt{</price>} \\
D &\rightarrow \texttt{<shipDate>}\, \{\texttt{text}\} \,\texttt{</shipDate>} \\
D &\rightarrow \epsilon
\end{aligned}
\tag{1.1}
$$

Figure 1.3: Grammar generated from the schema in Figure 1.1.

The context-free grammar for the schema fragment can be parsed and validated using recursive-descent algorithms [97, 183] though, recursive-descent parsers usually involve function calling and backtracking overheads. Furthermore, as we described earlier, a recursive-decent parser has the problem with recompilation and redeployment when the schema from which it is generated is updated. We observe that the grammar for the schema fragment exhibits the Left-most Left-derivation LL(1) property[1]. As a result, a table-driven parser can be developed for efficiently parsing and validating the instances of the grammar. The parsing table constructed from the Grammar 1.1 is shown in Table 1.1.

XPath query expressions specify criteria for selecting matched nodes and texts against the XML message. By exploiting the XPath expressions together with the grammar generated from the schema, the parsing table can be marked for matched nodes, thus the parsing table is turned into a query table. By modifying the parsing engine to parsing, validating and outputting only the matched nodes of the XML message, our approach can be adopted to an efficient validating query processor. For instance, the first XPath expression (Figure 1.6) selects all the elements `quantity`. For the XML instance (Figure 1.2), the output is `<quantity>1</quantity> <quantity>5</quantity>`. The second XPath query expression selects the elements `quantity` that the attribute `partNum` of its parent elements (`item`) is equal to the SKU value of "926-AA". Only the second element

[1]The statement is true for this simple schema fragment though, it is not typically satisfied for an arbitrary XML schema. We deal with this problem by augmented LL(1) grammars with permutation phrases and two-stack pushdown parsing engine.

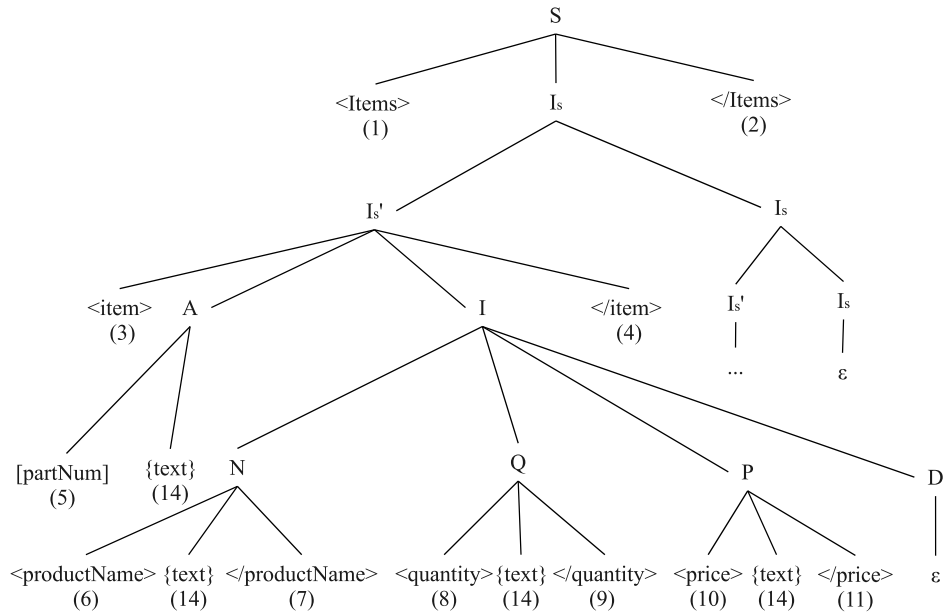


Figure 1.4: Parsing tree for grammar 1.1. The second `<item>` element is not shown in this syntax tree. The number inside the parenthesis represents the integral token.

Table 1.1: Parsing table for the grammar in Figure 1.3. The number inside the parenthesis represents the integral token for that tag.

nonterminals	<Items>(1)	</Items>(2)	<item>(3)	</item>(4)	<partNum>(5)	<productName>(6)	</productName>(7)	<quantity>(8)	</quantity>(9)	<price>(10)	</price>(11)	<shipDate>(12)	</shipDate>(13)	{text}(14)
S	$S \to 1\, I_s\, 2$													
I_s		$I_s \to \epsilon$	$I_s \to I'_s\, I_s$											
I'_s			$I'_s \to 3\, A\, 4$											
A					$A \to 5\,14$									
I						$I \to N\, Q\, P\, D$								
N						$N \to 6\,14\,7$								
Q								$Q \to 8\,14\,9$						
P										$P \to 10\,14\,11$				
D		$D \to \epsilon$										$D \to 12\,14\,13$		

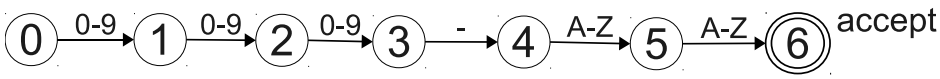


Figure 1.5: A DFA recognizing pattern "[0-9]{3}-[A-Z]{2}" for validating SKU

`quantity` satisfies the query expression, thus the output is `<quantity>5</quantity>`. This query evaluation can be accomplished by marking the parsing table. From the grammar, it is determined that only the production $Q \rightarrow$ `<quantity>{text}</quantity>` satisfies the query expression. For the first query expression, the table entry $T[Q,$ `<quantity>`$]$ is marked to inform the query engine to output the matched element. For the second query, a query predicate function is required to perform satisfactorily by the query engine. The nodes in the marked entries are extracted if and only if the predicate is satisfied. Because each of the query can be marked in the parsing table independently, multiple queries can be processed in an efficient manner.

```
1   /Items/item/quantity
2   /Items/item/quantity[parent::@partNum="926-AA"]
```

Figure 1.6: XPath query expressions selects element `quantity`.

1.2 Thesis Statement

An XML schema is a description of a type of XML document, typically expressed in terms of constraints on the structure and content of a document of that type. These constraints are generally expressed using some combination of grammatical rules governing the order of elements, boolean predicates that the content must satisfy, data types governing the content of elements and attributes, and more specialized rules such as uniqueness and referential integrity constraints. As demonstrated in the motivating example, structural constraints can be parsed and validated by context-free grammars. Text based value constraints can be validated through semantic actions associated with grammar productions. Such grammars are constructed from the schemas by consulting the mapping rules. Mapping rules translate the XML Schema components into grammar productions. However, some of the XML Schema components cannot be efficiently represented as regular expressions or traditional context-free grammars. The `xsd:all` group model, which specifies a set of elements that can appear in any order, and an arbitrary finite number of an element specified by `xsd:minOccurs` and `xsd:maxOccurs`, cannot be represented by the context-free grammars. To address such problems, We introduced *permutation phrase* and *multi-occurrence phrase* productions to extend the context-free grammars.

The augmented LL(1) grammars that are generated from one or more schemas can be used to develop table-driven parsers that integrate parsing and validating XML streams by applying linguistic principles and compiler techniques. By pre-processing XPath queries and marking the parsing table, an efficient validating XML streaming query processor can also be constructed. My thesis statement is:

A set of mapping rules that translate XML schema components into augmented LL(1) grammars can be constructed. When an XML schema is available, an efficient table-driven validating XML streaming parser can be constructed by consulting the mapping rules. Together with XPath query expressions integrated into the parsing table, an efficient validating XML streaming query processor can also be constructed. The performance of parsing, validating and searching XML streams can be significantly improved and a high degree of flexibility to address the schema changes can be achieved.

1.3 Contributions

The proposed table-driven framework for efficiently parsing, validating and searching XML will make the following contributions.

- A set of mapping rules that translate XML Schema components to extended context-free grammars is developed. The rules support full expressiveness of schema components including components that cannot efficiently be represented by traditional context-free grammars by introducing *permutation phrase* and *multi-occurrence phrase* grammars. The augmented grammars are capable of constructing a nonrecursive predicative parsing table.

- Algorithms for preprocessing the XPath expressions and constructing a query evaluation table and predicate validation table are also presented. By encoding the query constraints into the tables, runtime overhead introduced by query evaluation can be minimized, thus a high performance XML query processor can be developed.

- Efficient algorithms for parsing and validating XML documents or streams can be developed from the extended grammars. The parsing table and type-checking validation table are constructed at compile-time, thus the generated validating parser eliminates the access to or processing of the schema at runtime. Type-checking functions are specialized to the specific value constraints and thus are optimized. Parsing and validation are accomplished simultaneously. Furthermore, TDX only performs tokenization once and subsequent operations on the integral tokens is much more efficient than repeated operation on strings.

- An efficient validating query processor for parsing, validating and searching XML documents or streams can be developed by integrating XPath query expressions into the parsing table. XPath expressions are preprocessed at compile-time. The validating query processor integrates parsing, validation and searching in one stage. To the best of my knowledge, this is the first validating XML query processor in the XML query paradigm.

- TDX offers a high degree of modularity and flexibility for developing XML parser, query processor and XML based applications. Because the parsing or query tables are constructed at compile-time, validation functions are triggered by the use of indices, the runtime parsing or query engine is thus independent of the table. When the schemas or the query expressions are updated, the updated tables can be populated accordingly on-the-fly to the parsing or query engine. No recompilation is required. This approach efficiently solves the deployment problem that for which traditional XML schema-specific parsers suffer.

1.4 Organizations

The remainder of this dissertation is organized as follows. Chapter 2 introduces the preliminaries of XML processing stacks. In Chapter 3, the techniques for XML parsing, validation, query processing techniques and alternative XML representations are summarized. In Chapter 4, a set of mapping rules that translate the XML schema components to augmented LL(1) grammar is describes. A Table-Driven parsing and validation parser on XML streams is given in Chapter 5. Chapter 6 presents Table-Driven query processor over XML streams. Performance evaluation on the parser and query processor is given in Chapter 7. A conclusion is draw in Chapter 8.

CHAPTER 2

PRELIMINARIES

This chapter introduces the background to understand the techniques presented in this dissertation. I begin by describing the basic syntax and concepts of the XML markup language, then the W3C XML schema and XPath query languages are introduced. Performance challenges of XML parsing, validating and searching are also discussed in this chapter.

2.1 Extensible Markup language (XML)

XML is a hierarchically-structured language for data representation. It is a markup language offering extra information that indicates how the data is structured. XML, by itself, cannot be used to manipulate the data. It has become the *lingua franca* for messaging among industries, academia and organisations, in particular for loosely coupled organisations, due to its interoperability.

2.1.1 Syntax of XML

XML documents usually start with an XML declaration, which indicates the version of the XML specification to which the document conforms, and the character encoding in which the document is encoded, among other things.

The building blocks of an XML document are elements and attributes. An XML element corresponds to a starting tag and a closing tag, surrounding text, other elements (nested), both text and other elements, or nothing. A starting tag corresponds to a name, possibly followed by one or more attributes, enclosed in angle brackets. Attributes are separated from the starting tag's name and from each other by one or more whitespace characters. A closing tag is a name enclosed in angle brackets, and prefixed with a forward slash. A starting tag and a closing tag corresponding to the same element must have the same name.

Elements with no content (with nothing between the starting and closing tags), can be written using a single tag, empty-element tag, which differs from the closing tag in that the forward slash appears before the right angle bracket, and not before the tag's name.

Attributes are character strings in the form `name=value`, with `value` being enclosed in either single or double quotes. Names of tags and attributes are case-sensitive.

Figure 2.1 shows a sample XML document. In this figure, vehicle is an XML element that consists of three other elements: make, model, and year, each of which has text-only content. The vehicle element has a single attribute, type, having a value of car.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <vehicles>
3   <vehicle type="car">
4   <!-- This is a comment -->
5   <make>nissan</make>
6   <model>altima</model>
7   <year>2002</year>
8   </vehicle>
9 </vehicles>
```

Figure 2.1: A sample XML document.

The first element following the XML declaration in an XML document is the document element. All other elements must appear as children of the document element. The document element in Figure 2.1 is `vehicles`.

XML elements must be properly nested. That is, if an XML element `b` appears inside element `a` (`b` is said to be a child element of `a`, or, equivalently, `a` is a parent element for `b`), then `b`'s closing tag must appear before `a`'s closing tag in the XML document. This is called well-formedness.

The text surrounded by `<!--`and `-->`, in Figure 2.1, is a comment. It is ignored by an XML processor. Any legal XML characters can be used in a comment except that comments cannot be terminated with `-->`. In other words, the two dashes before the right angle bracket cannot be preceded by a dash.

2.1.2 Handling White Space in XML Documents

White space (spaces, blank lines and tabs) is used in XML documents to set apart markup. Such white space is typically not intended for inclusion in the delivery of the document, and is typically ignored and not passed to the application by an XML processor. For example, at least one whitespace character between attributes in an attributes list is required, but an unlimited number of whitespace characters can be inserted between attributes. Also, an unlimited number of whitespace characters can be inserted between the name and the equal sign, as well as the equal sign and the value in an attribute declaration.

On the other hand, white space that should be preserved in the delivery is also common. An XML processor must always pass all characters that are not markup through to the application. For example, white space in source code, poetry etc. should be preserved for readability.

Sometimes, the application decides whether whitespace is legal or not and, if legal, whether it should be ignored or not. For example, whitespace between a starting tag of an element and its corresponding closing tag of a subsequent element, which are known to only have element content is typically ignored by applications. One of the techniques underlying differential serialization, which is discussed in Section 3.3.2, is based on explicitly adding whitespace between XML tags.

2.1.3 XML Namespaces

With the widespread use of XML, it is common to have two elements with the same name, but with different semantics associated with them, especially in different specifications that are based on XML. The XML namespaces specification [191] was created to solve this problem. The specification

extends the XML syntax to allow names of elements and attributes to belong to namespaces, which are identified by URIs.

For the purpose of declaring namespaces, the XML namespaces specification reserves a set of attributes; all attributes that have names starting with `xmlns`. The reserved attributes declare namespaces in two ways. The first is by associating a namespace alias with a namespace and using the alias as a prefix to associate a namespace with it. The second, which only applies to element names, is by changing the default namespace, which is the namespace for element names that are not prefixed with an alias. In both ways, the scope of the declaration is the element in which the reserved attributes appear, as well as its descendant elements. Both the default namespace and namespace aliases declared in parent elements can be overridden by declarations in child elements.

For associating an alias with a namespace, the alias is declared as an attribute prefixed with `xmlns` and having a value that is a URI, which is the name of the namespace. For explicitly declaring an element or an attribute to belong to a certain namespace, the name of the element or attribute is prefixed with the alias, followed by a colon. The XML namespaces specification modified the XML grammar by removing the colon from the set of characters that are legal for a name of an element or an attribute. When the name of an element or an attribute is prefixed with an alias and a colon, it is called a qualified name. If it is not prefixed, then it is called an unqualified name. For example, the alias `ns1`, declared in element `b`, in Figure 2.2 is associated with the namespace `uri:namespace1`. The first child element of element `b` is explicitly declared to belong to the namespace `uri:namespace1` by prefixing it with `ns1` and a colon. The second child element does not belong to the same namespace since its not prefixed with an alias. In fact, the second child element does not belong to any namespace, since the default namespace is initially undefined. Element `b` also has two attributes named att. The first belongs to namespace `uri:namespace1` because its name is prefixed with `ns1` and a colon.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <a>
3    <b xmlns:ns1="uri:namespace1" ns1:att="in ns1" att="not in ns1">
4      <ns1:element>My element belongs to ns1</ns1:element>
5      <element>My element belongs to the default namespace</element>
6      <element xmlns="uri:namespace2" attr="not in ns1 or ns2">
7        <cElem>My element and parent element belong to ns2</cElem>
8      <ns1:element>My element belongs to ns1</ns1:element>
9      <element xmlns="">
10       My element does not belong to a namespace
11     </element>
12     <!-- Default namespace undefined here -->
13     </element>
14   </b>
15   <!-- The alias "ns1" cannot be used here -->
16 </a>
```

Figure 2.2: A sample XML document with namespace.

The default namespace is changed by setting the value of the `xmlns` attribute. In Figure 2.2, the third child element of element `b` declares the default namespace to be `uri:namespace2`. Note

how the name of its child element, `cElem`, implicitly belongs to `uri:namespace2`. However, attribute names are not affected by the declaration of the default namespace; the name of the attribute `attr` still does not belong to any namespace. Finally, note how the last child element of the third child element of element `b` re-declares the default namespace; the special empty value for the `xmlns` attribute can be used for this purpose.

2.2 XML Schema

For an XML document to be interpretable, an XML schema must be associated with it. An XML schema is a set of rules that define the legal contents of XML documents. For example, a rule in an XML schema can state that an item XML element can only have a price and a description child elements, in that order. An XML document that conforms to the XMLs syntax rules is said to be well-formed. A well-formed XML document that conforms to a particular schema is said to be valid with respect to that schema. All valid XML documents with respect to a particular schema are said to be instances of that schema.

An XML schema need not explicitly exist for an XML document to be processable. For example, a schema describing a particular SOAP document typically implicitly exists as part of the SOAP deserializers code. However, a more modular approach is to explicitly define the valid contents of XML documents using an XML schema language.

One of the early XML schema languages is the Document Type Definition (DTD). DTD is described as part of the XML 1.0 specification, and is, therefore, recognized by many XML parsers. Other XML schema languages include Document Structure Description (DSD) [60], RELAX NG [54], and Schematron. However, the most widely used XML schema language is the W3C XML schema language which is a W3C standard to describe XML Schemas in XML.

The XML Schema language is an XML-based schema language. That is, a schema is an XML document itself. The language supports a rich set of primitive (or simple) types and has powerful constructs for defining new types from existing ones. Primitive types can be used to define the legal contents of XML elements with character-only content (i.e., elements that do not contain child elements). For example, a schema document can indicate that a particular element in instance documents is of type int (an integer). This element must, in all valid XML documents, contain character data that is within the lexical space of the W3C XML schemas int type. Thus, an XML document containing the character data `12345` for an element of type int is valid with respect to the schema, whereas an XML document containing the character data `123a5` for the same element is not.

Figure 2.3 shows am XML schema document in the figure indicates, every XML schema document must begin with a schema element, and all XML schema XML elements are defined in the namespace *http://www.w3.org/2001/XMLSchema*. A `complexType` element can be used to define a type that corresponds to XML elements containing attributes, child elements, or a mix of child elements and character data (mixed content).

In the W3C XML schema document of Figure 2.3, a `complexType` defines a type named `VehicleElementType`, containing an ordered sequence of three XML elements, make, model, and year. The make, model, and year elements are defined to contain character-only content of types string, string, and int, respectively. In addition, the schema indicates that elements of type `VehicleElementType` can have an optional attribute of type `VehicleType`.

The `VehicleType` is a simple type defined by restricting the set of legal values, or the value space, of the type string. Specifically, `VehicleType` is defined so that only car, truck, and bus are its legal values. Finally, the element vehicles is defined to be an XML element containing a sequence of one or more child elements, named `vehicle`, and are of type `VehicleElementType`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="vehicleElementType">
    <xsd:sequence>
      <xsd:element name="year" type="xsd:int"/>
      <xsd:element name="model" type="xsd:string"/>
      <xsd:element name="make" type="xsd:string"/>
      </xsd:sequence>
      <xsd:attribute name="type" use="optional" type="vehicleType"/
          >
    </xsd:complexType>
    <xsd:simpleType name="vehicleType">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="car"/>
        <xsd:enumeration value="truck"/>
        <xsd:enumeration value="bus"/>
      </xsd:restriction>
    </xsd:simpleType>
  <xsd:element name="vehicles">
  <xsd:complexType>
    <sequence minOccurs="1" maxOccurs="unbounded">
      <xsd:element name="vehicle" type="vehicleElementType"/>
    </sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Figure 2.3: A sample XML schema.

2.3 XML Parsing and Validation

While the most common XML format is the text format, applications typically process XML documents in a more abstract format, XML's data model, that models an XML document as a tree-structured graph. Parsing XML involves transforming the textual XML format to its abstract data model. Several APIs exist for parsing XML. The most common are the Simple API for XML (SAX), the Document Object Model (DOM), and the Streaming API for XML (StAX). Applications using the SAX API register callback functions (handlers). When the XML parser recognizes a particular XML construct for which the application has registered a handler, it notifies the application by calling that handler. For this reason, SAX parsing is characterized as a push parsing model. DOM parsers, on the other hand, create a hierarchical data structure corresponding to the XML document before returning this structure to the application. The application can then conveniently traverse

this data structure to extract information from the XML document. Finally, with StAX parsing API, applications drive the parsing process by explicitly calling StAX primitives to extract XML constructs from the document. For this reason, StAX parsing is characterized as a pull parsing model.

Clearly, the three parsing APIs differ in their performance and ease-of-use characteristics. StAX is the fastest parsing paradigm, since it avoids the overhead of calling handlers and does perform any work besides recognizing XML constructs. It is also the most flexible, since both SAX and DOM can be efficiently implemented using StAX. However, it is typically the least convenient to use, especially when the XML document is complex or the application is only interested in a small part of the XML document. SAX parsing, on the other hand, can be slower than StAX due to function call overhead, but is more convenient to use when XML document is complex or the application is only interested in small parts of the XML document. Finally, DOM parsing is typically the most convenient to use and is most useful when the application needs to traverse the XML document multiple times. However, it is the slowest parsing paradigm, since it creates a data structure in memory, which involves memory copy and allocation operations. Furthermore, a DOM parser's memory requirements can be large, and, for this reason, may not be used to parse relatively large XML documents.

2.4 XPath Querying Languages

XPath 1.0 [193] is a language with a large number of features and therefore somewhat unwieldy for theoretical treatment. In this section, we introduce only some of these features, and to giving an informal explanation of their semantics. For a detailed definition of the full XPath language.

XPath is a language for addressing parts of an XML document. The XPath language treats XML documents as a tree of nodes (corresponding to elements) and offers an expressive way to specify and select parts of this tree. XPath expressions are structural patterns that can be matched to nodes in the XML data tree. The evaluation of an XPath query expression yields an object whose type can be a node-set, a Boolean, a number, or a string. For XPath expressions retrieval problem, an XML document matches an XPath expression when the evaluation result is a non-empty node set.

$$
\begin{aligned}
Q &\rightarrow N + [O] \\
N &\rightarrow \{/|//nodetest[P]\} \\
P &\rightarrow [F|OPconstant] \\
F &\rightarrow @attribute|nodetest[@attribute]|text() \\
O &\rightarrow @attribute|text()|cout()|last() \\
OP &\rightarrow > \mid \geq \mid = \mid < \mid \leq \mid \neq
\end{aligned}
\tag{2.1}
$$

Figure 2.4: EBNF for a subset of XPath.

A simplified grammar for XPath is depicted in Figure 2.4. An XPath query is an expression pf the form $N_1N_2 \ldots N_k/[O]$, which consists of a location path, $N_1N_2 \ldots N_k$, and an optional output function O. Each location step B_i is of the form $/a_i :: n_i[p_i]$ where a_i is an axis, n_i is a nodetest

that specifies the name of elements N_i can match, and p_i is an optional predicate that is specified syntactically using square brackets.

An XPath query is interpreted as follows. Each location step selects a set of nodes in the document tree. For every node x selected by N_{i-1}, N_i selects a set of nodes using x as the context node. The set of contest nodes selected by the last location step consists of the result set of the query. Because there is one root element id an XML document, N_i is always evaluated using the document root as the context node. The axis in a location step N_i specifies the relation between a node y selected by N_i and the context node x in which N_i is evaluated. In the simplified grammar, `/` is a shorthand for the `/child::` axis, which specifies that y must be xs child. Similarly, `//` is a shorthand for the `/descendant-or-self::node()/` axis, which specifies that y must be a descendant of x (not necessarily a proper descendant). After N_k is evaluated, the output function O is applied to every node in the result set to produce the final output. The output function may specify an attribute or the text value of an element. It may also use an aggregation function such as $sum()$ or $count()$. If no output expression is specified in the query, the elements in the result set are returned as the query result.

For example, the XPath query `/a/b//c` selects all `c` element descendants of all `b` elements that are direct children of the root element `a` in the document. XPath also allows the use of a wildcard operator ($\star$) to match any element name at a location step.

Each location step can also include one or more predicates to further refine the selected set of nodes. Predicate expressions are enclosed by "`[`" and "`]`" symbols. The predicates can be applied to the text or the attributes of the addressed elements, and may also include other path expressions. Any relative paths in a predicate expression are evaluated in the context of the element nodes addressed in the location step at which they appear. For example, the query `/a[b[@x100]/c]/*/d` specifies a tree-structured pattern starting at the root element a with two child "branches" `b/c` and `*/d` such that the element `b` has an attribute `x` with a value of at least 100.

2.5 Performance Challenges of XML Parsing, Validation and Searching

XML processing inefficiency arises from a number of causes. In this section, we presents the impacts that affect the XML processing performance and challenges of XML processing.

Text Based Formats and Verbosity. XML contains lots redundant information such as paired element tags and is text based format. Deserialization requires creation of an in-memory object and converts these text tag names and string values into its equivalent binary representation on a native machine. Serialization does this conversion in reverse. Such conversion between string and binary representation incurs intensive CPU computation. For example, many studies have shown that converting floating-point numbers from ASCII to a native machine representation or vice versa involves intensive CPU computation cost.

XML Namespaces. XML namespaces qualify XML element and attribute names by using namespace-prefix declaration. The declaration takes form of a special attribute with a reserved prefix (`xmlns`) followed by the prefix to be declared. The value of this attribute is the declared namespace. The scope of the namespace declaration includes the enclosing element, all of the sibling attributes, and the elements content. This arrangement, although natural, presents some difficulties for XML

processors. During the processing, namespace declarations prevent the qualified names of the element and its attributes from being conclusively know until the end of the tag is read. This determines that scanning of qualified names in XML requires infinite look-ahead to fully resolve names. Throughout the XML processing stack, markup and meta-markup (such as namespace declarations) assert scoped properties and declarations for the containing element and all of its attributes, as well as its content. In the case of XML namespaces and the dynamic typing mechanism used in XML schema, this pattern presents some difficulties for naive parser implementations.

Performance Challenges Imposed by XML Schema. XML schema, and the specifications on which it depends, present several challenges to schema-based parser generation. The dynamic typing features of XML Schema complicates the scanning markup. As a result, the schema grammar and the lexical production of XML are not easily combined with traditional grammar compilation techniques. Additionally, XML schema provides support for content models that are difficult to represent in traditional automaton models, making the traditional models inefficient as intermediate representations of the schema.

The syntax of `xsi:type` is similar to that of namespace declarations and poses the same kind of processing hurdles. In particular, the possibility of an `xsi:type` attribute prevents an XML processor from conclusively determining the type declaration to use for validation until the entire tag has been scanned. Furthermore, because the element type declaration governs the type declarations used to validate the attributes, the processor cannot conclusively determine the types — and therefore the validity—of the attributes until the entire tag has been read.

Like the Document Type Definition (DTD) grammar used in XML, XML schema can specify an element's content models as a regular expression over its contained element. In contrast to the grammars that can be specified with an XML DTD, however, XML Schema supports wider range of operators in the composition of content models. In particular, the arbitrary finite occurrence constraints and xsd:all groups of XML Schema pose challenges to automaton-based approaches to compilation.

Models of `xsd:all` group content are not represented in any standard regular expression syntax and require significant augmentation of the automaton model. If translated in directly into a standard automaton model, an `xsd:all` group results in an expansion of states that is combinatorial in the number of members of the group. Because the `xsd:all` compositor is not well represented in traditional models, much of the work on XML Schema compilation has treated xsd:all groups as a "corner case". In practice, however, `xsd:all` is considered to be a natural translation of a data structure with named fields, such as a C struct or a Java class, where the members are identified by name, rather than by position. In practice, we have seen that XML schemas for data stored in relational databases often have a plethora of `xsd:all` groups. These scenarios are quite common for Web services.

Arbitrary finite occurrence constraints can lead to an explosive growth in the number of states for simple automaton-based approaches. In the standard implementation, for example, an element declaration with a maximum occurrence constraint of 5000 will result in 5000 states corresponding to each possible occurrence in the range.

XML Character Data. In addition to the particular challenges posed by the various specifications involved in XML parsing and validation, the layering of the specifications presents challenges of its own. In particular, the constraints imposed by an XML schema operate on an abstract representation of an actual XML instance described in the InfoSet as an abstract tree of information items. All data in the InfoSet is represented in a fully expanded form, with entities and character references

expanded, CDATA sections replaced with their contents, and end-of-line and attribute normalization completed, as required by XML. This means that the lexical productions and value constraints used by XML Schema to constrain data are defined in such abstract form. As a result, these constraints are typically implemented in a two-pass method, where the content is first scanned according to the lexical-level productions of XML Schema, and then normalized and validated against its constraining type. This procedure is inefficient because it requires th data to be scanned twice.

Streaming XML Processing Challenges. Streaming XML data are available for reading only once and are provided in a fixed order determined by the data source. Applications that use such data cannot seek forward or backward in the stream and cannot revisit a data item unless they buffer it. This imposes significant performance challenges on XML streaming parsing and searching. Because well-formed XML message must be appropriately nested, an XML streaming processor may have to cache or buffer sufficient information until it is determined. Consider for example, the query `/pub[year > 2000]/book[price < 11]/author` on the input fragment depicted in Figure 2.5.

```
1  <!-- begin document -->
2    <pub>
3      <book id="1">
4        <price> 12.00 </price>
5        <name> First </name>
6        <author> A  </author>
7        <price type="discount"> 10.00 </price>
8      </book>
9      <book id="2">
10       <price> 14.00 </price>
11       <name> Second</name>
12       <author> A </author>
13       <author> B </author>
14       <price type="discount"> 12.00 </price>
15     </book>
16     <year> 2002 </2002>
17   </pub>
18 <!-- end document -->
```

Figure 2.5: An XML fragment.

Intuitively, it returns the authors of books that have been published after year 2000 and that have a price less than 11.

When we encounter the first author element (line 6) in the stream, it is easy to deduce that the sequence of its ancestor elements matches the pattern `/pub/book/author` (since the `pub` and `book` elements have been encountered earlier and are still open). The predicate `[year > 2000]` is not satisfied by the `pub` element (line 2) because we have not encountered any `year` child elements. However, qualifying child elements may occur later in the stream. Therefore, we cannot yet conclude that the predicate is false. For the `book` element (line 3), we have encountered the first price element (line 4), which does not satisfy the predicate `[price < 11]`. Again, we cannot yet conclude that the predicate is false for this `book` element because it may have additional

price child elements later in the stream. Thus, at line 6, we cannot determine whether the `author` element belongs to the result. The element must therefore be buffered.

When we encounter the `price` element on line 7, we can check that it satisfies the predicate for its parent `book` element. However, we still cannot determine whether the `pub` element on line 2 satisfies the predicate `[year > 2000]`. Consequently, it is still unknown whether the `author` element on line 6 belongs to the result. Therefore, we must continue to buffer the `author` element and record the fact that the second predicate has been satisfied but not the first one. Similarly, the two `author` elements on lines 12 and 13, which belong to the second book element, have to be buffered as well. At this point in the stream (line 13), there are three `author` elements in the buffer: two with value `A` and one with value `B`.

When the `price` element (line 14) is encountered, we note that it does not satisfy `[price < 11]`. Since its parent `book` element is still open, we cannot yet conclude that the `book` element fails to satisfy the predicate. That conclusion can only be made when we encounter `</book>` on line 15. At this point in the stream (line 15), the two author child elements of this `book` element should be removed from the buffer. The other `author` element (with value `A`) remains in the buffer because its first predicate may be satisfied by data encountered later in the stream.

As the `year` element is encountered on line 16, we may determine that the `pub` element on line 2 satisfies the predicate `[year > 2000]`. Recalling that this `pub` element is the ancestor of the `author` element remaining in the buffer, which has already satisfied the other predicate, we determine that this `author` should be sent to the output.

CHAPTER 3

RELATED WORK

There have been many different efforts directed to address the inefficiency of XML processing. In this chapter, previous work and techniques for boosting XML parsing, validation, serialization and deserialization, query and filtering are summarized briefly.

3.1 Parsing Optimizations

DOM, SAX, and XPP (XML Pull Parser) typically require two passes through the XML document: the parser tokenizes the document in the first pass by identifying element start-tags, end-tags and attribute tags, and the application processes the content in the second pass. As XML technology is used in more and more performance-critical contexts, these widely used first generation Application Programming Interfaces (APIs) present a gap between the XML parsing performance and the requirements of performance-critical aspect of the next generation of business computing infrastructure. They also impose a challenge on the applications with limited computing resources, for example cell phones and Personal Digital Assistants (PDAs) typically have very limited memory.

To address the performance problem, XML parsing techniques for boosting parsing performance have been studied including early work by XML formalisms (Murata et al. [132], Lowe et al. [117]).

3.1.1 Schema-specific Parsing

Schema-specific parsing refers to the technique of pre-compiling an available schema in some manner so that documents conforming to the schema are parsed efficiently. The product of the compilation can be represented as specialized APIs [97, 151, 183], data structures [155, 179, 184], or internal tables [142, 208, 209, 211, 210]. Schema-specific XML parsing achieves performance gains by exploiting schema information to compose a parser at compile time and utilizing the parsing states at runtime to verify schema validation constraints.

One theme of schema-specific XML parsers is to translate schema grammars to recursive functions. In [183], van Engelen presents a toolkit called gSOAP that generates schema-specific LL(1) (denotes Left to right, Leftmost derivation and one lookahead) recursive descent parser for XML with namespace support and limited validation by generating a specialized API in native C/C++ from a set of given schemas. This API implements a single-pass schema and dual-pass encoding of the application's object graphs in XML. As best of my knowledge, this is the first work for an integrated approach to schema-specific parsing by collapsing scanning, parsing, validation, and

deserialization into a single phase. However, recursive descent parsing incurs function calling and backtracking overheads. In addition, the generated parser has the property of a blocking parser that may suspend the entire program until sufficient XML content arrives over the network. Although new gSOAP releases handle the most common schemas, including schemas with `xsd:all` and `xsd:choice`, there are still limitations on schema validation.

Perkins et al. [151] present a framework for generation of efficient parsers through direct compilation of XML schema grammars. Based on this work, in their later work [97], they present a parser generator, called XML Screamer, which translates XML schema into a parser either in C or Java code. This work demonstrates that with careful API design and implementation, an efficient specialized XML parser can be generated, and XML can be processed much more efficient than most common practice APIs. The generated validating parser drives the optimized scanning process. Two complementary optimization scanning strategies, specialization and optimistic scanning, are used to boost scanning and validation. Specialization attention is paid on the use of specialized context-sensitive scanning primitives that can scan and validate the input efficiently. Optimistic scanning speeds the scanning of the common cases, such as simple data without comments or entity references. Similar to gSOAP [183], XML screamer implements a recursive-descent parser that integrates parsing, validation, and deserialization with scanning, and achieves high performance. This approach uses recursive descent with backtracking, and covers a large schema space, but does not offer full schema support or DTD external or internal subsets. As with all recursive descent parsers, XML Screamer is a blocking (synchronous) parser and may result in inefficient memory consumption. Their more recent work [125] builds an interpretive validating parser based on XML Screamer called iScreamer. iScreamer is a schema-directed interpretive XML parser and achieves high-performance gains by using a carefully tuned set of special-purpose bytecodes that are very similar to assembly languages. As with an interpretive parser, iScreamer is slower than compiled parsers. Because it was built on XML Screamer, it does not support full schema features. Also, its reliance on specialized bytecodes may hinder its acceptance.

Thompson et al. [179] present an approach to convert XML Schema content models to finite state automata (FSAs) capable of handling of numeric exponents and wildcards. The schema is first represented as a schema abstract data model represented with regular expressions. These regular expressions are converted into FSAs in which edges are labelled term schema components using a modified Aho and Ullman's algorithm [10]. Substitution group is treated as disjunctions and occurrence indicators other than 0, 1 or unbounded are handled by unrolling the sub-expression governed by the indicator. Enforcement of the Unique particle attribution constraint is performed by relabelling the constructed FSA with expanded element names, and then checking by cases for ambiguity for element-element, element-wildcard and wildcard-wildcard cases. A check algorithm is implemented using two augmented FSAs, which essentially works by attempting to parse every path through the derived FSA using the base FSA, with special attention to certain corner cases for wildcards. If the parse succeeds, the base FSA subsumes the derived FSA, which in turn therefore accepts a subset of what the base accepts. Unfortunately, conversion to an FSA is polynomial in time and exponential in space with respect to the size of the original content model considered as a regular expression. The time complexity of determination is quadratic in the number of automata states.

Chiu et al. [52] suggest an approach to merge all aspects of low-level parsing and validation by extending DFAs to nondeterministic generalized automata. They also describe a technique for translating these nondeterministic generalized automata into deterministic generalized automata. Generated parsers operate on a byte level, performing well-formedness checking and validation

concurrently as in [97, 151], thus speeding up the parsers. However, translating from nondeterministic automata to deterministic automata can cause a multiplicative blowup in the number of states, thus limiting these parsers to small number of occurrence constraints. This approach does not support XML namespace, which is an essential requirement for SOAP compliance. In general, this solution subsets the XML specification and XML Schema Recommendation in many important ways, excluding many commonly used features.

In [184], van Engelen presents a method that integrates parsing and validation into a single stage by using a two-level DFA in which a lower-level Flex scanner drives a DFA validation. The use of two-level DFAs can significantly reduce the number of states. The DFA is directly constructed from a schema based on a set of mapping rules. Unfortunately, the FLEX scanner does not take advantage of the schema knowledge, either in scanning tags or in scanning simple data. It handles more of the XML specifications than Chiu and Lu [52], however, this solution cannot process cyclic XML Schema content model which is not uncommon. The fact that the DFA lacks support for the cyclic Schema content model is a common limitation shared with all the finite automata based solutions such as in [52, 155] due the limitations of regular languages recognized by DFAs.

In our early works [208, 209], we present a novel approach called a table-driven XML (TDX) parser for efficiently parsing and validating streaming XML messages simultaneously. TDX expedites XML parsing by pre-recording the states of an XML parser in compact tabular forms and by utilizing an efficient runtime streaming parsing engine based on a push-down automaton. The parsing tables are automatically produced from schemas or WSDL service descriptions and encode parsing states in a compact tabular form. Because the schema constraints are pre-encoded in a parsing table, this approach effectively implements a nonrecursive predicative parser that combines parsing and validation into a single pass, thus eliminating the recursive function calling and backtracking overheads. This approach significantly increases the performance of XML parsing and validation. Because the parsing engine is independent of the schemas, the parsing table can be populated to the parsing engine on-the-fly when updates of schemas are presented. Therefore TDX offers a flexible framework to address the drawbacks of the traditional schema-specific XML parsers, that is, requirement of recompilation.

In our more recent work [210, 211], we extend the table-driven approach to handle all the XML Schema content models by introducing the *permutation phrase* grammar and the *multi-occurrence phrase* grammar. XML Schema content model `xsd:all` can be efficiently evaluated by a single permutation phrase production. Similarly, finite arbitrary occurrences constraints can be parsed by a single multi-occurrence production. TDX encodes parsing states in a very compact table and looking up the table is determined by the current token and the parsing state. Thus TDX achieves both performance and memory efficient. Furthermore, the table-driven approach is capable of associating user-provided application-specific actions, thus offering a high-level of flexibility.

Padovani et al. [142] present an lookahead left to right (LALR) grammar based XML parser for integrated parsing and validating streaming XML messages. LALR grammar based parsers typically implement table based bottom-up parsing technology. However, their solution relies on a SAX parser to break the input to the LALR parser, which may dominate the performance gains. In addition, unlike our table-driven solution, this solution does not handle the unordered Schema content models like `xsd:all` and arbitrary occurrences constraints in an efficient manner.

3.1.2 Parallel XML Parsing

Parallel XML Parsing (PXP) [118, 119, 120, 143, 144] presents a parallel processing model to speedup XML parsing by leveraging the parallelism provided by multi-core architectures. XML parsing is inherently sequential because the state of the parser depends on preceding characters when reading a character. The PXP implements a two-pass-scanning stage DOM parser: *preparsing* and *parallel XML parsing*. The former performs a quick scanning to determine the topological structure of the elements in the document to guide the parallel parsing. As a result, a light-weight tree structure, *skeleton*, is generated. The skeleton can be viewed as index of all well-formed fragments in the XML document. Once the skeleton is generated, the XML document is then portioned into disjoint chunks that are parsed in parallel by multiple threads. The partial result generated by a thread is the forest of the DOM fragments, each corresponding to a node of the skeleton. A final DOM tree is constructed after all the threads have finished parsing.

For the parallel XML parsing in multiple threads, one of the challenges is the partitioning and load-balancing. Three partitioning and load-balancing technologies have been introduced: *static partitioning*, *dynamic partitioning*, and *dynamic partitioning* [119, 120]. A static scheme [143] can reduce synchronization and load-balancing overhead, which may improve performance over dynamic schemes, but only for a small class of XML documents, for example, an XML document consisting of a large number of arrays. Unlike static schemes, dynamic schemes make partitioning and load-balancing decisions on the fly. A static scheme [118] implements a request-response scheme for the load-balancing on-the-fly, thus introduces request-response overhead between the threads. Dynamic partitioning [119, 120] improves performance by the use of stealing based load balancing scheme [27]. To solve the serial preparsing bottleneck, Pan et al. [144] propose a parallel solution for the preparsing using Meta-DFAs, whose transition function runs multiple copies of the original automaton simultaneously.

Parabix (Parallel bit streams for XML) [90] is high-speed XML parsing engine that employs parallel bit streams technology [37] to provide dramatic performance improvements over traditional byte-at-a-time parsing. Parabix offers a fundamentally new way to perform high-speed parsing of XML documents. It leverages the SIMD processor capabilities of commodity CPUs to process parallel bit streams that represent a byte-oriented character stream. Byte-oriented character stream data is first transformed into eight parallel bit streams, each bit stream comprising one bit per character code unit. Code units may be ASCII characters or UTF-8 bytes, for example, with one parallel bit stream defined for each of bit 0 through bit 7 of each code unit. Given such a representation, the 128-bit SIMD (single-instruction multiple-data) registers of the SSE (Intel architecture SIMD technology) or Altivec (Power PC architecture) may be used to process 128 code unit positions at a time. Parallel bit streams for XML technology with capabilities of hardware support offers a great performance gains compared with traditional byte-at-a-time text processing technology.

Parallel TRIE [100] achieves great performance speedup and memory consumption reduction for XML schema validation component of Intel XML Software Suite [55] with the use of Intel Streaming SIMD Extensions 4 (SSE4). Intel SSE4 is a new set of Single Instruction Multiple Data (SIMD) instructions that improve the performance and energy efficiency of a broad range of applications. Intel SSE4.2/STTNI offers four instructions for "string and text processing" to simultaneously operate on two 128-bits (16 bytes or 8 words) strings. Unlike traditional TRIE, parallel TRIE represents each layer from a sparse array to a compact string, combines adjacent two characters (bytes) as one short atomic integer to reduce the number of layers, and organizes parallel TRIE to fully utilize capability of parallel string comparison provided by Intel SSE4.2/STTNI for high performance string lookup.

Letz et al. [108] implement an XML parser that employs the parallelism of the Cell BE processor architecture. The Cell BE processor architecture features nine cores on a chip: one Power Processor Element (PPE) running the operating system, and eight independent Synergistic Processing Elements (SPEs) running the application threads [152]. The parser makes use of each SPE as stand-alone parsing engine that is implemented as a ZuXA XML parser. ZuXA is a state-machine technology [186, 187] based approach for XML parsing, which has a very small memory footprint and is thus ideally suited for the cell processor's SPEs, each of which has only 256KB of local memory.

In general, the sequential scanning of XML documents imposes a critical challenge for concurrently parsing. Amdahl's law suggests a high ratio of parsing/decoding time over XML scanning is needed to get reasonable speedups. In our earlier work on designing a table driven parser [208], the breakdown in scanning, parsing, and deserialization overhead with TDX parsing is reported and compared to other XML parsers. The analysis shows that scanning can be three times slower than parsing. From Amdahl's law we see that 14% speedup can be gained with two threads, and 23% with four threads. This observation suggests that parallel XML parsing may not be helpful for a single rooted XML document.

3.1.3 Differential Parsing

Differential XML parsing is based on the observation that typical XML processing systems do not expect to receive arbitrary XML. Rather, the common case is that incoming documents in a single use case follow some schema, either an explicit or an implicit one. A differential parser caches information on the XML documents it processes. When another document is then being processed, this cached information is used to efficiently process the parts in the document that match the stored information, leaving only the differences to be processed with the general system.

In [52], Takase et al. present a differential XML parser, called Deltarser. Deltarser optimizes XML parsing by reusing the parsing events stored in previous processed documents based on byte-level matching. It encodes each of byte sequences using a DFA and maintains processing contexts in the DFA states for partial parser of unmatched byte sequence. Each state transition in the DFA has a part of a byte sequence and its resultant parse event. It partially processes XML parsing only the parts that differ from the previously-processed documents. Each state of the DFA preserves a processing context required to parse the following byte sequences. It performs checking of well-formedness of the incoming XML documents without requiring analysis of the full XML syntax of those documents.

However, both the initial cost and runtime cost for constructing state transitions in Deltarser are expensive. To reduce the overhead, in their more recent paper [172], they propose an approach for optimizing the internal automaton by the use of static and dynamic information. The static information is obtained from an XML schema in a preprocessing stage and the dynamic information is gathered at runtime. An XML schema provides static information available before execution starts. The schema provides structural information and data types for its instances. By exploiting the schema information, an automaton is constructed in advance. As a result, the initial overhead of constructing an automaton is reduced at runtime. Furthermore, this static information can also be used to optimize the design of the state transitions. Dynamic information is obtained at runtime by aggregating a set of documents. Such information provides knowledge about the structural information and specific data types similar to the kind of information provided by the XML schema. Static information and dynamic information can be combined together or used in a complementary manner. Additionally, dynamic information can be used when a schema is not available.

Differential XML parsing is designed for efficiently processing XML documents similar to previously processed XML documents. Therefore, this technique is suitable for the environments like Web service application servers, where middleware needs to process many similar documents generated by other middleware or clients.

3.1.4 XML Hardware Acceleration

Hardware-based systems such as Datapower [110] benefit from running parser/validators in isolated systems where memory management, thread dispatching, etc. can be optimized. Some of these systems also include ASICs customized for XML processing, and most integrate support for security and other features not provided by XML Screamer. DataPower has also reported on the use of JIT-like virtual machine technology to optimize XML processing.

Efficient Accelerated String and Text Processing Architecture: Traditional microprocessors incorporate deep pipelines so multiple instructions can be assigned to different stages of the pipeline to improve computation throughput. All those tasks have low data dependency and require the same basic operations to be executed repeatedly on a large amount of data. However, the traditional architecture is not able to take advantage of such inherent parallelism for XML parsing due to the sequential nature of text-based XML processing.

Van Lunteren et al. [187] presents a programmable state machine technology, called B-FSM, which can be a key enabler for building a non-traditional processor architecture with dedicated instructions for character- and string-processing. Based on the B-FSM technology, a high-level concept of an XML accelerator engine, called ZuXA was introduced to demonstrate the performance improvement of XML parsing by proceeding a processing model that is optimized to B-FSM.

Intel SSE4 is a new set of Single Instruction Multiple Data (SIMD) instructions that improve the performance and energy efficiency of a broad range of applications. Intel SSE4.2/STTNI offers four "string and text processing" instructions, which can accelerate many string and text operations, typically, the functionalities of finding characters in a set, checking characters in a range set, comparing two strings and finding a substring in a string. The Intel XML Software Suite [55] achieves a great performance improvement by leveraging the four dedicated string and text processing instructions [100, 104].

XML Offload Engine: The basic idea of XML offload is to move typically computationally-intensive XML processing from stems that run the applications to other designated systems. Some efforts have been conducted to provide XML offload by the use of a dedicated XML hardware accelerator. The use of an XML offload engine is designed to take care of two issues: (1) freeing up some cycles from the host CPU that was previously used for XML parsing, and (2) using special purpose hardware to speed up certain parsing tasks.

Tarari RAX Content Processor (RAX-CP) is a hardware solution for XPath-based processing of XML data, having the form of PCI card and functioning as an XML co-processor [110]. At the heart of RAX is the use of XPaths to index into the document and/or to make assertions against it. The XPath statements are declared before the document is processed and grouped together. They are evaluated at the same time the document is parsed and the results of the XPaths may be used as indices to have direct access to the data of interest in the document. The XPaths are processed using XML chips. Since processing in silicon is very fast and since the processing is offloaded from the host CPU to the XML chip, RAX processing effectively takes place in near-zero CPU time. While RAX offers high performance for well optimized XPaths, some of XPaths such as statements with nested predicates, ancestor axes, and predicates that require type conversion, are hard to optimize.

Two XML acceleration appliances come from DataPower are called XA35 XML Accelerator and XS40 XML Security Gateway [163]. Whereas the XA35 handles XSLT transformations, the XS40 provides Web Services Security (WS-Security) support. Both appliances are network-attached hardware accelerators and can operate in co-processor or proxy mode. DataPower claims to deliver wire-speed functionality.

Letz et al. [107, 108] present XML offload acceleration using Cell technology. Cell is a new processor architecture with one Power Processor Element (PPE) running the operating system, and eight independent Synergistic Processing Elements (SPEs) running the application threads [152]. The high-performance XML parser is implemented based on the BaRT-based Finite State Machine (B-FSM) technology [186, 187].

3.1.5 Summary

In this section, a set of optimization techniques for parsing optimization on XML documents have been described. Among these different techniques, schema-specific parsing techniques achieve the most significant performance improvements by utilizing XML schema information attached. Schema-specific techniques can be combined with other techniques for further performance improvement. In addition, most schema-specific parsers presented in this chapter optimize XML processing by integrating parsing, validation, and even deserialization. Parallel XML parsing optimizations utilize the multi-core processors or other lower-level parallelism hardware. Parallel XML parsing techniques speed up the parsing efficiency by partitioning XML documents and processing these partitions simultaneously. However, there is a limitation for this approach due to the sequential property of partitioning and rooted architecture of XML. Specialized hardware acceleration achieves the best parsing performance by executing the XML processing using specialized hard ware, thus reducing the workload of CPUs. But requiring a special hardware may put hurdles on its usages. Differential parsing only applies to the applications where the continuous incoming document differs slightly such as Web service servers. This approach has not been studied intensively compared to other approaches.

3.2 Validation Optimizations

XML parsing allows for optional validation of an XML document against a DTD or an XML schema. Schema validation not only checks a document compliance with the schema but also determines type information for every node of the document. Applications that use the XML data often require the data to be modeled according to a schema for the sake of security. For example, it is critical for database systems and the XQuery language because they are sensitive to data types. Hence most processing of documents in a *data management context* not only requires parsing but also validation [137].

An XML document must be well-formed, and in most cases should be valid (conform to a specified document model) as well. The Extensible Markup Language (XML) 1.0 specification [205] explains the distinction between validating and non-validating XML processors in Section 5.1. In contrast to an XML ”well-formedness checker” or ”XML Syntax Checker,” an ”XML Validator” uses an XML validating processor to determine whether or not a candidate XML document conforms to its schema, expressed formally in the DTD (document type definition). To be a valid XML and not merely a well-formed XML, the XML document must satisfy the validity constraints expressed by

the declarations in the schema. Validation of XML documents helps ensure that the information is structured in a way that is sensible for applications which use it. XML documents that are merely well-formed can store any element inside of or adjacent to any other element whatever, which usually would not be very helpful.

XML validation, as performed by a *validating XML parser*, filters valid XML from invalid content based on a *schema*, e.g., XML schema, Relax-NG [54], or DTD. In the SOAP/XML Web services context where XML parsing is an essential component for message exchange, this simple principle guides the separation of concerns between message content verification and the service application logic.

Despite the key advantage of XML validation, it is often disabled for high-performance XML applications because it is well known that validation incurs significant processing overhead in the XML parser at the receiving server [117, 179]. Part of the overhead is caused by the checking of validation constraints in a separate stage after document parsing as described by the specifications [54, 195]. A validating parser must keep sufficient state information, or even the entire document in memory, to validate an inbound XML stream against a schema. Furthermore, the complexity of the validation process is exacerbated when streaming XML parsing requirements are considered, due to the inherent incompatibility of streaming with a staged validation process. Streaming techniques allow XML messages to be parsed and processed on the fly, eliminating the need to store a document in full, such as with DOM.

There have been different efforts aimed to improve XML validation performance. These techniques include variations of automata based validation, incremental validation, differential validation etc.

3.2.1 Automaton Based Approach

Using a Finite State Automaton (FSA) is a natural way for validating an XML document against a given XML schema. Reuter and Luttenberger [155] extend deterministic finite automata with cardinality constraints on state transitions that map naturally to the encoding of occurrence constraints. Unfortunately, a deterministic finite automata with cardinality does not perform well-formedness checking, but runs as a separate layer on top of a separate SAX parser, with the associated performance penalty. The major advantage is that these automata can easily take care of occurrences constraints imposed by schema. Unfortunately, this approach does not provide mechanisms for well-formedness checking. Similarly, Thompson et al. [179] present augmented finite sate automata constructed from XML Schema content models to perform schema validation (see 3.1).

To reduce both the memory and time cost, Segoufin et al. propose a finite automaton that is used to validate XML streaming data with respect to a non-recursive DTD under memory constraints [161]. They present an algorithm for constructing a finite automaton for any given nonrecursive DTD. Such an automaton is able to perform well-formedness checking and validation verification in a single pass, called *strong validation*, to distinguish it from sole validation. Under a restrictive assumption that the input stream is well-formed, the finite automaton can even be used to perform validation of XML streaming data with respect to a subclass of recursive DTDs. For example, the DTD $D = (r \rightarrow a, a \rightarrow a|\epsilon)$ can be validated using a finite automaton if and only if the input XML stream is well-formed.

Chitic et al. present a one-counter automaton for strong validation of XML data [47]. A one-counter automaton consists of a finite automaton and a counter that can hold any nonnegative integer and can only test if the counter is zero or not. The move of the automaton depends on the current

state, input symbol and whether the counter is zero or not. Compared to automaton generated using the algorithm in [161], this approach has two advantages. First, the one-counter automaton is deterministic; second, it recognizes a larger class language called *one-counter language*. For instance, the one-counter automaton can perform strong validation of the XML data with respect to the same DTD $D = (r \rightarrow a, a \rightarrow a|\epsilon)$ without the requirement of well-formedness. However, as a single counter variable is used, only a subset of DTDs can be validated. An example for a language that cannot be validated is $a^n b^m \bar{b}^m \bar{a}^n$ where a, b are opening tags and $\bar{a}$, $\bar{b}$ are their corresponding closing tags.

The major advantage of the automaton based XML streaming validation is that it does not require the whole XML document to be loaded to main memory. Both the automata presented here use a fixed size of memory and the size of the memory used is independent of XML document size. This makes it suitable for the online validation, very large XML document validation, in particular for the computing resource limited devices such as mobile computing. However, on the other hand, its limited recolonization power that it may hinder its application domains. Furthermore, the memory efficiency cannot be retained when validation against the W3C XML Schema in which validation of some of the components like `xsd:all` requires a combinatorial increase of the number of automata states.

3.2.2 Push-Down Automaton Based Approach

In [161], in addition to the finite state automaton for validating XML streaming data with respect to nonrecursive DTDs and a subclass recursive DTDs, the authors also proposes a depth-synchronous push-down automaton (PDA) for XML validation. It is demonstrated that every DTD can be strongly validated (well-formed parsing and validation) by a deterministic PDA with respect to a DTD. When the input XML streaming is well-balanced, the stack of the PDA is bounded in the depth of the tree representing the input string.

Hammerschmidt et al. [88] propose an approach to construct a PDA from an XML schema for incrementally validating XML data using an XML streaming validator. The XML instance of the schema is represented as a stream of opening and closing tags with their content. The PDA is constructed from an XML schema. The stack's content of the PDA encodes the path to the current element tag. The states of the stack representing the current position in the document model are pushed on or popped off the stack when the current tag is an opening tag or closing tag respectively. The PDA verifies the content models of the element types of the regular tree grammar by simulating the nested execution of the finite state automata (FSAs) that recognize the regular tree grammar generated from the schema. The transitions capture the valid tag sequences, i.e. the structural constraints imposed by the XML schema. The difference between the PDA [88] and the one in [161] is that it supports incremental validation that is faster and more memory efficient through the use of an index structure. It performs validation locally that is much more efficient than validating from scratch when a node is updated. In this case, the stack of the PDA is bounded only to the depth of the modification. Another difference is that it is constructed from an XML schema instead of from a DTD.

Kumar et al. [98] propose visibly pushdown automata (VPA) where the input determines the stack operation, and XML documents are pushed onto the stack on open-tags and popped off the stack on close-tags. This approach is a suitable model for processing streaming XML. Unlike the pushdown machines constructed in [161], VPAs reduce the number of states by combining structural conditions shared with the same tag, and across different tags.

3.2.3 Integrated Approach

Integrated validation improves validation performance via across the layers of the XML processing model. Validation is typically performed in a separate phase that sits on top of the generic XML parser module. This overhead can be eliminated by integrating parsing and validation (structural and data type and value checking) into an integrated XML parser and validator [51, 49, 52, 117, 182, 184, 208, 209]. These approaches construct specialized APIs, data structures or internal tables from a given set of schemas. These techniques are introduced in Section 3.1.

3.2.4 Incremental Validation

Potential validity checking enforces relaxed schema constraints for XML documents. Incremental validation of XML documents is a computational time-saving solution for validation of document updates, an efficient alternative to wheel document validation. However, incremental validation considers an initially validated document together with a set of update operations and checks the validity of the resulting document.

When an XML document is updated, it has to be verified that the new document still satisfies its type. Doing this efficiently is a challenging problem that is critical to many applications. Brute-force validation from scratch is inefficient and often not practical, because it requires reading and validating the entire database/document following each update. Instead, it is desirable to develop algorithms for incremental validation.

Incremental validation of XML documents is a computational time-saving solution for validation of document updates, an efficient alternative for whole document validation. Incremental validation refers to partial validation that only the modified part is validated in its local context. Given a schema and a sequence of updates applied to the document, the incremental validation problem is to determine whether or not the modified document is still valid with respect to the original schema. The goal of an incremental validation method is to perform such verification without checking the entire document, but just the part of of it which is concerned by the update. Incremental validation has been studied intensively in many works.

Some algorithms maintain state information with each node in the document that is used to validate a document incrementally [31, 17, 21]. In general, the amount of auxiliary states stored with a document can be quite large. When the schema in question uses a restricted language of regular expressions for content model, for example, conflict-free [21] or local [17] regular expressions, optimizations can be applied to improve the efficiency of incremental validation.

A multitude of approaches for partial validation of XML data after updates have been presented. All these approaches assume that the XML data is formed according to the DOM model, i.e. it is presented as a tree of nodes. This representation is inherently well-formed and enables the direct and efficient navigation within the nodes. For instance, it is possible to access all children of a given node. The DOM model works well for native XML database management systems where the tree-like representation is preserved. In contrast, [88] focuses on the string representation of XML data as it is used in XML column types, messages system, or SQL-XML update commands. In those systems, the XML data is represented as a sequence of tags and values. The sequence can be seen as the result of a pre-order traversal of the corresponding XML tree.

3.2.5 Push-Down Automaton Based Approach

Hammerschmidt et al. present a schema-aware Push-Down Automaton (PDA) that incrementally validates XML data [88]. The PDA is constructed directly from a Regular Tree Grammar (RTG) [133] that is generated from an XML schema. The XML instance of the schema is represented as a sequence of opening and closing tags with their content. The stack's content of the PDA encodes the path to the current element tag. The PDA states that represent the current position in the document model are pushed on or popped off the stack when the current tag is an opening tag or closing tag respectively. The PDA verifies the content models of the element types of the regular tree grammar by simulating the nested execution of the Finite State Machines (FSMs) that recognize the regular tree grammar generated from the schema. The transitions capture the valid tag sequences, i.e. the structural constraints imposed by the XML schema.

Partial validation is achieved through the use of an index table that enables PDA directly access the start element for revalidation and its corresponding PDA states. Each entry of the index table contains a key represented as simple path expression, offset from the current position to the key, a state associated with the key, and stack content associated with the key. Once a node is updated, the PDA locates the start element, restores the PDA state and stack content by consulting the index table.

This PDA based approach implements an efficient incremental validation for XML stream after local updates, because the validating automaton is schema-aware and is created only once. The validation time is linear in the size of the updated XML data for most of the cases. Schema violations are detected early as no transition can found for the next (invalid) tag. This approach is also memory efficient. Unlike DOM-based approaches, it does not require loading the full XML data into main memory. The only memory is a stack that may grow linearly in the depth of the XML data (in the worst case, the stack is the size of the XML data). In addition, this approach supports recursive schemas.

3.2.6 Logic Based Approach

Benedikt et al. propose a logic based approach to constraint checking in an updating environment [23]. A constraint language, called $\triangle X$, which is designed to support incremental validation is introduced. The incremental validation is done by translating constraints into logic formulas and then generating actual constraint checking code from these logic formulas. The $\triangle X$ language is based on XPath and an XML schema and enables constraints to be expressed by *value-based* expressions. A code generator, $\triangle X$ compiler, is developed for automatically generating incremental constraint-checking code from the logic representation that is more amenable to algorithm manipulation. The logic can express both the constraints and the parameterized constraints (formulas) that describe the incremental check for each update operation. However, this approach does not take into account schema constraints.

3.2.7 Query Based Approach

Kane et al. [94] present a technique based on query modification for handling incremental validation. The key idea is to turn an XQuery [197] engine that supports updates into incremental constraint-check engine by embedding constraint check sub-queries into a safe XQuery update statement. A safe XQuery statement ensures that an update to an XML document is performed only if the updated data will still conform to the given schema. This is enabled by the capability that

not only can the XQuery language query XML data but also XML schema because XML schema itself is expressed in XML format. This loosely-coupled incremental constraint check approach provides efficient mechanism for maintaining the structural consistency of the XML document with its associated XML schemas during an update. The traditional approach first loads and updates the XML document before constraints are checked, and second the update is verified against associated schemas, and rollbacks may be needed if the updated data no longer conform to the schemas. This query based approach is more efficient than the traditional first-change-document and then re-validate-it approach. In this approach validation is performed before an update is executed, therefore no loading data into memory or rollback operations are needed.

Though this approach is based on XQuery language, the query based technique can be extended to any XML query language such XPath. Query based incremental validation technique offers significant benefits for native XML support database management system (DBMS) and other applications such as XML editors that support query features. However, this approach sits on top of the XQuery engine that my be the performance bottleneck. To this end, this can be teated as an implementation rather than a validation technique.

3.2.8 Schema Based Differential Re-validation

Raghavachari and Shmueli [154] present a technique that makes use of information on validating an XML fragment with respect to one schema to improve the performance of validation of that XML fragment with respect to another schema. This is achieved by exploiting similarities and differences between the two schemas to avoid validating portions of a document. However, when the reduction of storage of size of a document is essential, for example, with main-memory XML processors, our algorithm may offer an appropriate solution for the incremental validators. Furthermore, this approach offers validation in general and is capable of being applied to other schemas.

The validation of streaming XML data wrt. DTDs has been investigated in [161]. They show that validation is possible by a depth-synchronous push-down automaton. It can be done even by a finite automaton if and only if the DTD is non-recursive, i.e., no element can occur inside an element with the same name, guaranteeing documents of bounded depth. A refined setting is to take as granted that documents are well-structured (i.e., opening and closing tags are nested correctly). In [161] a condition is presented that guarantees that a DTD can be validated by a finite automaton (under the well-formedness condition). It is conjectured that this condition is also sufficient and that the automaton construction given works in all cases. Validation for non-recursive one-unambiguous (and more generally, k -unambiguous) DTDs by finite document automata with an emphasis on the size of the resulting automata has been studied in [47].

3.2.9 Summary

In this section, we presented a variety of techniques that optimize XML validation. Automaton-based validation and incremental validation have been addressed intensively. Automaton is a natural way to represent and validate XML documents in particular for validation against DTDs. The automaton based validation techniques described in this chapter improve the validation performance by utilizing XML schemas or by constructing a variant automaton. These techniques reduce the memory requirement and have been successfully applied to XML streaming. Because the schema-specific techniques integrates parsing and validating, the validation for value constraints can be optimized by exploiting XML schemas, thus achieving high performance. Various techniques have

also been proposed for efficient incremental validation of XML. Incremental validation plays a important role in scenarios that require frequent update operations such as insertion or deletion of a node to or from XML documents (or native XML database systems).

3.3 Deserialization Optimizations

3.3.1 Static-dynamic Approach

Van Engelen et al. [181, 185] introduce a static-dynamic approach to XML deserialization/serialization. Static analysis is used to build a plausible data model from XML schemas or a WSDL description at compile time. The model represents the possible in-memory objects of the XML/SOAP instances. The model is then used to generate type-specific deserialization/serialization algorithms that parse the XML instances at runtime to effectively deserialize them to in-memory language specific objects or serialize the in-memory objects into XML document, using an XML schema to C/C++ mapping. As a result, generated deserialization/serialization routines are specialized and optimized to the data model. These routines are precompiled to minimize dynamic type inspection at runtime.

To serialize a graph data structure to ensure object-level coherence, the static-dynamic approach makes use of optimized XML data representations using schema extensibility. The SOAP specification allows the use of "multi-ref accessors" to refer to previously (de)serialized instances of specific elements of the SOAP call. The generated deserialization routines decode the contents to reconstruct the original data structure graph. When the data structure is constructed, temporarily unresolved forward references are kept in a hash table. When the target objects of these references have been parsed and the data is allocated in memory, the unresolved references are replaced by pointers. That is, the unresolved pointer are back-patched with pointer values to link the separate parts of the graph structure together.

3.3.2 Differential Deserialization

Differential Deserialization (DDS) is a caching strategy that is used to reduce deserialization cost of XML processing, in particularly in the Web services where the server may process many similar documents generated by other middleware or clients. The basic idea of differential deserialization is to avoid XML parsing and construction of in-memory application objects for unchanged or similar portions between incoming messages. DDS technique exploits similarities between incoming SOAP/XML messages to limit deserialization to application objects in memory that are different within a single message or across multiple messages. This is accomplished by comparing checksums of parts of the inbound message against a cache of previously deserialized objects and their checksums [3, 6, 4, 5], or by checking the state transition path of an automaton created by a previous deserialization pass [173]. When a checksum or an automaton's state transition path matches, the deserialized object is retrieved without full deserialization. As a result, only a portion of the document is deserialized. Ideally, an XML message that is identical to a previous message does not require deserialization.

Transition Path Based Differential Deserialization: Suzumura et al. [173] describe a DDS technique for optimizing Web services performance that is based on the transition path of an automaton. The automaton efficiently compares byte sequences with those from previously processed

messages. This approach optimizes processing time by recycling in-memory objects without deserialization of similar messages. Similar messages are determined by checking the state transition path of an automaton created by a previous deserialization pass. It is based on their Deltarser [175] that optimizes XML parsing by reusing the parsing events stored in previous processed documents based on byte-level matching. Deltarser encodes each of byte sequences using a DFA and maintains processing contexts in the DFA states for a partial parse of an unmatched byte sequence. As a result, only the differential parts from the previously processed documents are processed.

At the heart of the differential deserializer is the Deltaser-based matching machine. This matching machine dynamically creates an automaton on the one hand for the incoming XML messages and serializes these messages into in-memory application objects. These objects are stored in an object repository for reuse. On the other hand, it checks to see if the incoming message matches one of the existing automaton paths and returns a corresponding object from the object repository to the application without full deserialization of these similar messages.

Specifically, the deltarser's DFA states are classified into *fixed states* and *variable states*. A fixed state is a DFA state whose byte sequence is not changed from message to message such as element tag names and attribute names. Fixed states correspond to an XML structure, as indicates by an XML schema. A variable state is a DFA state whose sequence may vary from different messages, for example, the element character data values.

In order to recycle the object, the deserializer maintains a variable table with an entry for each variable state. The variable table is created when a message is first deserialized and is associated with the final state of the DFA, and it is updated accordingly for subsequence messages. Each entry of the variable table stores the following fields: a *variable ID*, an *object parent*, a *class type*, an *object value*, and an optional *method setter*. The variable ID is a key that is used to index the table for updating an entry. It is not clear how this key is implemented and whether or not it is explicitly stored with each variable state. The object parent is the object that contains the actual value of or the reference to the object corresponding to the variable state. The class type field identifies the type of the object value to the appropriate object. The object value field holds the latest value of the object corresponding to the variable state. Finally, the method setter is a reference to an object with a method that is used for setting the value to the object.

When a similar message is identified, the deserializer traverses the existing deserialization automaton to check the state transition path. During this traversal the byte sequence is partially parsed and deserialized until the next state is met. The deserializer then updates the value in the corresponding entry identified by the variable ID in the variable table for the variable state, because the value may likely be changed in the subsequent messages. After it reaches an accept state, the deserializer resets the object's variable fields according to the corresponding values in the variable table, and then returns the object for recycling.

Checkpointing Based Differential Deserialization: Abu-Ghazaleh et al. [3, 6, 4, 5] present techniques for differential deserialization using checkpointing. Such a deserializer runs in two modes: *regular mode* and *fast mode*, and switches between these two modes from each other as appropriate, while processing the message. In the former mode, the deserializer reads and processes XML tags and message contents, creates checkpoints, calculates checksums of the message portions, and saves parsing states. In fast mode, the deserializer considers the XML stream as a sequence of checksums, and compares them with the sequence of checksums associated with the most recently received message.

However, the deserializer can not determine if it is safe to avoid deserialization processing

without checking namespace bindings even if all the checksums match those ones already saved in previous messages processing. Consider for example, a namespace alias may be used to qualify XML tag names in two literal identical XML messages. As a result, the checksums of these two messages are the same. However, the second message can not be treated as the same as the first one because the tag names may be under different namespace bindings. To be able to perform the namespaces binding check, the deserializer maintains a namespaces aliases stack for each checkpoint in addition to the checksums. When a checksum mismatch is encountered or namespace bindings are different, the deserializer signals a difference in the incoming message from the corresponding portion of the previous message. When this happens, the deserializer switches from fast mode to regular mode. The deserializer also maintains several other stacks at each checkpoint to store the necessary information for the deserializer to encode deserialization states. During the regular mode, the deserializer safely switches to the fast mode when it recognizes that its parsing state is the same as the one already saved in a checkpoint as well as the namespace aliases mappings also match.

Checkpointing based DDS can be optimized by *differential checkpointing* (DCP) [3] and *lightweight checkpointing* (LCP) [5] techniques.

- **Differential Checkpointing:** A differential checkpointing technique stores differences between two consecutive checkpoints associated with different portions of the message, rather than storing a complete copy of each stack with each checkpoint regardless of the fact that some stacks may remain unchanged or have small changes between two consecutive checkpoints.

  Differential checkpointing is based on the observation that successive parsing states will differ only by the changes of subsequent message portions. When the message portion is small, or when the processing of that portion does not result in significant changes to the states of the parsing or deserializing, then the memory requirements can be significantly reduced.

  More specifically, regular checkpointing DDS maintains several stacks to store the parsing states, including a string reference stack, a strings stack, a matching stack, and a stack for namespace aliases. Therefore, differential checkpointing DDS involves stores and tracks these differences in these stacks to speedup the performance and reduce the memory requirements. To this end, differential checkpointing DDS must start to process from the most recently stored complete parsing state, and then applies the differences defined in the differential points.

  Differential checkpointing has two sources of overhead: the overhead of detecting changes in the states of deserializer that have occurred since previous checkpoints were created, and the overhead of tracking down and manipulating partial state information while restoring or comparing state. In most cases this overhead is smaller than the cost that will occur from manipulating larger data, as results, the differential checkpointing DDS leads to performance benefit gains, though the strength of this approach is efficient memory requirements.

  The major benefit of differential checkpointing is in the reduction of the amount of state manipulation resultng from the fact that the less data is copied after reading or restoring a checkpoint and less data is compared for switching to fast mode.

- **Lightweight Checkpointing:** Lightweight checkpointing optimizes DDS performance and reduces memory requirements by performing checkpoints at *strategic points* where changes of states of deserializer are minimal and can be statically predicted. This is achieved by exploiting the hierarchical structure of SOAP/XML messages by leveraging Schema information.

Lightweight checkpointing DDS maintains very little state information but can only be created at predefined points within the structure of message. By exploiting the schema information, the lightweight checkpointing DDS determines these predefined points where checkpoints can be created to minimizes the state information. Each lightweight checkpoint has a reference to a *base checkpoint* that contains state information it shares with other lightweight checkpoints. That is, the full state of the deserializer is defined by the combined states of the lightweight checkpoint and its base checkpoint.

Lightweight checkpointing and differential checkpointing share the fact that changes of states are saved with consecutive checkpoints, unlike the regular DDS that stores the complete state information with each checkpoint. However, unlike the differential checkpoint DDS may create checkpoint at arbitrary checkpoint boundaries, lightweight checkpointing uses strategies to create checkpoints only at the points where the deserializer state changes can be minimized. Lightweight checkpointing is typically more efficient than differential checkpointing because it does not require detecting state changes because most states are known before deserializing the message. This also makes the lightweight checkpointing DDS switch to fast mode more efficient. It is also more efficient in terms of memory usage than differential checkpointing because the amount of states that need to be stored is minimized.

However, the performance benefit of lightweight checkpointing comes at the price of flexibility. Unlike full or differential checkpointing techniques, lightweight checkpointing technique can not create checkpoints at arbitrary locations in the message.

3.3.3 Summary

To best of my knowledge, XML deserialization has not been addressed intensively. In this section, we investigated two techniques for XML deserialization optimization: Differential Deserialization (DDS) and static-dynamic deserialization.

The effectiveness of DDS depends on the degree of similarity between previously-deserialized messages and the incoming ones, as well as the overhead associated with the techniques underlying differential deserialization. DDS is in particular suitable for XML Web services because in a Web service scenario the server can expect to receive a large number of similar SOAP request messages from an individual client of different clients.

The performance of deserializing scientific XML data can be significantly improved with DDS because converting the text format of data to in-memory binary representation can be very expensive. Chiu et al. [51] report that converting scientific data can account for 90% of end-to-end time in a SOAP message exchange containing scientific data. DDS can efficiently avoid the need of reconverting unchanged data for such messages.

One major drawback of DDS is its application limitation. For XML messages that contain few repeated or similar elements, the performance gains from the differential XML processing technology can be easily overwhelmed by the overhead incurred the underlying differential technology. In worst case, when the message contains no similar elements, the performance of deserialization with DDS is slower than deserialization without using DDS.

Another major disadvantage of DDS is its inefficiency of memory requirements. Although there are some mechanisms to reduce the memory requirements, DDS is essentially a caching based technology that is applied to a category of XML messages that contain repeated or similar elements for performance optimization. As a result, it essentially requires to store much state information and

many XML message sequences. This memory intensive property of DDS may hinder its acceptance in resource limited web services such as mobile computing.

Unlike DDS, the static-dynamic approach is a generic deserialization technique for optimizing XML validation by mapping XML schema to programming languages data types. This approach can be applied to both deserialization and serialization, and it has been widely adopted along with the distribution of gSOAP toolkit.

3.4 XML Searching Optimizations

As XML have become the *de facto* standard of messages transformation and storage, it is crucial to efficiently extract nodes and contents from XML messages. XPath [193] was introduced by the W3C as a standard language for specifying node selection, matching conditions, and for computing values from an XML document. It is used in many XML standards such as XSLT [53] and lies at the core of the XQuery [197] database access language for XML messages. Since efficient XML content querying is crucial for the performance of almost all XML processing architectures, a growing need for studying high performance XPath-based querying has emerged. The most fundamental algorithmic question concerning XPath is query evaluation. That is, given an XPath query Q and an XML database D, return all elements that are selected by Q in D. XML query evaluation typically requires exponential time in the size of queries in the worst case. The query evaluation problem for various fragments of XPath has been researched quite intensively over the last decade (see [24] for an overview). Also different techniques for boosting XML searching on XML documents and native XML databases against XQuery language have been studied. Traditional XML searching implementations on XML messages reflects the XPath specification or XQuery language and usually incur exponential run-time complexity. The first polynomial-time algorithms for XPath processing was proposed by the authors in [74]. Evaluating XPath queries efficiently is especially essential to the effectiveness and real-world impact of these techniques [75]. In this chapter, the techniques used for efficient XML query evaluation are investigated.

3.4.1 Automata Based Approach

Automata based approach is one commonly used technique for evaluating XPath expressions on XML data streams. In these approaches, simple XPath expressions are transformed into automata representations. Elements of a path expression are mapped to states. A transition from an active state is triggered when an element if found in the document that matches that transition. If an accept state is reached, then the document is said to satisfy the query. Automata based approaches have been extensively studied for efficient XML query evaluation [48, 59, 78, 87, 91, 141, 148, 165, 166].

In [11], Altinel and Franklin present an XML filter based on a finite state machine (FSM), called XFilter, to efficiently filter XML documents for selective dissemination of information. By converting queries into a finite state machine representation and associating a sophisticated indexing structure to the finite state machine, XFilter is capable of efficiently checking element ordering and evaluating queries. XFilter builds a finite state machine for each query expression and employs a query index on all the finite state automata to process all queries simultaneously. Specifically, query expressions are converted to FSMs by mapping location steps of the query expressions to machine states. Arriving XML documents are then parsed with an event-based (SAX) parser, and the events raised during parsing are used to drive the FSMs through their various transitions. A query is determined to match

a document if an accepting state for that query is reached. A separate FSM is created for each distinct path expression and a sophisticated indexing scheme is used during processing to locate potentially relevant machines and to execute those machines simultaneously. The indexing scheme and several optimizations provide a substantial performance improvement over naive approaches. The drawback, however, is that by creating a separate FSM for each distinct query, XFilter does not leverage commonality among the query expressions, and as a result, redundant evaluation may be incurred.

XTrie [40] is designed to support large-scale filtering of streaming XML data. Unlike XFilter, XTrie supports complex XPath queries with predicates. Its basic idea is to use the trie to detect occurrences of substring matches for each event that it receives.

Motivated by the XFilter, Chen et al. present an YFilter [58, 59] where nondeterministic finite automaton (NFA) based execution model is used to represent multiple queries. The NFA combines all queries into a single machine. YFilter exploits commonality among path queries by merging the common prefixes of the paths so that they are processed at most once. The resulting shared processing provides tremendous improvements in structure matching performance over algorithms that do not share such processing or exploit sharing to a more limited extent. This NFA based implementation also provides additional benefits including a relatively small number of machine states, incremental machine construction, and ease of maintenance. DFA based approaches are usually more efficient than NFA based approaches for processing XPath expressions in terms of CPU processing time. However, when the number of XPath expressions is large, converting all the XPath expressions into a single DFA may become impossible, because the size of the DFA grows exponentially with that number. In [87], Gupta and Suciu propose a lazy DFA, called the XPush machine, which can efficiently evaluate a large number of XPath filters on a stream of XML documents. The XPush machine is constructed by creating an Alternating Finite Automaton (AFA) for each filter, and then transforming the set of AFAs into a single Deterministic Pushdown Automaton (DPDA). In order to prevent the exponential blowup of the number of states, the transformation into a DPDA is postponed until the XML data is evaluated. Very similar to Gupta and Suciu's approach, lazy-DFA is proposed by Green et al. [78] to reduce the number of machine states. In this lazy-DFA approach, all path expressions are combined into a single DFA. In this approach, the query trees are first converted into an NFA, and the NFA is converted into a DFA. The DFA is constructed lazily to prevent the blowup of the number of states, i.e., similar as in the Gupta and Suciu's approach. These lazy DFA-based approaches result in better performance than the NFA-based ones while avoiding an exponential growth of the number of states.

Filtering the XML stream means to send to output only those (parts of) data items from the stream that match the filtering conditions, e.g., XPath expression. In case that data transformation is needed before/after filtering, XQuery is more suitable than XPath. In the following paragraph, we present current results on XPath and XQuery processing on XML data streams.

In addition to finite state machines, some works present tree automata-based XPath query evaluation techniques [39, 69, 96]. In [96], Koch presents a nondeterministic tree automata for efficiently evaluating node-selecting queries over XML trees. This approach is based on selecting-tree automata [69], which only requires a single nondeterministic bottom-up traversal of the tree to compute the result of a node-selecting query. One of the major drawbacks of this approach is that it requires two linear passes over the XML document.

Carme et al. [39] present an approach for efficient query evaluation on unranked XML trees using stepwise trees automata. In this approach, an XML document is represented as an unranked

tree, and the query expression is converted into a stepwise tree automata representation. Stepwise tree automata are traditional tree automata that can either operate on unranked or ranked trees. They combine the advantages of selection and hedge automata. Furthermore, the algebraic approach behind stepwise tree automata renders a new systematic correspondence between queries for unranked and ranked trees.

Efficiency is the major advantage of the automata based approach. However, these techniques exhibit memory inefficiency due the large number of machine states. Also, all of these systems are limited to handling location path expressions that only contain forward axes. Additionally, some of the XPath features, for example, the arbitrary position predicate function `position()=500`, cannot be presented efficiently using automata due to the expressive power of automata. Furthermore, these implementations typically sit on top of SAX parser, thus introducing an extra processing layer of the query processor.

3.4.2 Push-down Automaton Approach

XPush [87], unlike the automata approach, translates the collection of filter statements into a single deterministic pushdown automaton. The *eager* XPush machine still needs an exponential number of states. However, a lazy implementation, which delays the discovery of the states and avoids redundant state enumeration, is proposed to reduce the number of states.

SPEX [34, 138] uses a network of transducers to evaluate regular path expressions with XPath-like qualifiers. For representing the transducer stack, SPEX needs memory quadratic in stream depth. Like SPEX, XSQ [149] uses a push-down transducer based approach for XPath filtering, where stacks keep track of matching begin and end tags and buffers store potential results to compute predicates.

3.4.3 Stack Based Approach

Canda et al. propose a stack based approach called an adaptable filter [41, 38]. This approach is composed of three components: a linear size data structure to index registered pattern expressions; a runtime stack to represent the current data branch; and a cache, which stores prefix sub-matches for re-use. AFilter has a base memory requirement linear in filter expression and data size. In addition, when additional memory is available, AFilter can exploit prefix commonalities in the set of filter expressions using a loosely-coupled prefix caching mechanism as opposed to tightly-coupled active state representation of alternative approaches. AFilter can also exploit suffix-commonalities across filter expressions, while simultaneously leveraging the prefix-commonalities through the cache.

3.4.4 Directed Acyclic Graph Based Approach

Barton et al. [20] present a streaming algorithm for evaluating XPath expressions that use backward axes (parent and ancestor) and forward axes in a single document-order traversal of an XML document. In this approach, an XPath expression is first represented as a tree structure and a directed acyclic graph (DAG) is then constructed from the tree structure by reformulating the ancestor and parent constraints in the tree as descendant and child constraints. This approach extends the ability to process both forward and backward axes of XPath expressions. A matching algorithm is presented to find the matched patterns from the DAG representation.

3.4.5 Parallel XML Query Optimizations

Parallelization of SQL queries has been extensively studied in the context of both distributed and centralized repositories. However, these parallelization technologies can not be straightforwardly applied to XML queries directly. Parallelization of SQL queries differs from the XPath parallelization in many ways, for instance, the relational data has a regular 2-D structure that is suitable for partitioning either along rows or along columns. The rooted hierarchical structure of XML is not inherently suitable for balanced data partitioning. Although XPath's typical execution model is inherently sequential, i.e., each location step operates on the nodes that are returned as a result of the previous location step or the starting context, the execution of a location step can be reordered as long as there are no intra-step dependencies. This provides opportunities for parallelization of XPath queries. A given XML query can be split into different independent sub-queries executed in parallel, and the final results can be obtained by either merging or performing a union operation on the temporary results. Another common paralleling approach is to perform the given query on different partitioned XML datasets using XML partitioning techniques.

XML Query Parallelization using Multi-core Processors: Leveraging the parallelism provided by multi-core resources to speedup software execution is becoming the trend of the software development. Several papers proposed algorithms for parallelization of XML query evaluation on multi-core processors. Bordawekar et al. [29] propose three techniques for parallelizing a single XPath query in a shared-address space multi-core environment: Data participating, Query participating, and Hybrid (Data and Query) participating. Parallelism is achieved through the partitioning traversals over the XML document. In the data partitioning approach, all participating processors shares the same query. A designated single processor performs the serial part of the XPath query on the entire document and assigns datasets to each participating processor, which executes the same XPath query on assigned datasets concurrently. The designated processor is responsible for performing the merging operation to return the final result. Parallelism is achieved by executing the same XPath query concurrently on different section of the XML document. In query participating approach, the original XPath is broken into a set of sub-queries that can ideally navigate different sections of the XML tree. To take advantage of all the processors, the number of sub-queries matches the number of participating processors. Each processor executes its assigned sub-queries on the entire XML document. Unlike the data participating, this approach achieves parallelization via exploiting potentially non-overlapping navigation patterns of the sub-queries. Hybrid participating combines data participating and query participating. In this approach, the input XPath query is first split into different sub-queries for a set of virtual processors. Each virtual processor is a set of physical processors and it performs its assigned sub-query using the data partitioning approach.

Unlike the data partitioning technique introduced in [29], Liu et al. [115] present a different data partitioning technique to parallelizing the XML query evaluation on a shared-memory space multi-core environment. In this data partitioning approach, an XML document is represented as an indexed DOM tree. From the indexed DOM tree, a list of ancestor elements and a list of descendent elements are partitioned and stored into a set of buckets. These buckets are then evenly assigned to all partitioning processors, each of which executes the given XPath query on the assigned buckets. The final result is computed using joint merging operation.

XML Query Parallelization on Distributed Environments: To achieve efficient parallel execution and local work balance on a shared-nothing architecture, for example, a cluster system, a critical issue is distributing XML datasets and balancing workload among all participating processors. Li et al. [111] present a data distribution methods on a cluster environment for evaluation of XQuery.

The basic idea is to first distribute XML data as evenly as possible among all the participating nodes. Each node performs query evaluations on its local datasets. Local outputs from each node are then broadcast to all other participating nodes using MPI calls, and the final result is merged based on the received local outputs. The distribution method is based on the technique by Marian and Simeon [123] to distribute the XML data among all the participating nodes. To balance the workload to each of the participating nodes, the original XML data is partitioned into the number of XML documents so that the number matches th number of participating nodes. Each document contains approximately the same number of elements. This is accomplished by the use of a common prefix among the set of paths as distribution criteria. Because this approach requires broadcasting local results to all other participating nodes, this may become the bottleneck of the performance, in particular in a cluster environment where nodes are connected via network and some of the nodes may be distributed. Additionally, to "evenly" distribute XML data among all the participating nodes, it is not uncommon to result in duplicated nodes to different XML documents and, it thus requires that multiple occurrences must be defined in the XML schema for the final result merging operation.

Machdi et al. [122] propose static and dynamic data partitioning strategies among cluster nodes for parallelizing XPath query processing. In this approach, a processing node is designated as a coordinator that determines the processing plan and distribute it among the processing nodes. Processing nodes maintain their own XML datasets resulted from XML data partitioning. Unlike the technique in [111], this approach requires no communication among the processing nodes. The communication occurs only between the coordinator and the processing nodes when the coordinator sends queries to the processing nodes and receives temporary results from them. They propose XML data partitioning strategies to efficiently address the workload imbalance by making use of the static and dynamic data distributions. The basic static data partitioning method is basically derived from GMX [121]. They propose grid-based and stream-based partitioning techniques for data distribution in a more balanced workloads fashion. In static data distribution, the XML data is partitioned and distributed onto the participating nodes by the coordinator in advance before the queries are executed. The data partitioning strategy used is the combination of the grid-based partitioning method and the stream-based partitioning method. The former is mainly used and the latter is only utilized only if a relatively great amount of imbalance of workload is present. The dynamic data distribution performs XML data partitioning in a fine granularity while queries are being executed in the system and redistributes the partitions onto unloaded cluster nodes. It improves the overall performance by rebalancing the workloads of query processing if low system utilization is present. The rebalancing strategy makes use of the stream-based partitioning method.

The major XML parallelization challenges are data partitioning strategy, load balance, and merging cost. Data partitioning plays a key role to achieving scalable performance from a parallel application. The ideal data partitions results in partitioning processors performing equal amounts of work. However, it is often, in practice, difficult to achieve load balance. As aforementioned in this chapter, workload balance can be improved by some data partitioning techniques. Dynamic data partitioning technique can further be applied to increase the scalability of workloads balance. By leveraging multi-core processors, or by designing an efficient parallel processing model (e.g., designating a coordinator) on distributed environments, communication cost can also be reduced. Another approach for parallelization of an XML query is achieved by partitioning the given query expression into sub-queries and executing these sub-queries in parallel.

It is worth mentioning that partitioning of an XML document is inherently sequential. Thus the partitioning may become the performance bottleneck. The structure of an XML document determines

the scalability of the XML query parallelization. For example, an XML document that consists of a large number of arrays of scientific data, may be relative easy for partitioning, while an XML that has a deep depth of nested children elements may lead to a high cost of partitioning and incur significant communication cost for the final result merging operation.

3.4.6 Schema Based Query Optimizations

An XML schema carries a great deal of information that may help improve XML query processing. In practice, most industrial, academic and scientific XML datasets often have predefined schemas. One can even be generated by employing schema inference technologies on XML documents [66] when a schema is not available. Many works explore opportunities for schema information to optimize XML query processing. The basic idea of schema-based query optimization techniques is that an XPath expression (or an XQuery expression) is replaced with a semantic equivalent but lower-cost one. For example, an XPath expression may contain redundant operators and cost expensive axis such as wildcards ($\star$) and the descendent operators ($//$). The redundant operators can be eliminated and the wildcards ($\star$)and descendent operators ($//$) can be extended to lower-cost alternate paths through exploring schema information.

The first query pruning technique is introduced by Fernández and Suciu [63] who used graph schemas to optimize regular path expressions. This technique takes a user-defined query represented as a regular path expression and a graph schema as input, and constructs the product of automaton of two NFAs: one that accepts the paths denoted by the query and another that accepts the paths denoted by the graph schema. A pruned query is then constructed from the product NFA by using only those paths in the NFA that lead from the initial state to one of the accept states.

The first work that optimizes XPath expressions in the presence of DTDs is presented by Kwong and Gertz [99]. XPath expressions are optimized at query compile time with the knowledge of DTDs before they are passed to a query evaluation engine. A Class Implication and Contradiction graph (CIC-graph), which captures the possible paths' relationships represented by compact path equivalent classes, is constructed from the given DTD. At query compile time, the CIC-graph is used to efficiently determine redundant path conditions, simplification of XPath expressions, and the satisfiability of XPath expressions. Thus an initial XPath expression is rewritten to an optimized one by the elimination of redundant path and unsatisfiable paths and by simplification of the path.

Wang et al. [198] propose two XPath optimization techniques, path shortening and path complementing, to speedup the XPath query processing. These path shortening optimization of XPath expressions is very similar to the techniques of Kwong and Gertz [99]. The path shortening technique reduces the query cost by substituting a long XPath expression with a shorter one. The path complementing chooses the least cost but semantic equivalent XPath expression.

Lee et al. [103] present a very similar idea to the technique of Fernández and Suciu [63] for pruning XPath filters with a DTD. A product automaton is constructed from a DTD automaton constructed from the DTD and an XPath automaton constructed from the XPath expression to eliminate the wildcards ($\star$)and descendent operators ($//$). In addition to the pruning technique using graph schemas, they also introduce an XPath algorithm that optimizes XPath expressions by elimination of void XPath queries and unnecessary axes. This is accomplished by exploring DTD constraints: Child constraint, Descendent constraint, Sibling constraint, and exclusive constraint. In this approach, the DTD automaton is constructed once from a DTD schema; however, the XPath automaton is constructed every time from incoming XPath queries. This approach only applies for a subset of DTDs and XPath expressions because the construction of DTD automata

and the elimination of all descendent operators only apply to nonrecursive DTDs. Furthermore, the elimination of wildcards ($\star$) may lead to the exponential size of the pruned expression.

Silvasti et al. [166] present a query pruning algorithm that specializes in optimization of XML filtering automata by exploring the knowledge of an XML schema. The algorithm utilizes the DTD in the preprocessing phase of the filtering automaton to prune out wildcards ($\star$) and descendent opreators ($//$)from the linear XPath filters. Specifically, all occurrences of wildcards ($\star$) and as many as descendent opreators ($//$)are replaced with all combinations of substituting elements by exploring the DTD. The basis of the algorithm is a pattern-matching automaton with backtracking. The query pruning optimization is first applied to eliminate all wildcards ($\star$) and as many as descendent opreators ($//$)as possible from the linear XPath filters. The pattern-matching automaton is then constructed for the set of keywords formed from the XML element strings separated by descendent opreators ($//$)operators that may possibly remain in the pruned query. This pruning optimization can be used as a preprocessing task of any filtering algorithms to eliminate wildcards ($\star$) and descendent opreators ($//$)from the original subscriber-provided XPath filters when the XML documents are known to conform to a DTD. This approach also allows for different heuristics to be used to regulate the elimination of of wildcards ($\star$) and descendent opreators ($//$)when exhaustive elimination is impossible (e.g. when a nonrecursive DTD is given) or elimination of wildcards ($\star$) would result in an exponential increase of XPath expression size.

Paparizos et al. [146] present a pattern tree based optimization technique for XML query processing by exploiting XML schema information. An XML schema is represented as a Schema Information Graph (SIG) that stores schema information that is useful for query optimizations. They observed that not all schema information is useful for query optimization. SIG captures precisely the schema knowledge that is only useful for the query optimizations, but in a compact form. At compile time, alternate paths are generated from SIG, and each alternate paths is associated with a proper cost that represents the penalty that the query evaluation engine will pay for performing the query evaluation. At runtime, the query evaluation engine will replace an XPath expression with a more cost-efficient alternative from the Alternate path list generated from the SIG. Although the optimization techniques is based on a framework of a pattern tree algebra [146], they can be widely adopted in many XML query evaluation engines.

Although all aforementioned optimization techniques focus on improving XPath evaluation performance, XML schemas can also enables opportunities for XQuery optimization. In [30, 81], Groppe and Bottcher present a schema-based query optimization technique for XQuery queries. An XML schema is first transformed to an ordered schema graph. Each node of an ordered schema graph represents an element node of the schema, the document node or a dummy node that represents a whole choice expression or a whole sequence expression. The ordered schema graph is then extended to an XQuery graph that represents the XQuery expression. The XQuery graph is used to determine which sub-expressions of the XQuery expressions are not used for the retrieval of the final results of the given XQuery expression.

In short, XML schemas such as DTD and XML Schema impose structural information for their instances–XML documents. These constraints on XML documents provides information that can be used to speedup XML query evaluation. XML schema-based XML query optimizations can be categorized into path expression optimizations, i.e., rewriting the original query expressions into lower-cost ones at compile time (or preprocessing stage). One of the typical used techniques is the elimination of wildcards ($\star$) , descendent opreators ($//$)and redundant sub-expressions. Another

one is the substitution of an original path expression with a more cost-efficient one. Although these techniques presented here are based on some specific processing models, the schema-based optimization techniques can be widely adopted by many other XML query processing models.

In addition to the aforementioned techniques used for XPath or XQuery optimizations, there are some other interest in optimizing XPath expressions making use of any available schema knowledge in [129, 135, 204]. These works focus on testing for containment of XPath expressions using various constructors and axes. There is also a work that optimizes semi-structured data with the knowledge of DTD constraints [145]. Also Bohm et al. [28] present an algebra based approach to derive constraints from DTDs in order to optimize expressions.

3.4.7 Logic-based Query Optimizations

XPath query evaluation can be optimized by the use of logic reasoning. Genevés and Vion-Dury [70, 71] present a framework for XPath optimization using logic-based optimization rules. They present a set of containment-based rules that are represented in logic forms. By using logic reasoning, these rules can be used to transform a query expression into a more efficient form by eliminating implicit redundancies and void paths (i.e., contradictions in XPath queries).

Unlike the technique in [71] that optimizes the XPath expression using logic reasoning, Baumgartner et al. [21] present an approach that optimizes the XPath evaluation by guiding the query evaluation engine to not traverse unnecessary parts of the XML document that yield empty result via description logics. In this approach, the DTDs are transformed into a knowledge base in description logic. The description logic representation of DTDs enables reasoning capabilities to determine whether or not a given XPath expression can be satisfied by a document, and to guide the XML query engine to traverse only possibly successful branches of the document. Therefore, an XPath query evaluation is optimized by avoiding unnecessary traversal of nodes of the document that will not yield results.

In short, the basic idea of the logic-based query optimization techniques is to speedup the query processing using logic reasoning. One of the major advantages of this approach is that the logic-based optimizations can be applied at syntactic level, and thus can be combined with other optimizations on an XPath engine.

3.4.8 Incremental Query Evaluation

Incremental query evaluation from cached previous query results generally outperforms full query evaluation from the source updates. Takekawa and Ishikawa [176] propose an efficient approach for incremental XPath query evaluation of streaming XML data processing based on merging automata. In this approach, an XPath expression is first transformed into a query tree, and then the XPath expression is transformed into a sub-XPush machine by a compiler. The compiler is basically the same as the one employed in Gupta and Sucius XPush machine [87], and is based on AFAs. In this method, construction of the state transition tables is postponed until the XML data is evaluated in order to prevent exponential blowup of the number of states. The sub-XPush is then fed into an integrated XPush machine in which an integrated state table and integrated state transition tables can be both incrementally updated.

In [131], Matsumura and Tajima present an incremental approach to perform evaluation of a monotone XPath expression. This approach is based on the partial matching using a bottom-up strategy [127]. A DOM tree of participating elements to each of which is associated with a counter

representing the number of matchings of each query is constructed to reduce the re-computation cost. This approach is useless in practice because it only applies to the insertion and deletion operations at leaf nodes and that only these two operations can occur for the query results. In [26], Björklund et al. present an algorithm for incrementally maintaining an XPath query on an XML database under updates. Unlike technique presented in [131], this technique applies to a wide range of operations, including replacing the current label of specified node by another node, inserting a new leaf node as the next sibling or as the first child of a specified node and deleting a specified node etc. The incremental algorithm stores auxiliary data in compact form to reduce the memory space complexity. The auxiliary data structures for maintenance are linear in the size of the database and polynomial in the size of the query.

In addition to the incremental query evaluation techniques presented above, there is another technique used for incremental query evaluation. The incremental query evaluation typically requires updating the cache appropriately to reflect dynamic source updates. Efficiently maintaining of path expression views provides opportunities for efficiently answering general XML queries. For example, in [16] the authors propose an approach to efficiently answer XML queries using materialized path expression views. The problem of efficient incremental view maintenance has been addressed extensively in the context of relational data models (e.g., [79, 86]), but only few works have addressed it in the context of semistructured data models [1, 112, 153, 212]. There are few works that present techniques for efficiently maintaining path views using incremental methods. In [158], Sawires et al. present an efficient approach to incrementally maintain the views represented by XPath expressions when an update of the source data is present. One of the challenges of the view maintenance is the size of the auxiliary data. In this approach, the size of the auxiliary data is bounded as the product of the size of cached results and the size of the view specification. Unlike the techniques in [158], i.e., only leaf insertion and deletion operations of the XML documents are supported for the incremental view maintenance, Onizuka et al. present an algorithm for incrementally maintaining XPath/XSLT views defined by XPath expressions that support insertion and deletion operations of entire subtrees. In this approach, a dynamic executing flow of an XSLT program is first stored as an XML transformation tree. When an update of the source data is present, the impacted portion of the transformation tree is identified and maintained accordingly by partially re-evaluating the XSLT program. The major disadvantage of this approach is that its worst case performance may be as time consuming as re-evaluation.

Although incremental XPath view maintenance techniques requiring tight coupled are not applied directly to loosely coupled system, these techniques can be extended to loosely coupled systems. Incrementally maintaining consistency of the views requires frequent view recomputation when applying view maintenance techniques for tight coupling systems. In [159], the authors present an approach that reduces the view re-computations to efficiently incremental XPath view maintenance by reducing the irrelevance to checking the intersection and containment of XPath expressions.

3.4.9 XML Query Techniques in Native XML Databases

There has been much work on processing XPath (as a fragment on XQuery) and tree-pattern queries on XML documents stored in databases, that is, in secondary storage, both in the context of native XML databases for efficient query processing and updates [64, 83, 84, 126, 140, 164, 177, 200]. Clearly, once the data is to be stored in a database in a way other than a single monolithic document to allow the addressing and indexing of data, the smaller data chunks (usually document tree nodes) require identifiers of some form. Much work has been done for storing XML data relationally

(e.g., [164, 177]), but numbering schemes for XML nodes that assign unique identifiers to tree nodes that implicitly contain navigation information are also relevant in native XML database systems. It is implicit in [177] that, when designing a node numbering scheme for XML data, a trade-off is necessary between the scheme's support for efficient navigation (tree-pattern queries) and the efficiency for updates. Numbering schemes in which the node identifiers contain much position information allow for more efficient query processing than do schemes which assign only local information that is relative to parent and ancestor nodes; however, updates to the data are more likely to require a relabeling of many nodes with numbers.

Currently two numbering schemes have become prominent in most major research and commercial implementations. The first is Dewey numbering schemes [126, 177] in which a node that is the *j*th child of a node with identifier *i* is assigned the identifier *i.j*; thus the Dewey numbering scheme is the familiar scheme used to label hierarchies of sections and subsections in most books. Given a Dewey numbering scheme, the ancestors of a given node are completely determined, and checking if another node satisfies one of the axes is easily decided. The second one [64, 83, 84] is a form of *global* numbering scheme (refer to Tatarinov [177]. It assigns a preorder and postorder traversal index. In addition, the preorder-index of the parent is stored with each node. Here all axes can be computed using simple θ-joints. This scheme does not require nodes to be labeled consecutively. Reasonable update performance can be achieved by not requiring preorder- and postorder-index to be consecutive and initially leaving some indexes unused. Nodes can be inserted by choosing a suitable pre- and post-order index from the unsigned indexes. A slight modification of this idea uses floating point numbers for the indexes; insertion is done by assigning pre- and post-numbers halfway between those of nodes and between which the new node is to be placed.

One common used technique for XML query evaluation in the native XML databases is assigning indices to the stored data. There are few works have addressed XML query evaluation in context of native XML databases via assigning indices [45, 80, 82, 92]. In [82], the XPath accelerator uses a pre-and post-order encoding system to store XML data elements in a relational database system along with node value and level information. The advantage of this encoding scheme is that it permits traversals from any arbitrary node in the XML document. In [124], the authors use an index structure based on a Metamodel for XML database system combined with relational database technologies to facilitate fast access to XML document elements. The query process involves transforming XPath expressions to SQL which can be executed over an optimized query engine. It employs a preorder encoding scheme based on the technique in [82] and extends it by adding FullPath information. A FullPath is a Metadata construct stored in the index that provides knowledge on data instances in the data tree. The Full-Path-Table provides additional information such as the number of instances of each element in the document tree.

Another commonly used technique is the use of rewriting rules [20, 139]. By the use of rewriting rules, an path expression with reverse axes matching can be transformed into an equivalent reverse-axis-free one. Instead of looking back from the context node, by looking from the beginning of the document to match the element in predicates with reverse axes improves efficiency of evaluation in both time and memory space.

3.4.10 Other Query Optimization Approaches

In [77], the authors present an indexed based approach XML stream query optimization that assumes indices are interleaved within an XML stream [77]. The stream indices are assigned to the positions of the beginning and end of each element. If an element is found to be irrelevant, the

processor can skip to its end without parsing anything in the middle, thus improving the overall performance. However, combining efficiently such indices at runtime with the presence of schema constraints at compile time is still open and an interesting direction to explore in the future.

In [72], a dynamic programming algorithm for full XPath is presented in a straightforward way that XPath can be evaluated in polynomial time. This algorithm is the first of its kind to the best of our knowledge. However, this dynamic programming algorithm computes many useless intermediate results and consumes much memory. To fix this, a more efficient top-down algorithm is also given in [72] as well. This algorithm still runs in polynomial time, with better worst case upper-bounds on running time and memory consumption. Further work on polynomial-time algorithms for XPath, which elaborates on the results of [72] and integrates them into a native XML database management system, can be found in Brantner [33]. This work also shows how to integrate XQuery and XPath processing using a single native algebra.

Path expressions can also be optimized using algebra [32, 147]. In [147], a notion of logical classes of pattern tree nodes is introduced. By using the Tree Logical Class (TLC) algebra, this approach is capable of handling the heterogeneity arising in XML query processing, while avoiding redundant work. An optimization technique for unnested complex XPath queries is proposed in [32] by transforming nested expressions into unnested but equivalences via algebra.

The authors in [12] present a pattern tree minimization technique for minimizing tree pattern using schema constraints information to efficiently perform query evaluation.

3.4.11 Summary

In this section, we presented a set of optimization techniques for efficient XML query evaluation, in particular on XPath navigation. Automata are mainly used and successfully employed in the context of streaming XML. The inherent root-based architecture of XML documents and sequential partitioning still impose big challenges for parallelization XML query evaluation. A schema-based approach can be combined with other optimization techniques with the presence of a schema, and a logic-based approach can be also combined with other query evaluation techniques for extra performance gains. Incremental evaluation offers query optimization for the frequently repeated queries in the context of source date updates, for example, Web services and native XML database systems.

Although all presented techniques here can improve query evaluation performance, most of them are limited to process a subset of XPath expressions or XQuery queries. Full XPath support is still interesting and challenging future work. Furthermore, all these implementations approaches for querying on XML streams sit on top of SAX based parsers. The separation of the query evaluation and parsing introduces an extra layer of processing and the underlying SAX parser may also degrade the performance. Furthermore, to maximum the query evaluations, validation is turned off for these implementations, that is, all these approaches implement a nonvalidating XML query processor.

3.5 Binary XML Encoding Attempts

Two often-cited shortcomings of XML-based applications are XML's processing inefficiency and lack of compactness. Various techniques addressing the processing inefficiency have been introduced though, these aforementioned XML techniques are used to optimize the XML processing performance solely, thus not applicable for reducing the size of XML documents. As one of the

design goals of XML states that "*Terseness in XML markup is of minimal importance*" [205], the verbosity nature of XML results in the huge size of XML documents. The huge size leads to storing, transmitting, processing, and querying inefficiency. As XML usage keeps continuous to grow and large repositories of XML documents are pervasive, a great demand for efficient XML compression tools are demanded.

Since XML data are stored as plain text files, the conventional general purpose compressors can be directly used for compressing XML documents. This group of XML compressors (e.g., GZip [116], BZip [162]), treat XML documents as usual plain text documents and thus apply the traditional compression techniques [157] without utilizing the XML structure. A large number of XML-specific compression techniques have been recently proposed in the literature. By exploiting and utilizing the structure information of XML documents during the encoding and decoding processes, XML-specific compression techniques, in general, achieve better compression ratio than those traditional compression techniques used in general purpose compressors, and may offer the ability of supporting query evaluation, i.e., XML queries can be evaluated over the compressed data. The XML-specific compression techniques can be largely classified into two categories.

XML compression has been studied from a variety of perspectives. For example, some XML compression techniques aim to achieve the highest compression ratio to get the minimum size of compressed archives [7, 42, 113]; some focus on offering the ability to perform query evaluation directly over the compressed representation [130, 168, 180]. We classify the XML-specific compressors into three groups in this paper based on the availability of XML schema information, ability of query evaluation support, and manner of online or offline.

Based on the ability of supporting queries, XML compressors are divided into *Queriable* and *Non-queriable* XML compressors.

- **Queriable XML compressors:** This group of compressors allows XML queries to be evaluated directly over their compressed data [14, 44, 105, 106, 114, 130, 136, 168, 180, 199, 202]. Compressors in this group trade some compression ratio with the ability of query evaluation over the compression format. These compressors focus on avoiding full document decompression during query execution. The ability to perform query evaluation directly on compressed data is important for many applications that are hosted on resource-limited computing devices such as mobile devices.

- **Non-queriable XML compressors:** This group of compressors does not allow XML queries to be processed over their compressed data [43, 73, 101, 109, 105, 168, 170]. Unlike the queriable compressors that balance the compression ratio and query support on compressed data, the major goal of these compressors is to achieve the highest possible compression ratio. General purpose compressors fall into this non-queriable group of compressors.

Based on whether or not XML compressors take advantage of XML schema information during the encoding and decoding processes, XML compressors are classified into *Schema-specific* and *Non-schema-specific* XML compressors.

- **Schema-specific XML compressors:** This group of compressors takes advantages of XML schema information during the encoding and decoding processes [73, 101, 170]. XML schema specifies the structure and element types that may appear in XML documents. By exploiting and utilizing such information, schema-specific XML compressors may in general improve the compression ratio. This group of compressors require access to the same schema information

during encoding and decoding processes on both encoding and decoding. This may hinder its wide acceptance in practice because there is no guarantee that the schema information is always available. But there are still some scenarios that benefit from these schema-specific compression techniques, for example, Web services, where a WSDL description that contains schema information is publicly exposed and available.

- **Non-schema-specific XML compressors:** This group of compressors does not require schema information to perform encoding and decoding processes [42, 105, 113, 167].

Based on the feature that the original structure of the XML document is reserved or not, XML compressors are classified into *Homomorphic* and *Non-homomorphic* XML compressors.

- **Homomorphic compressors:** In this group of XML compressors [130, 168, 180], the original structure of the XML document is retained and the compressed format can be accessed and parsed in the same way as the original document. Homomorphic compressors generally achieve slightly lower compression ratio than non-homomorphic ones.

- **Non-homomorphic compressors:** In this group of XML compressors [14, 44, 105, 114, 136, 199], the structure of the compressed format is different from but semantically equivalent to that of the original XML document.

XML compression can significantly reduce the size of XML documents, and thus may solve the storage, bandwidth and transmission issues. Unfortunately, the problem with compression is that it does not reduce XML processing cost, and it actually introduces additional compression and decompression layers at the sending and receiving ends, respectively.

To address both the processing inefficiency and verbosity shortcomings as well as the limitations of XML compression, alternative forms of XML (often refer as binary XML)—designed to significantly alter the processing, bandwidth, and storage penalties that currently plague XML—have been proposed to substitute for the XML format. Alternative XML formats take advantages of XML language grammar to simultaneously compress, validate, and optimize the processing of XML documents. Many organizations, corporations, and individual researchers proposed their own binary encodings, such as Fast Infoset [171], WAP Binary XML (WBXML) [192], Xebu [95], Efficiency Structured XML (esXML) [203], Efficient XML [89], Binary eXtensible Markup Language (BXML) [61], Binary XML for Scientific Applications (BXSA) [50], XML-binary Optimized Packaging (XOP) [85] to name a few.

To investigate the costs and benefits of an alternative form of XML, and formulate a way to objectively evaluate the potential of a substitute format for XML, the XML Binary Characterization Working Group (XBC WG) was chartered by the World Wide Web Consortium (W3C). The XBC WG conducted experiments and measured performance characteristics of several binary proposals [196]. Based on the measurements, the XBC WG selected Efficient XML ([EffXML]) to be the basis, proposed and recommended Efficient XML Interchange (EXI) [190] as a candidate W3C Recommendation.

EXI is a very compact, high performance XML representation that was designed to work well for a broad range of applications including devices with limited capacity. The EXI format aims to simultaneously improve performance and significantly reduce bandwidth requirements without compromising efficient use of other resources such as battery life, code size, processing power, and memory.

The EXI format is derived from Efficient XML format [89] developed by AgileDelta Inc. EXI does not rely on accurate, complete or current schemas to work. However, EXI can utilize available schema information to improve compactness and performance. It supports arbitrary schema extensions and deviations and also works very effectively with partial schemas or in the absence of any schema.

By using a straightforward encoding algorithm and a small set of data types, EXI produces efficient encodings of the XML event stream. A set of grammars are used to determine which events are most likely to occur at any given point in an EXI stream. It encodes the most likely alternatives in fewer bits.

EXI defines a minimal set of datatype representations called *Built-in EXI datatype* representations that define how values are represented in EXI streams. Values typed as binary are represented as a length-prefixed sequence of octets representing the binary content. Values typed as float are represented as two consecutive integers. The first integer represents the mantissa of the floating point number and the second Integer represents the base-10 exponent of the floating point number.

EXI uses a string table to assign compact identifiers to some certain string values: *uris*, *prefixes* and *uri and local-name in QNames*. Occurrences of string values found in the string table are represented using the associated compact identifier rather than encoding the entire string literal. The string table is initially pre-populated with string values that are likely to occur in certain contexts and is dynamically expanded to include additional string values encountered in the document.

In general, an alternative XML refers to any specification which defines the compact representation of XML in a binary format, i.e., aims to simultaneously reduce the processing cost and verbosity. An alternative XML addresses processing performance inefficiency and verbosity problems by encoding the data format into binary data types used in the programming languages and by using compression techniques to reduce the number of bits processed. It is desired that an alternative XML supports random access and streaming processing mode. Random access enables direct access to particular sections of interest without sequential processing of the document. Streaming processing eliminates the necessity of reading the entire XML document before it can be processed, supporting applications such as partial rendering and interleaving.

Although many of the approaches described in this paper could be used to parse, validate and query an alternative (or binary) XML stream, these efforts inherently lose some of the benefits and flexibility of XML format, provide limited speedups, and can be detrimental to interoperability. With speedups as large as we have seen on standard XML streams, it is not clear that the potential performance improvement of binary XML forms justifies the potential cost in interoperability and standardization, the core strength of XML. The nature of alternative XML is that it runs against the interoperability and human readability – the key advantages of XML, may hinder the wide acceptance of alternative XML. Bayardo et al. provide a thorough analysis of alternative solutions in their paper [22].

CHAPTER 4

MAPPING XML SCHEMA CONTENT MODELS TO AUGMENTED CONTEXT FREE GRAMMARS

This chapter introduces a set of mapping rules that define the translations from XML schema components to extended context free grammars. Structural constraints imposed by the schema are preserved by the grammar rules. Element and attribute value constraints are accomplished by associated semantic actions that are generated from the schema. I begin with introduction of *permutation phrase* and *multi-occurrence phrase* grammars used to efficiently translate some of the XML schema components for free-ordered components such as `xsd:all` and `xsd:attribute`, and occurrences constraints imposed by `xsd:minOcurs` and `xsd:maxOccurs`, respectively. Such components are not easily presented in traditional state machines. Mapping rules for translating each of the XML components are then described in details after an XML schema content model is introduced.

4.1 Permutation Phrase Grammars

Many programming languages and other formal notations have syntactic constructs in which the order of constituents is of no importance. Examples include attribute specifiers in C declarations, class declarations in C++ and in Java to name a few, where members are referenced by their names rather than their positions. XML schema components `xsd:all` and `xsd:attribute` also exhibit the unordered property of constituents. Context-free grammars (and languages) are widely used due to their generation power, simple way of derivation, and efficiency of parsing. Unfortunately, context-free grammars are not well suited to the description of such free-order constructs. Context-sensitive grammars have more powerful expressiveness than context-free grammars. However, there is a big gap between the efficiency of context-free and context-sensitive grammars. Therefore, several branches of extensions of context-fre grammars were introduced for generating proper subfamilies of context-sensitive languages. Some examples in this fields are the matrix grammars [2], indexed grammars [9], macro grammars [65] and programmed grammars [156]. However, rules in ordered grammars are partially ordered in these approaches and derivations are controlled together with some other mechanisms. Nagy [134] introduces a permutation grammars in generating formal linguistics to extend context-free grammars by introducing $AB \rightarrow BA$ interchange (permutation) rules. Although this approach enables efficient expressiveness of permutation languages, it falls into the category of

context-sensitive grammars. Therefore permutation grammars require complexity of greatly parsing, which prevents the advantages of context-free grammars from being utilized.

Therefore, to develop a parser for a context-free grammar with permutation rules, one might first consider parsing such rules by expending these rules. However, each permutation rule consisting of n constituents in the grammar yields $n!$ alternative productions from the same nonterminal. Parsers generated from such expanded grammars quickly grow to an unwieldy size. Furthermore, such grammars will in general break the LL property. For any permutation rule of $n \geq 3$ constituents, there will be at least two different permutations that share the same constituents. Since a constituent may include nonterminals and may derive strings of unbounded length, the expanded grammar may not be LL(k) for any k. Consider for example, a production consisting of three nonterminal constituents is equivalent to the expanded form in context-free grammars with six productions in which three pairs begin with the same constituents A, B and C respectively (see expanded form). In general, a permutation phrase composed of n constituents yields $n!$ alternative phrases in a context free grammars. For example, an element with five attributes, not uncommon in the Web services, would generate 120 different productions in the context free grammar. Furthermore, such grammar violates the LL(1) property. Applying `left-factoring` to these productions to eliminate such ambiguity may result in even more productions.

To address problems of permutation expanding, a term called *permutation phrase* is introduced by Cameron [36] to extend context-free grammars. A *permutation phrase* is a grammatical phrase that specifies a syntactic construct as any permutation of a set of constituent elements. Such a phrase is denoted

$$\langle\langle e_1 \parallel e_2 \parallel \cdots \parallel e_n \rangle\rangle$$

where each $e_i, 1 \leq i \leq n$, is an arbitrary grammatical phrase. It is equivalent, by definition, to an alternation of $n!$ phrases that arise as all possible orderings of its elements. For example, the grammatical production

$$X \rightarrow \langle\langle A \parallel B \parallel C \rangle\rangle$$

is equivalent to the expanded form

$$\begin{aligned} X \rightarrow & A\,B\,C \\ | & A\,C\,B \\ | & B\,A\,C \\ | & B\,C\,A \\ | & C\,A\,B \\ | & C\,B\,A \end{aligned} \qquad \text{(expanded form)}$$

A grammar $\mathcal{G}$ containing permutation phases is the equivalent context-free grammar generated by expanding all the permutation phrases as above. Although permutation phrases do not extend the class of languages described by context-free grammars, permutation phrases allow for the concise and natural expression of free-order constructs in many cases. For example, permutation phrases provide a suitable mechanism for the expressions of free-ordered XML schema components (`xsd:all` and `xsd:attribute`) in an efficient and concise manner.

4.2 Multi-Occurrence Phrases

Many programming languages and other formal notations like in natural languages have syntactic constructs in which the number of constituents must be in a range. For example, the XML components often are constrained with finite arbitrary occurrences. As we have discussed, the traditional automata approach cannot address such constraints since the number of states typically increases to an unmanageable number. To reduce the number of automata states, Reuter [155] extends deterministic finite automata with cardinality constraints on state transitions that map naturally to the encoding of occurrence constraints. Unfortunately, the deterministic automata with cardinality does not perform well-formedness checking, but runs as a separate layer on top of a separate SAX parser, with the associated performance penalty. Furthermore, context-free grammars cannot handle such constraints without extension. To address these problems, we introduce the *multi-occurrence phrase* concept that is used to extend the context-free grammar. A *multi-occurrence phrase* is denoted by

$$\lfloor X \rceil_{\{m..n\}} \tag{4.1}$$

where $1 \leq n \leq m$, and X is an arbitrary grammatical symbol, terminal or nonterminal. It is equivalent, by definition, to an alternation of $m - n$ phrases that arise as all possible number of its occurrences. For example, the grammatical production

$$Y \rightarrow \lfloor X \rceil_{\{3..8\}}$$

is equivalent to the expanded form

$$\begin{aligned} X \rightarrow & A\ A\ A \\ & |A\ A\ A\ A \\ & |A\ A\ A\ A\ A \\ & |A\ A\ A\ A\ A\ A \\ & |A\ A\ A\ A\ A\ A\ A \\ & |A\ A\ A\ A\ A\ A\ A\ A \end{aligned} \qquad \text{(expanded form)}$$

Multi-occurrence phrases have the same problems as permutation phrases, that is, size and LL property. Applying `left-factoring` to eliminate such ambiguity results in more productions.

A grammar $\mathcal{G}$ containing multi-occurrence phases is the equivalent context-free grammar generated by expanding all the multi-occurrence phrases as above. Although multi-occurrence phrases do not extend the class of languages described by context-free grammars, multi-occurrence phrases allow for the concise and natural expression of arbitrary occurrences constructs in many cases. For example, multi-occurrence phrases provide suitable mechanism for the expressions of free-ordered XML schema components (`xsd:all` and `xsd:attribute`) in an efficient and concise manner.

4.3 XML Schema Content Models

To represent and operate on the XML schema grammar, we first introduce the schema components. The schema components, taken in aggregate, are referred to as the schema. It is assumed that the schema for any given grammar is fully resolved before grammar generation using the mapping

rules begins; that is, there are no missing subcomponents, and no attempt is made to further resolve components. The justification of this assumption is provided by the schema recommendation itself. It also assumed that XML namespace constraints associated with attributes are fully resolved; that is, no attribute tag can contain two attributes that have identical names or have qualified names with the same local part and with prefixes which have been bound to namespace names that are identical. Figure 4.1 shows the relevant schema components.

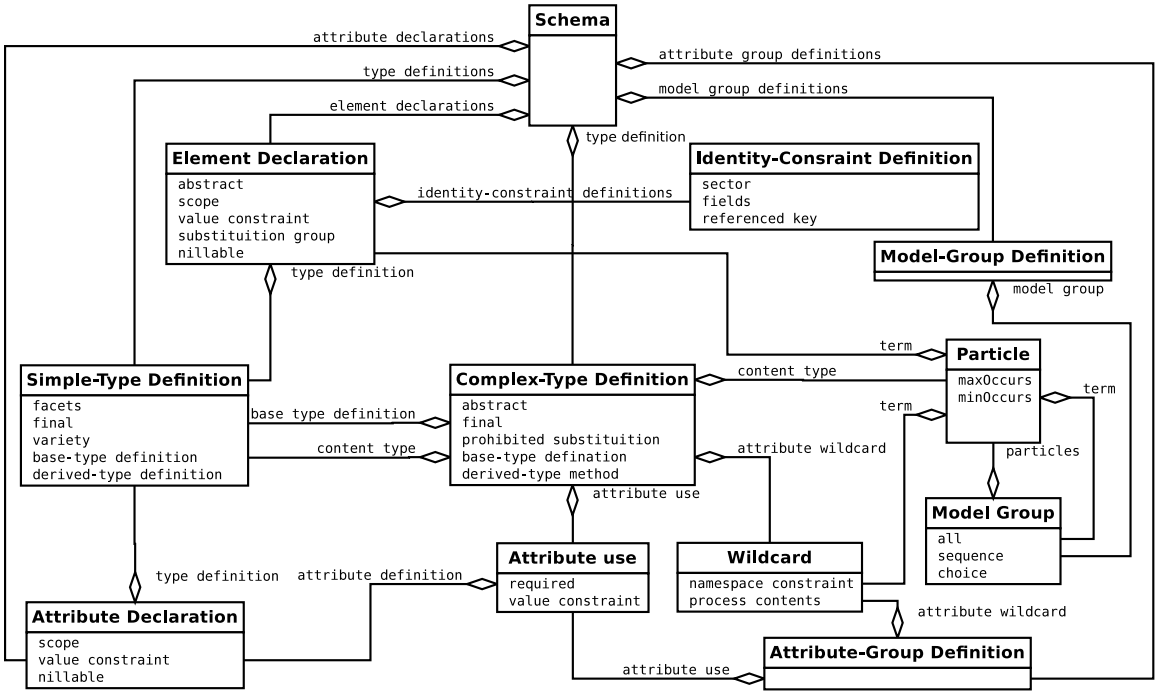


Figure 4.1: XML schema content model

The schema components have six primary component types: element declarations, attribute declarations, complex type definitions, simple type definitions, and model group definitions. The major distinction between definitions and declarations is that definitions create new types that are both simple and complex while declarations specify the elements and attributes with specific names and types, both simple and complex, to appear in XML document instances.

Complex type definitions are used to define new types for the schema used for elements and attributes. Such definitions typically contain a set of element declarations, element references, attribute declarations, and attribute references. Complex type definitions also reference a set of helper components: particle, model group, wildcard, and attribute use.

Complex types may have content that is simple, complex, or empty. When the content is simple, the value of the content-type property is a simple-type definition that defines the content. When the content is empty, the content type is empty. If the complex type has complex content, then the content-type is a particle, which defines a complex content model. The content model for such a complex type is defined in terms of the helper components (particles, model groups, and wildcards).

Complex types have mechanisms including abstract, final, prohibited substitution base type and derived definitions to constrain substitution and inheritance. An XML schema provides a mechanism

to force substitution for a particular element or type. When an element or type is declared to be "abstract", it cannot be used in an instance document. When an element is declared to be abstract, a member of that element's substitution group must appear in the instance document. When an element's corresponding type definition is declared as abstract, all instances of that element must use `xsi:type` to indicate a derived type that is not abstract. XML schema provides a couple of mechanisms that control the derivation of types. One of these mechanisms allows the schema author to specify that for a particular complex type by "final", new types may not be derived from it, either by *restriction*, or *extension*.

Particles and model groups structure the content model for validating element content, which is eventually validated by element declarations or wildcards. The basic unit of the content model is the particle. A particle has a pair of occurrence constraints, min-occurs and max-occurs, and a term. The term of a particle can be an element declaration, a model group, or a wildcard. Model groups, in turn, compose groups of particles, using one of three composition models (`xsd:sequence`, `xsd:choice`, `xsd:all`). These components can be combined freely, within the constraints. In order to facilitate processing, the XML Schema Recommendation places extra restrictions on the use of model groups with the `xsd:all` compositor. XML schema stipulates that an `xsd:all` group must appear as the sole child at the top of content model. In addition, the contents of the `xsd:all` group must be composed of elements and no groups.

Simple type definitions define new types based on built-in types or already defined simple types. Such new types that either restrict or extend built-in's or types defined by simple types are used for elements content values and attributes values.

Element declarations are not themselves types, but rather an association between a name and constraints which govern the appearance of that name in instances. The number of times it occurs is constrained by the component particle. The model group specifies the order of a set of elements names. Constraints are imposed by the attributes `abstract`, `scope`, `nillable`, `value constraint` and `substitution group`. Element contents can be built-in types, simple types, or complex types that may be constrained by particle components. Element declarations may have reference to elements.

Attribute declarations are similar to element declarations. The attribute declarations may be optional by the value of attribute use and may have references to attributes.

4.4 Mapping Schema Components to Augmented Grammar

4.4.1 Terms and Notations

In the mapping rules described below, an X is used to represent any arbitrary schema components that are valid in that component. The symbol N denotes a non-terminal. The mapping operator $\Gamma[\![X]\!]N$ takes schema components X and an optional designated non-terminal N, and returns a set of extended grammar productions and/or the mapping operator with reduced schema component. The mapping operator may also take an optional $\mathbb{M}(mixed)$ used for generating mixed contents of elements. It may also generate semantic actions to validate the elements and attributes values associated with the generated productions. Typically such semantics are associated with character

data PCDATA. In this paper, we use $x.f(text)$ to refer to a semantic action associated with the grammar symbol x.

Tag names, including elements start-tags, elements end-tags, and attributes tags are tokenized in the mapping rules that are typically represented as integers. Tokenization is only performed once and thus subsequent operations on tokens (integers) are much more efficient than on string literals. To this end, other integral types like enumeration types are also represented as tokens to improve performance. Because tags are associated with namespaces, either by namespace prefixes that bind to namespace URIs or unqualified tags bound to default namespace URI, tokens are defined with normalized tags. A normalized tag name is a pair consisting of a namespace URI and a local name. For example, the following XML schema snippet defines two elements purchaseOrder which is bound to default namespace http://www.example.com/PO1 and comment under namespace prefix ns1 which is bound to http://www.example.com/PO2.

```
1  <schema xmlns="http://www.w3.org/2001/XMLSchema"
2    xmlns:ns1="http://www.example.com/PO1" targetNamespace="http://www.
       example.com/PO2">
3
4    <element name="purchaseOrder" type="po:PurchaseOrderType"/>
5    <element name="ns1:comment"       type="string"/>
6
7    <complexType name="PurchaseOrderType">
8      <sequence>
9        <element ref="ns1:comment" minOccurs="0"/>
10     </sequence>
11   </complexType>
12 </schema>
```

The element purchaseOrder is represented as purchaseOrder and the element comment is represented as ns1:comment in an unnormalized form. In a normalized form, however, they are represented as {http://www.example.com/PO1}{purchaseOrder} and {http://www.example.com/PO2}{comment}, respectively. The token of a tag name is defined in a normalized form. Thus two identical tag names under different namespaces are represented as different tokens. For the sake of clarity, the namespace is skipped in this section. Thus the $\mathbb{S}(purchaseOrder)$ is used to present the token of a start-tag of the element {http://www.example.com/PO1}{purchaseOrder}. In general, the $\mathbb{S}(x)$ represents the token of start-tag of the element x in a normalized form with namespace skipped. Similarly, the $\mathbb{E}(x)$ and $\mathbb{T}(x)$ represent the token of end-tag of the element x and the token of the attribute tag x respectively, both in normalized forms with namespaces skipped.

A set of grammar productions are generated from the root element by applying the mapping rules until no more schema components are left in the mapping operator. The symbols N', N'', and N_i represent new and different nonterminals derived from nonterminal N.

4.4.2 Mapping Rules

Mapping Schema Components. Mapping schema components to extended context free grammars starts from components immediately under the xsd:schema element, that is, the global (or top-level) declarations and definitions. The global declarations and definitions can be referenced

from other declarations and definitions. The global declarations and definitions must be unique. No two global declarations or definitions can have the same name. The schema model consists of two declarations and four definitions. One of the declarations is the element declaration and the other is the attribute declaration. Top-level definitions include the model group definition, the attribute group definition, the complex type definition and the simple type definition.

A schema has at least one top-level element declaration that serves as the root node in the instances of the schema. Typically, there are a set of element declarations that can be referenced from other local declarations or definitions. Conceptually, any top-level element can serve as a root element. The mapping operator takes the start symbol S (nonterminal) to generate the root node. Rule 1.a in Table 4.1 shows the mapping rule. The recursive mapping operation generates a set of rules to translate the element. Such mapping rules are introduced later in this section.

Table 4.1: Mapping rules translating top-level schema components.

Mapping Rules		
(1.a)	$\Gamma[\![\text{element } e_1 \text{element } e_2 \ldots \text{element } e_n]\!] S$	$=\Gamma[\![\text{element } e_1]\!] S_1 \cup \Gamma[\![\text{element } e_2]\!] S_2 \cup \ldots \cup \Gamma[\![\text{element } e_n]\!] S_n \cup \{S \rightarrow S_1 \mid S_2 \mid \ldots \mid S_n\}$
(1.b)	$\Gamma[\![\text{attribute } name = \text{“}A\text{”}\, x]\!]$	$=\Gamma[\![\text{attribute } name = \text{“}A\text{”}\, x]\!]\mathbb{N}(A)$
(1.c)	$\Gamma[\![\text{group } name = \text{“}G\text{”}\, x]\!]$	$=\Gamma[\![\text{group } name = \text{“}G\text{”}\, x]\!]\mathbb{N}(G)$
(1.d)	$\Gamma[\![\text{attributeGroup } name = \text{“}G\text{”}\, x]\!]$	$=\Gamma[\![\text{attributeGroup } name = \text{“}G\text{”}\, x]\!]\mathbb{N}(G)$
(1.e)	$\Gamma[\![\text{complexType } name = \text{“}C\text{”}\, x]\!]$	$=\Gamma[\![\text{complexType } name = \text{“}C\text{”}\, x]\!]\mathbb{N}(C)$

The attribute declaration specifies a set of attributes for reference by other components like complex type, attribute group, etc. Because the attributes that an element may have must be defined by a complex type definition, these attribute declarations cannot be used directly by the top-level element declarations. They are typically referenced in complex type definitions. Thus top-level attribute declarations are identified by their names and namespaces (unqualified names are bound to target namespaces). In general, all top-level components must provide names in order to be referenced. Because XML schema requires unique names for the top-level declarations and definitions, these names together with target namespaces provides a mechanism to generate unique nonterminals. As we have described earlier, the operator $\mathbb{N}(N)$ generates a nonterminal for the pair of the name and the namespace with the namespace part ignored for the clarity purpose only. The rules 1.b-1.e show these rules in Table 4.1. The recursive mapping operations are described later in this section.

Mapping Element Declarations. Element declarations specify the elements with specific names and types, simple types or complex types, to appear in the instances of the schema. Element declarations provide for local validation of element information item values using a type definition, specifying default or fixed values for an element information items, establishing uniquenesses and reference constraint relationships among the values of related elements and attributes, and controlling the substitutability of elements through the mechanism of element substitution groups.

The occurrences of the elements in instances of the schema are constrained by specifying the values of the `xsd:minOccurs` and `xsd:maxOccurs`. These attributes can be reduced by applying the mapping rules for the occurrence constraints introduced in Table 4.6. Thus, to simplify the description of the mapping rules for translating element declarations, we assume that the occurrences attributes are resolved by applying the mapping rules for the occurrence constraints first.

Schemas can be constructed by defining sets of named types and then declaring elements that reference the types. However, an element may be declared with a type that can be defined as an

Table 4.2: Mapping rules translating element declarations.

Mapping Rules		
(2.a)	$\Gamma[\![\text{element ref} = \text{“R”}]\!]N$	$=\{N \rightarrow \mathbb{N}(R)\}$
(2.b)	$\Gamma[\![\text{element name} = \text{“E“ type} = \text{“T”}\, x \text{ nillable} = \text{“true”}]\!]N$	$=\Gamma[\![\text{element name} = \text{“E“ type} = \text{“T”}\, x\,]\!]N \cup\{\mathbb{N}(T) \rightarrow \epsilon\}$
(2.c)	$\Gamma[\![\text{element name} = \text{“E“ type} = \text{“T”}\, x \text{ nillable} = \text{“false”}]\!]N$	$=\Gamma[\![\text{element name} = \text{“E“ type} = \text{“T”}\, x\,]\!]N$
(2.d)	$\Gamma[\![\text{element name} = \text{“E“ type} = \text{“T”}\, x \text{ fixed} = \text{“value”}]\!]N$	$=\Gamma[\![\text{element name} = \text{“E“ type} = \text{“T”}\, x\,]\!]N \cup\{\mathbb{N}(T).f(text, value, FIXED\}$
(2.e)	$\Gamma[\![\text{element name} = \text{“E“ type} = \text{“T”}\, x \text{ default} = \text{“value”}]\!]N$	$=\Gamma[\![\text{element name} = \text{“E“ type} = \text{“T”}\, x\,]\!]N \cup\{\mathbb{N}(T).f(text, value, DEFAULT\}$
(2.f)	$\Gamma[\![\text{element name} = \text{“E” type} = \text{“T”}]\!]N$	$=\{N \rightarrow \mathbb{S}(E)\mathbb{N}(T_a)\mathbb{N}(T)\mathbb{E}(E)\}$
(2.g)[a]	$\Gamma[\![\text{element name} = \text{“E“ type} = \text{“T” subsGroup} = \text{“G”}]\!]N$	$=\Gamma[\![\text{element name} = \text{“E“type} = \text{“T”}\,]\!]N \cup\{\mathbb{N}(G) \rightarrow N\}$

[a] In this rule, subsGroup=substitutionGroup for short

anonymous type. Consider for example, the schema fragment (modified from XML schema specification [195]) in Figure 4.2, both the elements `items` and `quantity` have the types defined by anonymous types. By designating a unique name, an anonymous type can be converted semantically to a named type. The converted schema fragment is shown in Figure 4.3.

```
1  <xsd:element name="item">
2    <xsd:complexType>
3      <xsd:sequence>
4        <xsd:element name="productName" type="xsd:string"/>
5          <xsd:element name="quantity">
6            <xsd:simpleType>
7              <xsd:restriction base="xsd:positiveInteger">
8                <xsd:maxExclusive value="100"/>
9                </xsd:restriction>
10             </xsd:simpleType>
11           </xsd:element>
12       </xsd:sequence>
13     </xsd:complexType>
14 </xsd:element>
```

Figure 4.2: Two anonymous type definitions.

Because any anonymous type definition can be semantically converted to its named form, the mapping rules only take named type definitions for simplicity. Although we assume here that anonymous types have been converted to their semantically equivalent forms before the mapping rules are applied, the conversion is not necessary in implementations. The mapping rules that translate the element declarations into context free grammars are shown in Table 4.2.

Each element specifies a name with a type either as a primitive (built-in) type, or a type restricted by a simple type definition or defined by complex type definition. The element name is represented as a start element name and an end element name. In the grammar rules, the start and the end element names are represented as different tokens using $\mathbb{S}(name)$ and $\mathbb{E}(name)$ respectively. As

```
<xsd:simpleType name="quantityType">
  <xsd:restriction base="xsd:positiveInteger">
    <xsd:maxExclusive value="100"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="itemType">
  <xsd:sequence>
    <xsd:element name="productName" type="xsd:string"/>
    <xsd:element name="quantity" type="quantityType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="item" type="itemType"/>
```

Figure 4.3: Named type definitions converted from the anonymous type definitions in Figure 4.2.

discussed earlier, both $\mathbb{S}(name)$ and $\mathbb{E}(name)$ returns the tokens by the normalized names under the namespaces.

Because an element may have optional attributes that are defined in a complex type definition, a nonterminal is used for the optional attributes rules. When the element type is defined by a complex type definition declaring attributes, the nonterminal is used for generating the productions with attributes. When the element type is a primitive type, a type defined by simple type definition, or a type defined by a complex type definition but with no attribute declarations, an empty production is generated for this nonterminal.

The element declaration component has a set of properties specified using attributes. Some of these attributes have no influence on the generated grammar. They are used to validate the local schema. As the mapping rules are applicable to fully resolved (valid) schemas, such attributes are skipped by the mapping rules. Such attributes include `scope`, `abstract`, `substitution group exclusions`.

When the value of `nillable` is `true`, the element can be present without its normal content. Thus an empty production is generated for the type (rule 2.b). When its value is false, there is no influence on the grammar (rule 2.c).

The *value constraint* establishes a default or fixed value for an element. If default is specified, and if the element being validated is empty, then the canonical form of the supplied constraint value becomes the schema normalized value of the validated element in the post-schema-validation infoSet. If fixed is specified, then the element's content must either be empty, in which case fixed behaves as default, or its value must match the supplied constraint value. The value constraint is validated by the generated semantic actions associated with the productions (rule 2.d and rule 2.e).

Substitution groups allow elements to be substituted for other elements, that is, elements can be assigned to a special group of elements that are said to be substitutable for a particular named element called the head element (rule 2.f).

Mapping Complex Type Definitions. In XML schema, complex types are defined using the `xsd:complexType` element and such definitions typically contain a set of element declarations,

element references, and attribute declarations. Table 4.3 shows the mapping rules that translate the complex type definitions.

Table 4.3: Mapping rules translating complex type definitions.

Mapping Rules*		
(3.a)	$\Gamma[\![\text{cT ref} = \text{“R”}]\!]N$	$=\{N \to \mathbb{N}(R)\}$
(3.b)	$\Gamma[\![\text{cT name} = \text{“T”}\ seq\ x\ ]\!]N$	$=\Gamma[\![seq\ x\ ]\!]N \cup \{N \to \mathbb{N}(T)\}$
(3.c)	$\Gamma[\![\text{cT name} = \text{“C”}\ choice\ x\ ]\!]N$	$=\Gamma[\![choice\ x\ ]\!]N \cup \{N \to \mathbb{N}(T)\}$
(3.d)	$\Gamma[\![\text{cT name} = \text{“T”}\ all\ x\ ]\!]N$	$=\Gamma[\![all\ x\ ]\!]N \cup \{N \to \mathbb{N}(T)\}$
(3.e)	$\Gamma[\![\text{cT name} = \text{“T”}\ seq\ x\ attr\ a\ ]\!]N$	$=\Gamma[\![seq\ x\ ]\!]\mathbb{N}(T) \cup \Gamma[\![attr\ a\ ]\!]\mathbb{N}(T_a) \cup \{N \to \mathbb{N}(T)\}$
(3.f)	$\Gamma[\![\text{cT name} = \text{“T”}\ choice\ x\ attr\ a\ ]\!]N$	$=\Gamma[\![choice\ x\ ]\!]\mathbb{N}(T) \cup \Gamma[\![attr\ a\ ]\!]\mathbb{N}(T_a) \cup \{N \to \mathbb{N}(T)\}$
(3.g)	$\Gamma[\![\text{cT name} = \text{“T”}\ all\ x\ attr\ a\ ]\!]N$	$=\Gamma[\![all\ x\ ]\!]\mathbb{N}(T) \cup \Gamma[\![attr\ a\ ]\!]\mathbb{N}(T_a) \cup \{N \to \mathbb{N}(T)\}$
(3.h)	$\Gamma[\![\text{cT name} = \text{“T”}\ seq\ x\ atGp\ a\ ]\!]N$	$=\Gamma[\![seq\ x\ ]\!]\mathbb{N}(T) \cup \Gamma[\![atGp\ a\ ]\!]\mathbb{N}(T_a) \cup \{N \to \mathbb{N}(T)\}$
(3.i)	$\Gamma[\![\text{cT name} = \text{“T”}\ choice\ x\ atGp\ a\ ]\!]N$	$=\Gamma[\![choice\ x\ ]\!]\mathbb{N}(T) \cup \Gamma[\![atGp\ a\ ]\!]\mathbb{N}(T_a) \cup \{N \to \mathbb{N}(T)\}$
(3.j)	$\Gamma[\![\text{cT name} = \text{“T”}\ all\ x\ atGp\ a\ ]\!]N$	$=\Gamma[\![all\ x\ ]\!]\mathbb{N}(T) \cup \Gamma[\![atGp\ a\ ]\!]\mathbb{N}(T_a) \cup \{N \to \mathbb{N}(T)\}$
(3.k)	$\Gamma[\![\text{cT name} = \text{“T”}\ seq\ x\ \text{mixed} = \text{“true”}]\!]N$	$=\Gamma[\![seq\ x\ ]\!]N\mathbb{M}(mixed) \cup \{N \to \mathbb{N}(T)\}$
(3.l)	$\Gamma[\![\text{cT name} = \text{“C”}\ choice\ x\ \text{mixed} = \text{“true”}]\!]N$	$=\Gamma[\![choice\ x\ ]\!]N\mathbb{M}(mixed) \cup \{N \to \mathbb{N}(T)\}$
(3.m)	$\Gamma[\![\text{cT name} = \text{“T”}\ all\ x\ \text{mixed} = \text{“true”}]\!]N$	$=\Gamma[\![all\ x\ ]\!]N\ \mathbb{M}(mixed) \cup \{N \to \mathbb{N}(T)\}$
(3.n)	$\Gamma[\![\text{cT name} = \text{“T”}\ seq\ x\ attr\ a\ \text{mixed} = \text{“true”}]\!]N$	$=\Gamma[\![seq\ x\ ]\!]\mathbb{N}(T)\ \mathbb{M}(mixed) \cup \Gamma[\![attr\ a\ ]\!]\mathbb{N}(T_a) \cup \{N \to \mathbb{N}(T)\}$
(3.o)	$\Gamma[\![\text{cT name} = \text{“T”}\ choice\ x\ attr\ a\ \text{mixed} = \text{“true”}]\!]N$	$=\Gamma[\![choice\ x\ ]\!]\mathbb{N}(T)\ \mathbb{M}(mixed) \cup \Gamma[\![attr\ a\ ]\!]\mathbb{N}(T_a) \cup \{N \to \mathbb{N}(T)\}$
(3.p)	$\Gamma[\![\text{cT name} = \text{“T”}\ all\ x\ attr\ a\ \text{mixed} = \text{“true”}]\!]N$	$=\Gamma[\![all\ x\ ]\!]\mathbb{N}(T)\ \mathbb{M}(mixed) \cup \Gamma[\![attr\ a\ ]\!]\mathbb{N}(T_a) \cup \{N \to \mathbb{N}(T)\}$
(3.q)	$\Gamma[\![\text{cT name} = \text{“T”}\ seq\ x\ atGp\ a\ \text{mixed} = \text{“true”}]\!]N$	$=\Gamma[\![seq\ x\ ]\!]\mathbb{N}(T)\ \mathbb{M}(mixed) \cup \Gamma[\![atGp\ a\ ]\!]\mathbb{N}(T_a) \cup \{N \to \mathbb{N}(T)\}$
(3.r)	$\Gamma[\![\text{cT name} = \text{“T”}\ choice\ x\ atGp\ a\ \text{mixed} = \text{“true”}]\!]N$	$=\Gamma[\![choice\ x\ ]\!]\mathbb{N}(T)\ \mathbb{M}(mixed) \cup \Gamma[\![atGp\ a\ ]\!]\mathbb{N}(T_a) \cup \{N \to \mathbb{N}(T)\}$
(3.s)	$\Gamma[\![\text{cT name} = \text{“T”}\ all\ x\ atGp\ a\ \text{mixed} = \text{“true”}]\!]N$	$=\Gamma[\![all\ x\ ]\!]\mathbb{N}(T)\ \mathbb{M}(mixed) \cup \Gamma[\![atGp\ a\ ]\!]\mathbb{N}(T_a) \cup \{N \to \mathbb{N}(T)\}$

* cT=complexType, seq=sequence, attr=attribute, atGp=attributeGroup

To simplify the mapping rules, the complex type definitions are broken into four categories based on whether or not a complex type definition contains attribute declarations or an attribute group, and whether a complex type definition defines element-only or mixed elements with texts. A mixed content consist of mixed elements and texts, thus can be parsed and validated by DOM trees. However, parsing and validating DOM trees at runtime is inefficient. We believe the mixed content model can also be generated using the context-free grammar productions. For example, the schema fragment in Figure 4.4 defines mixed contents with three elements. The following XML message is a valid instance of this schema fragment, and it can be parsed by the grammar in Figure 4.5.

Table 4.4: Mapping rules translating complex type definitions (continued).

Mapping Rules*	
(3.t)	$\Gamma[\![$cT name = "T" simpleContent extension base = "B"$attr\, a\,]\!]N$ =$\Gamma[\![attr\, a\,]\!]\mathbb{N}(T_a) \cup \{N \rightarrow \mathbb{N}(T)\}$
(3.u)	$\Gamma[\![$cT name = "T" simpleContent restriction base = "B"$attr\, a\,]\!]N$=$\Gamma[\![attr\, a\,]\!]\mathbb{N}(T_a) \cup \{N \rightarrow \mathbb{N}(T)\}$
(3.v)	$\Gamma[\![$cT name = "T" complexContent extension base = "B"$x\,]\!]N$ =$\Gamma[\![x\,]\!]N$
(3.w)	$\Gamma[\![$cT name = "T" complexContent restriction base = "B"$x\,]\!]N$ =$\Gamma[\![x\,]\!]N$

* cT=complexType, seq=sequence, attr=attribute, atGp=attributeGroup

```
<xs:element name="letter">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="orderid" type="xs:positiveInteger"/>
      <xs:element name="shipdate" type="xs:date"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 4.4: An example of schema fragment defining mixed content

```
<letter>
  Dear Mr.<name>John Smith</name>.
  Your order <orderid>1032</orderid>
  will be shipped on <shipdate>2001-07-13</shipdate>.
</letter>
```

$$
\begin{aligned}
S &\rightarrow \texttt{<letter>}L\texttt{</letter>} \\
L &\rightarrow T\,N\,T\,I\,T\,D\,T \\
T &\rightarrow \{\texttt{text}\}|\epsilon \\
N &\rightarrow \texttt{<name>}\{\texttt{text}\}\texttt{</name>} \\
I &\rightarrow \texttt{<orderid>}\{\texttt{text}\}\texttt{</orderid>}\{\texttt{text}\} \\
D &\rightarrow \texttt{<shipdate>}\{\texttt{text}\}\texttt{</shipdate>}
\end{aligned}
$$

Figure 4.5: Grammar for the parsing mixed content model.

Based on this observation, we extend the mapping operator to take an extra argument $\mathbb{M}(mixed)$ for the group model to generate corresponding rules for such a mixed content model. Translating these components are accomplished by applying rules 3.b-3.s.

Mapping references is straightforward (see rule 3.a).

In addition, the content of complex type definitions may contain derived type definitions by extending or restricting already existing types, called base types. In such components, the base

types have no influence on generated grammars. The mapping rules 3.t-3.u are shown in Table 4.4. Although these rules simply ignore the base types that generate productions elsewhere, duplicate productions may generated because the contents of such models contain the duplicate elements or attributes. Such duplicate productions must be eliminated from the grammars to reduce the number of productions and eliminate ambiguity.

Mapping Simple Type Definitions. Simple type definitions define data types for constraining character values of elements and attributes. Table 4.5 shows the mapping rules for the non-normative simple components `enumeration`, `list`, and `union`.

Other simple types including built-in types are typically validated by generated semantic actions ($\mathbb{N}(T).f(text)$) associated with the production $\mathbb{N}(T) \rightarrow \overline{CD}$, where the $\mathbb{N}(T)$ denotes the nonterminal generated from the data type and the special token $\overline{CD}$ denotes the token of the character data (PCATA). The semantic actions are specialized for the built-in types and types restricted by the simple type definitions (by facets such as `pattern`, `length`, `maxLength` etc). The semantic actions validate the values, converts to binary representations, and passes the binary representations to applications. For applications that do not operate on binary representations, generated semantic actions do not need to do the text-to-binary conversion, in favor of performance.

Table 4.5: Mapping rules translating non-normative simple type components.

	Mapping Rules*	
(4.a)	$\Gamma[\![$simpleType name = "T" restriction enum value = "v_1" ... enum value = "v_n"$]\!]$N	$=\{N \rightarrow \mathbb{N}(T)\}$ $\cup\{\mathbb{N}(T) \rightarrow \mathbb{T}(v_1)\| \ldots \|\mathbb{T}(v_n)\}$
(4.b)	$\Gamma[\![$simpleType name = "T" restriction list itemType = "T"$]\!]$N	$=\Gamma[\![T]\!]N'$ $\cup\{N \rightarrow N_1N_2, N_1 \rightarrow N', N_2 \rightarrow N'N_2, N_2 \rightarrow \epsilon\}$
(4.c)	$\Gamma[\![$simpleType name = "T" union memberTypes = "$T_1 \ldots T_n$"$]\!]$N	$=\{N \rightarrow \mathbb{N}(T), \mathbb{N}(T) \rightarrow \mathbb{T}(T_1)\| \ldots \|\mathbb{T}(T_n)\}$

* enum=enumeration

Mapping Occurrence Constraints (Particle Component). The particle component specifies an occurrence range for an element, a model group or a wildcard component to appear by the values of `minOccurs{}` and `maxOccurs{}`. When the value of the `minOccurs{}` attribute in its declaration is 0, the element or the group model is optional. Optional appearance is mapped straightforwardly to an *epsilon* production in context free grammars. In general, an element or a group model constrained by the attribute `minOccurs{}` required to appear is determined by the value of `minOccurs{}` is 1 or more. The value of a `minOccurs{}` attribute may be any non-positive but must be less than or equal to the value of the `maxOccurs{}` attribute. The maximum number of times that an element or a group model may appear is determined by the value of a `maxOccurs{}` attribute in its declaration. This value may be any positive number but must not be less than the value of `minOccurs{}` attribute if the latter is specified. The default value for both of the occurrence attributes is 1. The value of the `maxOccurs{}` attribute may be the term `unbounded` to indicate there is no maximum number of occurrences constraint; that is, it can appear an arbitrary number of times. Such an unbounded maximum occurrence constraint is mapped to a recursive production in context-free grammars. An arbitrary finite range specified by `minOccurs{}` and `maxOccurs{}` is mapped to a *multi-occurrence phrase* production. Table 4.6 shows the rules of the mapping occurrence constraints.

Table 4.6: Mapping rules translating occurrence constraints.

Mapping Rules		
(5.a)	$\Gamma[\![x \text{ minOccurs} = \text{“1”}]\!]N$	$=\Gamma[\![x]\!]N$
(5.b)	$\Gamma[\![x \text{ maxOccurs} = \text{“1”}]\!]N$	$=\Gamma[\![x]\!]N$
(5.c)	$\Gamma[\![x \text{ minOccurs} = \text{“1”} \text{ maxOccurs} = \text{“1”}]\!]N$	$=\Gamma[\![x]\!]N$
(5.d)	$\Gamma[\![x \text{ minOccurs} = \text{“0”}]\!]N$	$=\Gamma[\![x]\!]N \cup \{N \to \epsilon\}$
(5.e)	$\Gamma[\![x \text{ minOccurs} = \text{“0”} \text{ maxOccurs} = \text{“1”}]\!]N$	$=\Gamma[\![x]\!]N \cup \{N \to \epsilon\}$
(5.f)	$\Gamma[\![x \text{ minOccurs} = \text{“0”} \text{ maxOccurs} = \text{“unbounded”}]\!]N$	$=\Gamma[\![x]\!]N' \cup \{N \to N'N, N \to \epsilon\}$
(5.g)[a]	$\Gamma[\![x \text{ minOccurs} = \text{“0”} \text{ maxOccurs} = \text{“n”}]\!]N$	$=\Gamma[\![x]\!]N' \cup \{N \to \epsilon, N \to \lfloor N' \rceil_{\{1..n\}}\}$
(5.h)	$\Gamma[\![x \text{ maxOccurs} = \text{“unbounded”}]\!]N$	$=\Gamma[\![x]\!]N' \cup \{N \to N_1N_2, N_1 \to N'\}$ $\cup \{N_2 \to N'N_2, N_2 \to \epsilon\}$
(5.i)	$\Gamma[\![x \text{ minOccurs} = \text{“1”} \text{ maxOccurs} = \text{“unbounded”}]\!]N$	$=\Gamma[\![x \text{ maxOccurs} = \text{“unbounded”}]\!]N$
(5.j)[a]	$\Gamma[\![x \text{ maxOccurs} = \text{“n”}]\!]N$	$=\Gamma[\![x]\!]N \cup \{N \to \lfloor N \rceil_{\{1..n\}}\}$
(5.k)	$\Gamma[\![x \text{ minOccurs} = \text{“1”} \text{ maxOccurs} = \text{“n”}]\!]N$	$=\Gamma[\![x \text{ maxOccurs} = \text{“unbounded”}]\!]N$
(5.l)[a]	$\Gamma[\![x \text{ minOccurs} = \text{“n”} \text{ maxOccurs} = \text{“unbounded”}]\!]N$	$=\Gamma[\![x]\!]N' \cup \{N \to N_1N_2, N_1 \to \lfloor N' \rceil_{\{n..n\}}\}$ $\cup \{N_2 \to \epsilon, N_2 \to N'N_2\}$
(5.m)[b]	$\Gamma[\![x \text{ minOccurs} = \text{“m”} \text{ maxOccurs} = \text{“n”}]\!]N$	$=\Gamma[\![x]\!]N \cup \{N \to \lfloor N \rceil_{\{m..n\}}\}$

[a] In these rules, n is a positive number, and $n > 1$.
[b] In this rule, both m and n are positive numbers, and $n \geq m > 1$.

Mapping Model Group Definitions. Model group definitions are provided primarily for reference from the XML representation of complex type definitions (`xsd:complexType` and `xsd:group`). Thus, model group definitions provide a replacement for some uses of XML's parameter entity facility. A model group definition associates a name with a model group. By reference to the name, the entire model group can be incorporated by reference into a *term*. Table 4.7 shows such mapping rules translating model group definitions to context-free grammar.

Table 4.7: Mapping rules translating model group definitions.

Mapping Rules		
(6.a)	$\Gamma[\![\text{group name} = \text{“G”}\ x]\!]N$	$=\Gamma[\![x]\!]\mathbb{N}(G) \cup \{N \to \mathbb{N}(G)\}$
(6.b)	$\Gamma[\![\text{group ref} = \text{“G”}]\!]N$	$=\{N \to \mathbb{N}(G)\}$

Mapping Model Group Components. The definitions of complex types provide three different model groups, `xsd:sequence`, `xsd:choice` and `xsd:all`, which specify the order of their appearances in instances. In an instance, elements declared in model group `xsd:sequence` must appear in the exact order they are declared. The `xsd:choice` group element allows only one of its children to appear in an instance. The third option for constraining elements in a group is `xsd:all`. All the elements in the `xsd:all` group may appear in any order. To facilitate processing for this model, W3C puts limitations to this content model, that is, the all group is limited to the top-level of any content model. Moreover, the group's children must all be individual elements (no groups), and no element in the content model may appear more than once, that is, the permissible values of `xsd:minOccurs` and `xsd:maxOccurs` are 0 and 1. For example, to allow the child elements of `purchaseOrder` to appear in any order, a group content model `xsd:all` is used in the following schema fragment:

```
<xsd:complexType name="PurchaseOrderType">
  <xsd:all>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="items"  type="Items"/>
  </xsd:all>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>
```

By this definition, a `comment` element may optionally appear within `purchaseOrder`, and it may appear before or after any `shipTo`, `billTo` and `items` elements, but it can appear only once. Moreover, the stipulations of an `xsd:all` group do not allow declaring an element such as `comment` outside the group as a means of enabling it to appear more than once. XML Schema stipulates that an `xsd:all` group must appear as the sole child at the top of a content model. Such restrictions ensure that constituents in the `xsd:all` content model must not be nested, thus facilitating the validation. Mapping rules for these three model groups are shown in Table 4.8.

In addition, the complex type may define mixed content that the text may appear between elements and their child elements (Rules 6.h - 6.n)

Table 4.8: Mapping rules translating model group components.

Mapping Rules		
(7.a)[a]	$\Gamma[\![\text{sequence}\, x_1]\!]N$	$=\Gamma[\![x_1]\!]N$
(7.b)[b]	$\Gamma[\![\text{sequence}\, x_1 x_2 \dots x_n]\!]N$	$=\Gamma[\![x_1]\!]N_1 \cup \Gamma[\![x_2]\!]N_2 \cup \dots \cup \Gamma[\![x_n]\!]N_n \cup \{N \to N_1 N_2 \dots N_n\}$
(7.c)[a]	$\Gamma[\![\text{choice}\, x_1]\!]N$	$=\Gamma[\![x_1]\!]N$
(7.d)[b]	$\Gamma[\![\text{choice} x_1 x_2 \dots x_n]\!]N$	$=\Gamma[\![x_1]\!]N_1 \cup \Gamma[\![x_2]\!]N_2 \cup \dots \cup \Gamma[\![x_n]\!]N_n \cup \{N \to N_1 \mid N_2 \mid \dots \mid N_n\}$
(7.e)[a]	$\Gamma[\![\text{all}\, x_1]\!]N$	$=\Gamma[\![x_1]\!]N$
(7.f)[c]	$\Gamma[\![\text{all}\, x_1 x_2]\!]N$	$=\Gamma[\![x_1]\!]N_1 \cup \Gamma[\![x_2]\!]N_2 \cup \{N \to N_1 \mid N_2\}$
(7.g)[d]	$\Gamma[\![\text{all}\, x_1 \dots x_n]\!]N$	$=\Gamma[\![x_1]\!]N_1 \cup \Gamma[\![x_2]\!]N_2 \cup \dots \cup \Gamma[\![x_n]\!]N_n$ $\cup \{N \to \langle\!\langle N_1 \parallel N_2 \parallel \dots \parallel N_n \rangle\!\rangle\}$
(7.h)[a]	$\Gamma[\![\text{sequence}\, x_1]\!]N\mathbb{MM}(mixed)$	$=\Gamma[\![x_1]\!]N' \cup \{N \to TN'T, T \to \overline{CD} \mid \epsilon\}$
(7.i)[b]	$\Gamma[\![\text{sequence}\, x_1 x_2 \dots x_n]\!]N\mathbb{MM}(mixed)$	$=\Gamma[\![x_1]\!]N_1 \cup \Gamma[\![x_2]\!]N_2 \cup \dots \cup \Gamma[\![x_n]\!]N_n$ $\cup \{N \to TN_1TN_2 \dots TN_nT, T \to \overline{CD} \mid \epsilon\}$
(7.j)[a]	$\Gamma[\![\text{choice}\, x_1]\!]N\mathbb{MM}(mixed)$	$=\Gamma[\![x_1]\!]N' \cup \{N \to TN'T, T \to \overline{CD} \mid \epsilon\}$
(7.k)[b]	$\Gamma[\![\text{choice} x_1 x_2 \dots x_n]\!]N\mathbb{MM}(mixed)$	$=\Gamma[\![x_1]\!]N_1 \cup \Gamma[\![x_2]\!]N_2 \cup \dots \cup \Gamma[\![x_n]\!]N_n$ $\cup \{N \to TN_1T \mid TN_2T \mid \dots \mid TN_nT, T \to \overline{CD} \mid \epsilon\}$
(7.l)[a]	$\Gamma[\![\text{all}\, x_1]\!]N\mathbb{MM}(mixed)$	$=\Gamma[\![x_1]\!]N' \cup \{N \to TN'T, T \to \overline{CD} \mid \epsilon\}$
(7.m)[c]	$\Gamma[\![\text{all}\, x_1 x_2]\!]N\mathbb{MM}(mixed)$	$=\Gamma[\![x_1]\!]N_1 \cup \Gamma[\![x_2]\!]N_2 \cup \{N \to TN_1T \mid TN_2T, T \to \overline{CD} \mid \epsilon\}$
(7.n)[d]	$\Gamma[\![\text{all}\, x_1 \dots x_n]\!]N\mathbb{MM}(mixed)$	$=\Gamma[\![x_1]\!]N_1 \cup \Gamma[\![x_2]\!]N_2 \cup \dots \cup \Gamma[\![x_n]\!]N_n$ $\cup \{N \to \langle\!\langle N'_1 \parallel N'_2 \parallel \dots \parallel N'_n \rangle\!\rangle\}$ $\cup \{N'_1 \to TN_1T, N'_2 \to TN_2T, \dots, N'_n \to TN_nT, T \to \overline{CD} \mid \epsilon\}$

[a] There is exactly one item in the model group.
[b] There are more than one items in the model group, $n > 1$.
[c] There are exactly two items in the model group.
[d] There are more than two items in the model group, $n > 2$.

Mapping Attribute Uses. An *attribute use* is a utility component which controls the occurrence and defaulting behavior of attribute declarations. It plays the same role for attribute declarations in complex types that particles play for element declarations. When the `use` is specified with "required", the attribute must be present in its element. When the `use` is specified with "optional", the attribute may or may not be present. The value "prohibited" indicates the attribute is only used in a restriction of another complex type.

Table 4.9: Mapping rules translating attribute uses.

Mapping Rules	
(8.a)	$\Gamma[\![attribute\, x \text{ use} = \text{“required”}]\!]N \quad = \Gamma[\![attribute\, x\,]\!]N$
(8.b)	$\Gamma[\![attribute\, x \text{ use} = \text{“optional”}]\!]N \quad = \Gamma[\![attribute\, x\,]\!]N\ \{N \rightarrow \epsilon\}$
(8.c)	$\Gamma[\![attribute\, x \text{ use} = \text{“prohibited”}]\!]N = \phi$

Mapping Attribute Group Definitions. Attribute groups define a set of attribute definitions for reference. They are identified by their names, thus attribute group identities are unique within the XML schema. Mapping rules translating attribute group definitions are shown in Table 4.10.

Table 4.10: Mapping rules translating attribute group definitions.

Mapping Rules	
(9.a)	$\Gamma[\![\text{attributeGroup name} = \text{“G”}\, x]\!]N = \Gamma[\![x\,]\!]\mathbb{N}(G) \cup \{N \rightarrow \mathbb{N}(G)\}$
(9.b)	$\Gamma[\![\text{attributeGroup ref} = \text{“G”}\,]\!]N \quad = \{N \rightarrow \mathbb{N}(G)\}$

Mapping Annotations. Annotations have no influence on grammars and thus are ignored.

Mapping Wildcards. The wildcard schema component describes a flexible mechanism that enables content models to be extended by any elements and attributes belonging to specified namespaces. It imposes namespaces and process content constraints. A wildcard provides for validation of attribute and element information items dependent on their namespace name, but independently of their local name.

The process content model controls the impact on assessment of the information items allowed by wildcards using three values, `strict`, `skip` and `lax`. The value of `strict` requires the item must be valid as appropriate. The value of `skip` indicates no constraints at all, that is the item must simply be well-formed XML without validation. The value of `lax` requires the item must be valid if and only if the item has a uniquely determined declaration available. When a declaration is not determined, validation is not required.

The namespace constraint provides for validation of attribute and element items that:

- `any`: Any namespace or not namespace-qualified is valid;
- `not and a namespace name`: Namespace-qualified with a namespace other than the specified namespace name is considered valid;
- `not and absent`: Namespace-qualified;

- `a set whose members are either namespace names or absent`: Any of the specified namespaces and/or, if absent is included in the set, are unqualified.

Because the schema information may not be available until run-time, parsing and validation for a wildcard can only be accomplished at runtime. Special tokens representing these values are introduced to inform the TDX parser to perform parsing and validation using non-compile validation techniques for wildcards (`xsd:any` and `xsd:anyAttribute`) at runtime.

Generating Semantic Actions. TDX performs data type validation for the element and attribute text values via semantic actions associated with the grammar productions. These semantic functions are generated from the schema definitions and therefore are specialized to these definitions. Specialized functions are typically more efficient than generic APIs. For example, the following schema fragment defines a zip code using a regular expression. Traditional validation for this zip code takes regular expression and checks validation by calling a generic function (or third party lib) that matches the regular expression. This typically is implemented using recursive function to support the wildcard (*) in regular expressions. In TDX, this validation can be much efficiently achieved by a DFA (Figure 4.6).

```
1 <simpleType name='us-zipcode'>
2   <restriction base='string'>
3     <pattern value='[0-9]{5}(-[0-9]{4})?'/>
4   </restriction>
5 </simpleType>
```

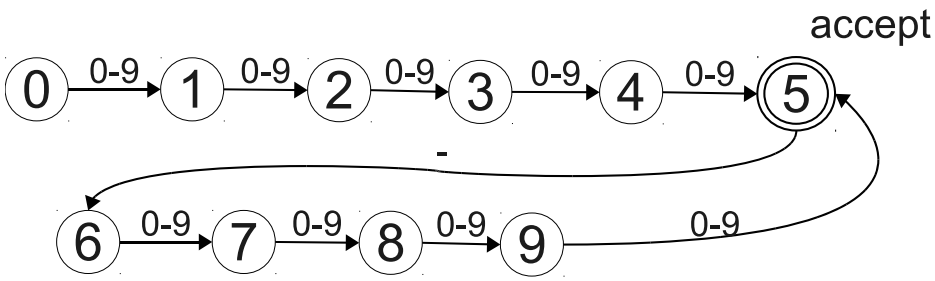


Figure 4.6: DFA recognizing US zip code.

XML applications typically require conversions from text representations to binary in-memory representations. Conversions are usually expensive to compute and may become a performance bottleneck [51]. Application should be not burdened by such conversions. To this end, TDX generates semantic functions that not only performs validation but also convert text representations to in-memory objects for the applications. To maximize the overall performance, TDX supports two strategies for generating semantic validation functions. For *pure* validation parsers or applications that do not require in-memory representations different for text formats, TDX generates functions optimized for validation-only functions such as DFAs. For applications that operate on in-memory objects that are different from text formats, TDX generates functions that convert the text strings to the in-memory representations for the applications and perform validation at one time. Consider for instance, the following schema fragment that defines an integer range [100, 999], for *pure* validation,

this can be accomplished using a DFA for boosting performance. However, when it is used by an application heavily manipulating on binary in-memory objects, TDX converts the text value to a binary type of integer and performs a range check. Only the valid binary representation is fed to the application. TDX supports both build-in types or derived data types by simple type definitions.

```
1 <xsd:simpleType name="myInteger">
2   <xsd:restriction base="xsd:integer">
3     <xsd:minInclusive value="100"/>
4     <xsd:maxInclusive value="999"/>
5   </xsd:restriction>
6 </xsd:simpleType>
```

CHAPTER 5

TABLE-DRIVEN XML PARSING AND VALIDATION

This chapter presents the table-driven XML parsing and validation techniques—called table-driven XML processing (TDX)—for parsing, validation, and searching XML streams. TDX implements a single pass, nonrecursive predictive top-down XML parser without backtracking, resulting in a high-performance XML parsing and validation parser is both time and space optimal. The idea is that by mapping XML schemas to augmented grammars stored in compact tabular forms, an efficient compiler-based approach can be used to quickly parse, validate XML streams. To be able to handle the full expressive power of the W3C XML schema [195], *permutation phrase* and *multi-occurrence* grammar productions are introduced. Novel algorithms are developed for parsing and validating the augmented grammars using nonrecursive predicative parsing techniques. The predictive parsing validation tables are constructed from the augmented grammar at compile-time. The tabular forms can be compiled with the parsing engine or can be populated on-the-fly at runtime. An efficient specialized streaming parsing engine that is independent of the schemas combines the parsing and validation into a single pass by consulting the parsing table at runtime. Because the predictive parsing table pre-encodes the parsing states and schema constraints, TDX implements an efficient top-down validating parser that does not require backtracking or accessing the schema at runtime. Looking up the parsing table is determined by the current input symbol and the token on top of the stack of the parsing engine, thus it takes constant time.

Type-checking validations are accomplished via semantic actions associated with grammar productions. These semantic actions are specialized to the content value constraints defined by the schema components or to the built-in data types. Special tokens in grammar productions are used to perform type-checking validation by triggering validation routines.

5.1 Overview of TDX

Figure 5.1 shows the architecture of the TDX parser. The system consists of a scanner, a parsing engine, a set of tables: token table, parsing table and validation table, and possible back-end application logic. Tables are constructed at compile-time from XML schema(s) while the scanner and parsing engine are independent of an XML schema, thus they are hard-coded efficiently.

The token table stores the token representation of normalized namespace bounded tag names, opening and closing element tag names and attribute tag names as well as some special tokens, which

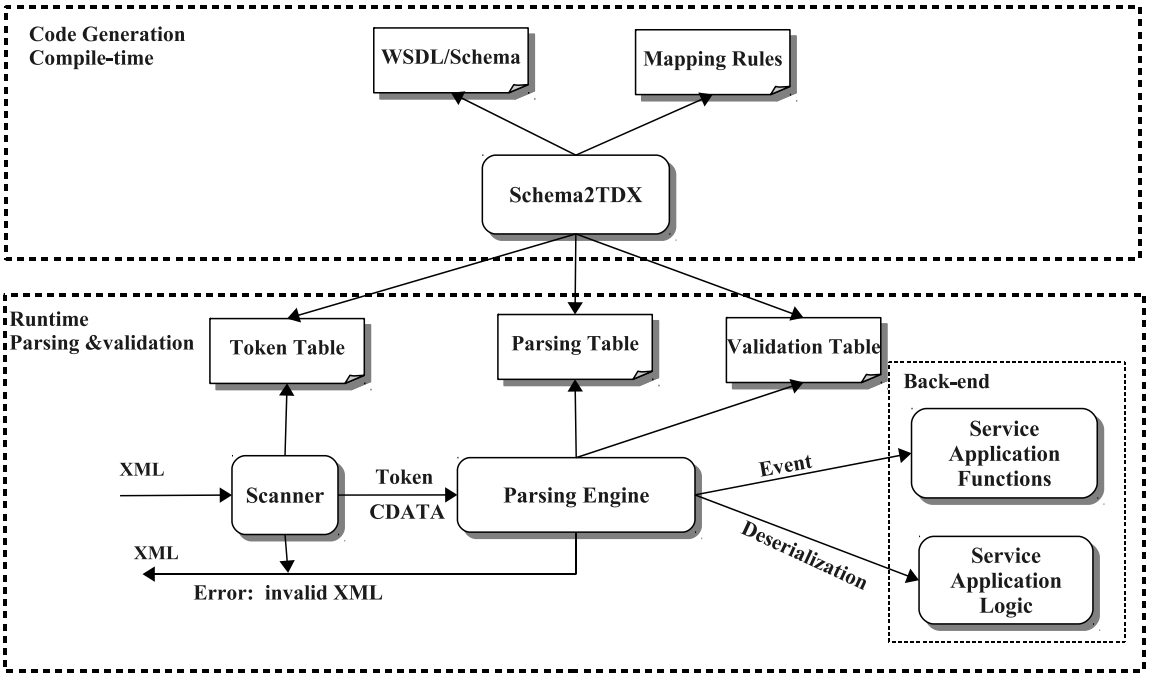


Figure 5.1: System architecture of a table-driven validating parser

are represented as integers. Manipulating of integers by the parsing engine is much more efficient than operating on the normalized tag tokens in string literal. To support integral representations of the tag tokens, TDX also maintains hash tables for the namespaces and tag names. The parsing table encodes the augmented LL(1) grammar productions in correct entries. Error recovery information is also stored in the table. The validation table stores indices or pointers to the validation routines that perform type checking for the element content data values. By using indices, the semantic actions associated with grammar productions do not need to be pre-compiled. This ensures that the modular tables are swappable.

The scanner takes XML message as input, scans and breaks the element names, attribute names and element content values into tokens for the parsing engine by consulting the token table generated from XML schema(s) or WSDL descriptions. An error message is returned when parsing or validation fails.

Upon receiving a token from the scanner, the parsing engine performs well-formedness checking by consulting the parsing table. Because the parsing table encodes the structure constraints imposed by the XML schema(s), structure validation is integrated and accomplished automatically. Element content data values are validated by invoking the corresponding routines by consulting the validation table when special tokens are met. The validation routines also convert text values to internal binary presentations when imposed by the schema. Invalid error information is also returned when validation fails. Well-formed and valid tokens and data are delivered directly or deserialized to the applications in the back-end. In addition, user-defined application semantic routines can be manually inserted into grammar productions and invoked automatically.

TDX works in two different modes: static and dynamic modes. The former mode compiles all

the tables together with the scanner, parsing engine and back-end applications. Thus static TDX implements a generic compile-time parser that maximizes the runtime performance compared to the latter. However, the static mode requires re-compilation when schema update is present.

In dynamic mode, tables can be populated on-the-fly at runtime. Thus recompilation is not required when the schema is updated. This offers an efficient and flexible mechanism to address the schema updates. The cost of this flexibility is the overhead of re-loading the tables. However, this initialization cost is only performed once.

The token table, parsing table and validation table are constructed from the XML schema automatically using a parser generator. The generator takes XML schema(s) as input and generates a set of tables by consulting mapping rules. To support static and dynamic modes of TDX, the generator can be configured to generate two different forms. One is language-dependent source codes for the parser for static mode. Another form is able to be populated on-the-fly and is used for the dynamic mode.

5.2 Constructing Parsing Table

A non-recursive predictive top-down parsing table can be constructed using the algorithm described in the famous "dragon book" [8]. Constructing such a parsing table requires the grammar exhibits the LL(1) property and uses $FIRST$ and $FOLLOW$ sets. It is critical to ensure that the generated grammar exhibit LL(1) properties to construct a parsing table. Moreover, TDX augments the context-free grammar by introducing two new productions: *permutation phrase* and *multi-occurrence phrase* grammar productions. In this section, we first describes the non-LL(1) property handling. Construction of $FIRST$ and $FOLLOW$ sets are then presented.

5.2.1 Preserving the LL(1) Property

It is critical to preserve the LL(1) property for constructing the LL(1) parsing table, otherwise parsing construction fails and thus the parsing engine will not be able to handle the parsing and validation correctly. The mapping rules map element tag names and attribute tag names to unique tokens and makes use of namespace bonded names as nonterminals. Element content value are represented as special but unique token in grammars. As result, most grammar productions generated by those mapping rules including *permutation phrase* and *multi-occurrence phrase* mapping rules exhibit the LL(1) properties.

However, a few special cases though rarely in practice, typically involving in the `xsd:choice` group model, may end up producing a non-LL(1) grammar due to ambiguity in the schema. Consider for example the schema fragment shown in Table 5.1. The `xsd:choice` content has two child elements with the same name `"A"`. This generates a non-LL(1) grammar shown in Table 5.1 (second columns). Note that both of the two productions with the same nonterminal, $C \rightarrow C_1$ and $C \rightarrow C_2$, can be used to generate element `"A"`, thus causing ambiguity. This results in a parsing conflict when constructing the LL(1) parse table, i.e., either one of the productions ends up at the same place in the parse table.

To address this ambiguity, two approaches can be applied. One is to augment the mapping rules and deeply exploit the schema information to generate the unambiguous grammar like the one in the third column in Table 5.1. Another approach is to perform *left factoring* to eliminate conflicts [8]. The former approach requires that a generator performs two passes of the schema to

Schema Fragment	Non-LL(1) Grammar	LL(1) Grammar
<complexType name="C"> <choice> <sequence> <element name="A" type="T"/> <element name="B" type="T"/> </sequence> <element name="A" type="T"/> </choice> </complexType>	$C \rightarrow C_1$ $C \rightarrow C_2$ $C_1 \rightarrow C_3 C_3'$ $C_3' \rightarrow C_4$ $C_3 \rightarrow$ <A> T </A> $C_4 \rightarrow$ <B> T </B> $C_2 \rightarrow$ <A> T </A>	$C \rightarrow C_1 C_1'$ $C_1 \rightarrow$ <A> T </A> $C_1' \rightarrow$ <B> T </B> $C_1' \rightarrow \epsilon$

Table 5.1: Schema fragment generating non-LL(1) grammar

gather information to determine whether or not there are potential ambiguity to ensure generated grammars to preserve the LL(1) property. To be on the safe side, the TDX generator performs a *left recursion* check to make sure the generated grammar is an LL(1) grammar. Note that other group models such as `sequence`, `all` do not allow their child elements to have the same name under the same namespace. As result, such models do not result in a parsing conflict arising from ambiguities caused by `choice`.

It should be noted that the ambiguous grammars described here are a rare corner case encountered only when there are elements shares the same normalized names in a `xsd:choice` schema component. This is in contrast in strictly requirements in W3C XML schemas.

5.2.2 FIRST and FOLLOW sets construction for LL(1) grammar productions

$FIRST$ and $FOLLOW$ sets construction for *permutation phrase* grammar productions. The permutation phrase production holds the property that each element in the right hand side can appear in any position. This means that the composition symbols in this production are commutative and associative. This lead to the FIRST and FOLLOW sets being computed as the union of all elements in the permutation phrase production. Consider for example, a permutation production, $A \rightarrow \langle\langle X_1 \parallel X_2 \parallel \cdots \parallel X_n \rangle\rangle$, to compute the FIRST set, apply the following rules:

1. $FIRST(A) = FIRST(X_1) \cup FIRST(X_2) \cup \cdots \cup FIRST(X_n)$.
2. Add ε to $FIRST(A)$ if for all $i, X_i \stackrel{*}{\Rightarrow} \varepsilon$, where $1 \leq i \leq n$.
3. $FIRST(\langle\langle X_1 \parallel X_2 \parallel \cdots \parallel X_n \rangle\rangle) = FIRST(X_1) \cup FIRST(X_2) \cup \cdots \cup FIRST(X_n)$.
4. Add ε to $FIRST(\langle\langle X_1 \parallel X_2 \parallel \cdots \parallel X_n \rangle\rangle)$ if for all $i, X_i \stackrel{*}{\Rightarrow} \varepsilon$, where $1 \leq i \leq n$.

The $FOLLOW$ set is constructed as follow:

1. $FOLLOW(X_1) = FIRST(X_2) \cup FIRST(X_3) \cup \cdots \cup FIRST(X_n)$.
2. $FOLLOW(X_n) = FIRST(X_1) \cup FIRST(X_2) \cup \cdots \cup FIRST(X_{n-1})$.
3. $FOLLOW(X_i) = FIRST(X_1) \cup \cdots \cup FIRST(X_{i-1}) \cup FIRST(X_{i+1}) \cup \cdots \cup FIRST(X_n)$, where $1 < i < n$.

FIRST and FOLLOW sets construction for *multi-occurrence phrase* grammar productions. An occurrences constraint production defines a rule that the number of occurrences of the nonterminal constrained by a lower bound and an upper bound must be in the range of the lower and upper bounds. Because the mapping rules guarantees that the lower bound is greater than one and less than or equal to the upper bound, both the lower and an upper bounds do not change the FIRST and FOLLOW sets. Methods for constructing the FIRST and FOLLOW sets for occurrences constraint productions are the same as regular LL(1) productions. To compute FIRST set of a nonterminal $FIRST(X)$ for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then $FIRST(X) = \{X\}$.
2. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then place a in $FIRST(X)$ if for some i, a is in $FIRST(Y_i)$, and ϵ is in all of
3. If $X \rightarrow \epsilon$ is a production, then add ϵ to $FIRST(X)$.

FIRST and FOLLOW sets construction for grammar productions. To compute the set of $FOLLOW(A)$ for all nonterminals A, apply the following rules until nothing can be added to any $FOLLOW$ set.

1. Place \$ in $FOLLOW(S)$, where S is the start symbol, and \$ is the input right endmarker.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in the $FIRST(\beta)$ except ϵ is in $FOLLOW(B)$.
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where $FIRST(\beta)$ contains ϵ, then everything in $FOLLOW(A)$ is in $FOLLOW(B)$.

5.2.3 Constructing augmented LL(1) parsing table

As described above about the constructing of the FIRST and FOLLOW sets for the augmented LL(1) grammar, the construction of the parsing table is no different from the method described in [8].

5.3 Parsing Engine

A traditional table-driven predictive parser consists of an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by \$, a symbol used as a right endmarker to indicate the end of the input string. The stack contains a sequence of grammar symbols with \$ on the bottom of the stack. Initially the stack contains the start symbol of the grammar on top of \$. The parsing table is a two-dimensional array $M[A, a]$, where A is a nonterminal and a is a terminal or the symbol of endmarker \$. The parser is controlled by a driver program. However, this traditional parsing model cannot parse the *permutation phrase* and *multi-occurrence phrase* grammar productions generated from the XML schemas. TDX extends the traditional table-driven parsing model by using a streaming two-stack push-down automaton model to parse and validate the extended grammars generated from the XML schemas to efficiently parse and validate XML instances of the schemas. Parsing and validation are accomplished simultaneously

by consulting the parsing table and validation tables which store the indices to the semantic functions, either type-checking functions performing element and attribute values or application-specific APIs. Figure 5.2 shows the model of a table-driven predictive validating XML parser.

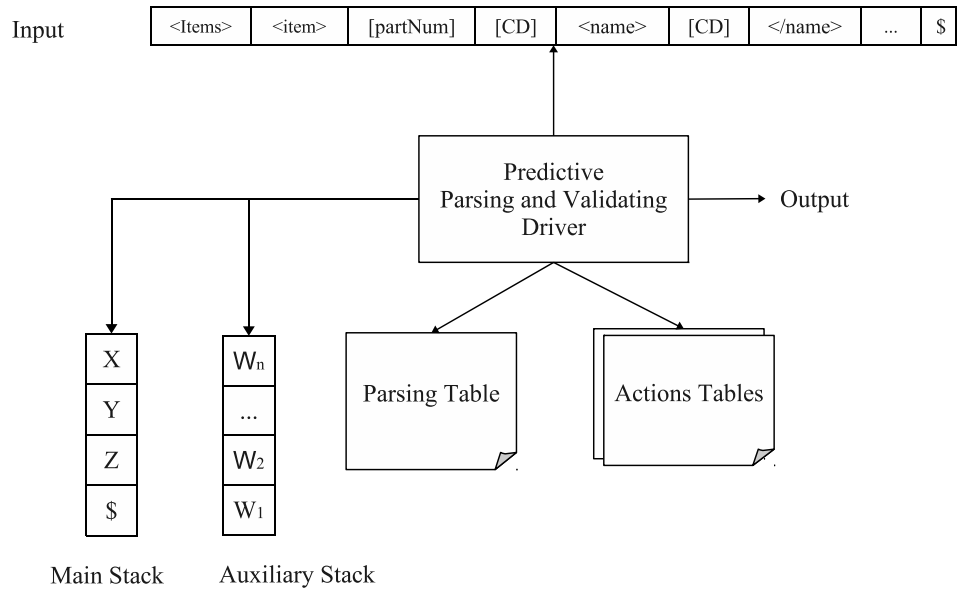


Figure 5.2: Model of a table-driven parsing and validating engine.

The streaming parsing and validating driver consumes the tokens from an input stream as soon as a token arrives to the driver, thus buffering the entire input string is not required. The main stack works in the similar way as the traditional table-driven parser for parsing the regular LL(1) grammar productions. The extra auxiliary stack is used for parsing *permutation phrase* and *multi-occurrence phrase* productions. The action tables store the type-checking functions for validating element and attribute value types and application-specific APIs to perform the application logic. To this end, TDX allows parsing, validation and application logic in a single stage. The behavior of the parsing and validating driver is driven by the states of the internal stacks, the current token of the input, and the states and current token of the input determines the unique entry of the parsing table, thus ensures independence of the driver and XML schemas. This allows the development of a high performance, generic program for the driver.

5.3.1 Parsing Permutation phrases

Permutation phrases can be parsed efficiently using a two-stack push-down automaton. we first introduce an algorithm that parses a permutation phrase without optional elements, i.e. no productions that produce empty strings, and then relax such constraints to extend the solution to cover optional elements.

Assume a permutation phrase holds the following two constraints:

- **Nonemptiness:** No constituent of any permutation derives an empty string, that is, no optional elements.

- **Uniqueness:** No constituent of any permutations shares the same first symbol, that is, $FIRST(X_1) \cap FIRST(X_2) \cap \cdots \cap FIRST(X_n) = \emptyset$ for the permutation phrase $\langle\langle X_1 \parallel X_2 \parallel \cdots \parallel X_n \rangle\rangle$.

A two-stack push-down automaton can efficiently parse the permutation phrase. The constituent symbols in a permutation phrase are pushed on to the main stack. The nonterminal on top of the main stack is checked to see if it derives the current input token. If it does, then the parser pops it off from the main stack and pushes all the symbols in the auxiliary stack onto the main stack, and empties the auxiliary stack. In this case, the current input token and the symbol that derives the current input token uniquely determine the entry of the parsing table. The parser then parses the regular production by expanding this selected production by pushing right hand side symbols onto the main stack in reverse order. The driver then checks again and consults the parsing table to replace symbols in the right hand side until the symbol on top of the main stack is a terminal. If the terminal matches the current input symbol, the terminal is popped off from the main stack and the driver reads the next input token. If the terminal does not match the current input symbol, a parsing failure is reported. If the element on top of the main-stack does not derive the current input token, it is popped off and pushed onto the auxiliary stack. If no permutation phrase symbol on the main stack can derive the current input token, the parser reports a parsing error. Therefore the parser can detect errors as early as no permutation element derives the input symbol without consuming all permutation elements.

Handling Optional Elements. Technically, there is no mechanism to transform a grammar with permutation phrases containing optional constituent elements into an unambiguous context-free grammar, because there is no way to determine where an empty constituent is derived; i.e. whether it occurs before the first nonempty element, between two nonempty elements, or after the last one. However, the order of the derivation is of no importance to a permutation phrase. As proposed by Cameron [36], the strategy to parse such a permutation phrase is: "parse nonempty constituents as they are seen until a symbol is encountered which is not a first symbol of any of the constituents remaining to be parsed." Thus, the two-stack machine adopts this strategy intuitively: all the optional elements are left in the auxiliary stack once all the nonempty elements have been parsed. Checking the symbols left in the auxiliary stack to see if all of them can derive an empty string once no permutation phrase symbol derives the current input symbol. In this situation, any symbol left in the auxiliary stack that cannot derive an empty string indicates a parsing failure and an error is therefore issued. The permutation parsing algorithm with optional elements support is given in Algorithm 1.

5.3.2 Parsing Multi-occurrence Phrases

A multi-occurrence phrase is constrained by a lower bound and an upper bound. The nonterminal along with the two bounds are first pushed onto the auxiliary stack. If the constrained nonterminal can derive the current input token, the nonterminal is expanded to a regular production that is stored in the parsing entry, which is determined by the nonterminal and the current input token. Before the nonterminal is expanded, a check against the upper bound is performed to ensure that the current value of the upper bound is greater than zero. An error is issued when the upper bound is equal to zero that indicates the number of appearances has already appeared to the upper bound. The upper bound in the auxiliary stack is decreased by one when the nonterminal has been replaced. So is

Algorithm 1 Parsing a permutation phrase production.

Ensure: p is in form $p \rightarrow \langle\langle X_1 \parallel X_2 \parallel \cdots \parallel X_n \rangle\rangle$

1: **procedure** PARSEPERM(p)
2: $pop(mainStack)$ ▷ pop off p from main stack
3: push $X_n, X_{n-1}, \ldots, X_1$ onto main stack
4: $tok \leftarrow read(input)$ ▷ read a token from input
5: **repeat**
6: $sym \leftarrow top(maiStack)$
7: **if** $M[sym, tok]$ is not an error entry **then**
8: $prod \leftarrow M[sym, tok]$ ▷ Get the production from the parsing table
9: $pop(mainStack)$
10: **while** auxiliary stack is not empty **do**
11: $t \leftarrow top(auxStack)$
12: $push(mainStack, t)$
13: $pop(auxStack)$
14: **end while**
15: call $PARSEREG(prod)$ ▷ Parsing the regular production
16: **else**
17: $t \leftarrow top(mainstack)$
18: $push(auxStack, t)$
19: $pop(mainStack)$
20: **end if**
21: **until** no nonterminal derives the current token
22: **while** auxiliary stack is not empty **do**
23: $t \leftarrow top(auxStack)$
24: **if** $not\ t \Rightarrow \epsilon$ **then**
25: $error()$
26: **end if**
27: **end while**
28: Set to the next input token
29: **end procedure**

the lower bound in the auxiliary stack when the nonterminal has been replaced. If the nonterminal cannot derive the current input token, check the current value of the lower bound in the stack to ensure that the required minimum number of the appearances has satisfied. If the current value of the lower bound in the stack is greater than zero, an error is reported. The algorithm for parsing a multi-occurrence phrase production is shown in Algorithm 2.

Algorithm 2 Parsing a multi-occurrence phrase production.

Ensure: p is in form $p \rightarrow \lfloor X \rceil_{\{m..n\}}$

procedure PARSEMULTIOCC(p)
  $pop(mainStack)$ ▷ pop off p from main stack
  push X, m, n onto auxiliary stack
  $tok \leftarrow read(input)$ ▷ read a token from input
  $upper \leftarrow top(auxStack)$
  $pop(auxStack)$
  $lower \leftarrow top(auxStack)$
  $pop(auxStack)$
  $sym \leftarrow top(auxStack)$
  if $M[sym, tok]$ is not an error entry **then**
    $prod \leftarrow M[sym, tok]$ ▷ Get the production from the parsing table
    if $upper <= 0$ **then**
      $error()$
    else
      $lower \leftarrow lower - 1$
      $push(auxStack, lower)$
      $upper \leftarrow upper - 1$
      $push(auxStack, upper)$
      call $PARSEREG(prod)$ ▷ Parsing the regular production
    end if
  end if
  $pop(auxStack)$ ▷ pop off the upper bound from auxiliary stack
  $lower \leftarrow top(auxStack)$
  $pop(auxStack)$ ▷ pop off the lower bound from auxiliary stack
  $pop(auxStack)$ ▷ pop off the nonterminal
  if $lower > 0$ **then**
    $error()$
  end if
  Set to the next input token
end procedure

5.3.3 Table-Driven Parsing and Validation

Extended grammars generated from the schemas contains regular grammar productions and two extended productions, *permutation phrase* production and *multi-occurrence phrase* production.

Algorithms for predictive table-driven parsing the latter two productions using two stacks have been given. The grammar productions rather than *permutation phrase* productions or *multi-occurrence phrase* productions are referred to regular ones. The table-driven parsing algorithm for such regular productions is similar as the one introduced by Aho et al. [8]. Only one stack (main stack) is used to parse these regular productions. The driver considers X, the symbol on top of the main stack, and a, the current input symbol (token). If X is a nonterminal, the driver chooses an $X - production$ by consulting the entry $M[X, a]$ of the parsing table M, and replaces the nonterminal X with the right side hand symbols of the $X - production$. Otherwise, it checks for a match between the terminal X and the current input token a. If the entry is an error, an error is issued to indicate a parsing failure. Algorithm 3 describes the behavior.

Algorithm 3 Table-driven parsing and validating regular productions.

Ensure: X is regular symbol

```
1:  procedure PARSEREG(X)
2:      a ← read(input)                          ▷ read a token from input
3:      while X ≠ $ do                           ▷ Main stack is not empty
4:          if X = [CD] then          ▷ Special token indicating a value of text string
5:              pop(mainStack)
6:              call semantic action to do validation
7:              set to the next input token
8:          else if a is a terminal then
9:              pop(mainStack)
10:             set to the next input token
11:         else if M[X, a] is an error entry then
12:             error()
13:         else if M[X, a] = X → Y_1 Y_2 ... Y_k then
14:             pop(mainStack)
15:             push Y_k, Y_{k-1}, ..., Y_1 onto the main stack with Y_1 on top
16:             X ← top(mainStack)           ▷ set x to the top main stack symbol
17:         else if X = [WILDCARD] then      ▷ Special token indicating wildcards
18:             call SAX-based procedure to parse and validating
19:         end if
20:     end while
21: end procedure
```

To construct a table-driven validating parsing engine, a flag, F, is used to indicate the parsing state for the three different productions. The value $REGULAR\ REGULAR$ indicates the parsing state is for regular productions while PERM and $MULTIOCC$ indicate parsing permutation phrase productions and multi-occurrence phrase productions respectively. The main stack is initialized with \$, the endmarker, and S, the start symbol on top. The current symbol X is the symbol on top of the main stack, and a is the current input token (symbol) generated from a scanner. The flag, F is initialized with $REGULAR$. The predictive table-driven parsing and validating algorithm is shown in Algorithm 4.

Algorithm 4 A table-driven parsing and validating driver.

1: **procedure** PARSING(X)
2: $F \leftarrow REGULAR$
3: $push(mainStack, \$)$
4: $push(mainStack, S)$
5: $a \leftarrow read(input)$ ▷ read a token from input
6: $X \leftarrow top(mainStack)$
7: **while** $X \neq \$$ **do** ▷ Main stack is not empty
8: **if** $M[X, a]$ is an error entry **then**
9: $error()$
10: **else if** $M[X, a] = X \rightarrow \langle\langle Y_1 \parallel Y_2 \parallel \cdots \parallel Y_n \rangle\rangle$ **then**
11: $F \leftarrow PERM$
12: call $PARSEERM(X)$
13: **else if** $M[X, a] = X \rightarrow \lfloor Y \rceil_{\{m..n\}}$ **then**
14: $F \leftarrow MUTIOCC$
15: call $PARSEMULTIOCC(X)$
16: **else**
17: $F \leftarrow REGULAR$
18: call $PARSEREG(X)$
19: **end if**
20: **end while**
21: **end procedure**

5.4 Scanning and Tokenization

XML is encoded using a plain text format and manipulating strings is much more inefficient than manipulating integers, for example. The TDX scanner scans the XML messages, identifies namespace prefixes, tag names, text values and namespace declarations as string tokens, and converts these string tokens into integral tokens for the parsing engine. XML namespaces are extensively used in XML documents. Namespaces qualify XML element and attribute names by using a namespace-prefix declaration. The declaration takes the form of a special attribute with a reserved prefix (`xmlns`) followed by the prefix to be declared. The value of this attribute is the declared namespace of URI. The scope of the namespace declaration includes the enclosing element, all of the sibling attributes, and the element's content. This arrangement, although natural, presents some difficulties for XML scanners. During scanning, namespace declarations prevent the qualified names of the element and its attributes from being conclusively known until the end of the tag. This means that scanning of qualified names in XML requires infinite look-ahead to fully resolve names. Consider the following XML instance:

```
1 <n1:e1 xmlns:n1="http://example.org/ns1" xmlns:n2="http://example.
     org/ns1">
2   <n2:e1 a="1" b="2"...z="26" xmlns:n2="http://example.org/ns2">
3   </n2:e1>
4 </n1:e1>
```

Because the namespace declaration can be redeclared in a start element, scanners typically use a stack to store namespace prefixes and the corresponding URIs. The number of defining `xmlns` namespace bindings in an XML message is typically much smaller than the number of uses of this namespace prefix. As a result, maintaining the stack can result in several comparison operations, and can hurt the overall performance of the de-serialization module. Processing of `xmlns` attributes can be optimized by using just one table lookup to determine a corresponding internal namespace prefix. The table stores the hash values of the tags and namespaces by exploiting the schemas at compile time. To this end, the TDX scanner implements a schema-directed scanner. The namespace stack can simply record the translated prefixes to provide efficient matching of qualified tags and avoid storage and expensive comparison of namespace URIs.

In addition, the TDX scanning can also be optimized by the state of the parsing engine stack. A terminal on top of the main stack indicates the specific token is expected. Scanning the specific text representing the token is much more efficient than searching the arbitrary string and then breaking it into a token.

In our early work on designing table-driven parser, a schema-directed FLEX description of the scanner is fed to FLEX to generate a DFA-based scanner in C. However, the major disadvantage of this approach is that it breaks the independence of the scanner from the schema. When a schema is updated, recompilation is required for the parser or the application. As a result, the high degree of flexibility offered by the TDX framework is undermined.

5.5 Pipelined Scanning, Tokenization, and Parsing and Validation

Parallel XML parsing methods [118, 119, 120, 143, 144] have been studied to gain performance speedup by leveraging the multi-processor or multi-core architectures. Such approaches typically

involves multi-pass of the XML message: a master node performs scanning to gain topological structure of the elements of the document to guide the partitioning, then partitions the document into chunks and sends those chunks to different processing nodes to parse the chunks. A master node finally merge the partial results to accomplish the parsing. Furthermore, communication among the processing node, in particular between the parsing nodes and the master node are required. Moreover, scanning the single tooted XML document is inherently sequential. As a result, the speedup gained in such approaches is not promising. In addition, such approaches are not applicable to streaming XML where pre-scanning the stream to gain the skeleton of the structure of the document. In this section, we present our parallel approach by pipelining the different stages, i.e., scanning, tokenization, and parsing and validation.

It is observed that our table-driven approach consists of several different stages: scanning, tokenization, parsing and validation. Those stages can be pipelined and executed simultaneously. Such an approach parallelizes XML parsing and validation in a high level to gain the speedup. This differs from the traditional parallel XML parsing approach in that it does not require partitioning or merging partial results, thus extra scanning and communication costs are eliminated. Figure 5.3 illustrates the scheme of pipelined scanning, tokenization, and parsing and validation.

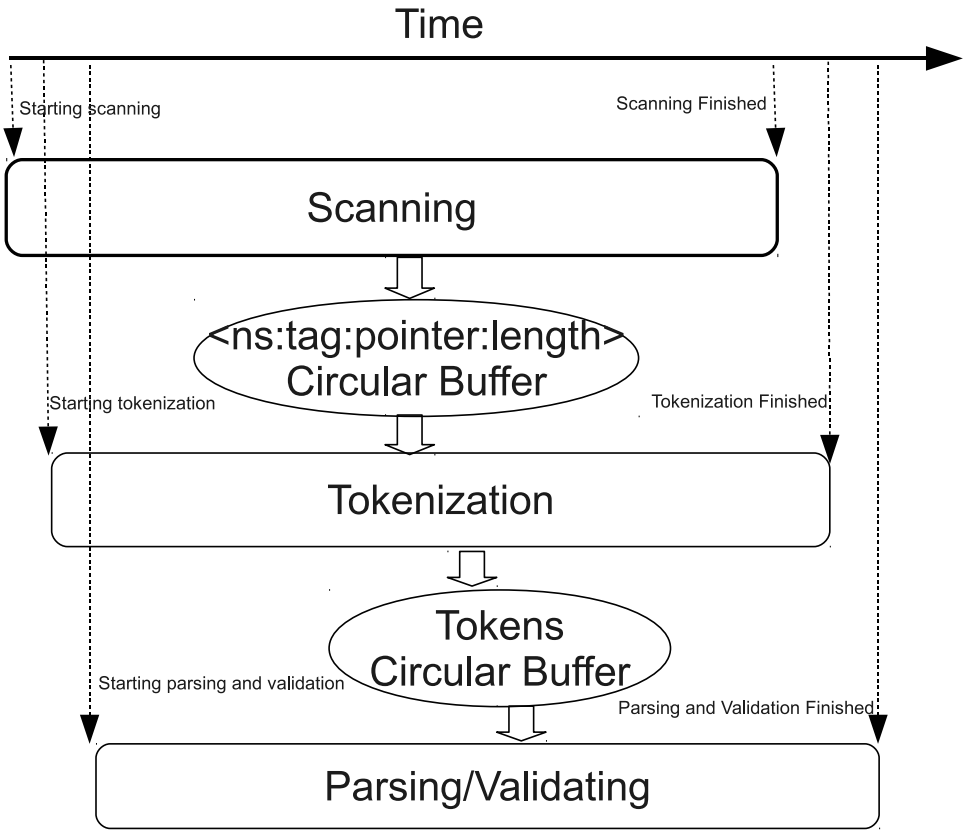


Figure 5.3: Pipelined scanning, tokenization, parsing and validation.

A scanner scans the XML stream (or XML document) and breaks the XML message into a sequence of tuples, consisting of a normalized namespace, a tag name, and an optional pointer with a

length that represents a pointer to a start position of the text value and its length for attribute value and element text value respectively, and enqueues these sequence of tuples into a ring buffer. As soon as the ring buffer has contents to be tokenized, a tokenizer starts to consume the namespaces and tag names and converts these pairs into integral tokens, and puts those tokens into a ring buffer containing those tokens representing the namespace-tag names. Similarly, as soon as the token buffer contains token to be parsed and validated, a streaming parsing engine consumes these tokens and performs parsing and validation in a single stage. As a result, scanning, tokenization, parsing and validating are performed simultaneously, thus to gain performance speedup. According to Amdhal's law, the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. In our earlier work on designing a table-driven parser [209, 208], the breakdown in scanning, parsing, and validation overhead with TDX parsing and validation is reported and compared to other XML parsers. The analysis shows that scanning time can be up to two times slower than parsing and validation. This is because scanning of XML stream is inherently sequential and identifying and normalizing namespaces poses another challenge. Thus the speedup of our pipelined approach is dominated by the scanner.

The use of ring buffers offers efficient memory usage in our TDX. This provides developers with capability to configure the sizes of the buffers. They are not necessary to be the same. For devices with limited memory resources like mobile phones, small buffer sizes may be preferred while for servers that have more memory available the size can be set to a larger number to maximize the performance.

CHAPTER 6

TABLE-DRIVEN XML STREAM SEARCHING

This chapter describes the Table-Driven validation XPath query processor — called TDXPath — for efficiently evaluating XPath query expressions over XML streams. TDXPath integrates XML parsing, validation and query searching into a single pass without backtracking. This novel approach is based on the efficient table-driven XML parser, TDX. By pre-processing one or a set of given XPath query expressions and marking the parsing table entries at compile-time, an efficient specialized streaming query evaluation engine performs the parsing, validation, searching and returns the matching results of the XML streams at runtime. Like TDX parser, the query engine is independent of XML schemas and XPath query expressions. As a result, marked parsing table (query table, thereafter I use both interchangeably) can be populated on-the-fly.

Because the query expressions are pre-processed and encoded in the query table, the query engine efficiently evaluates the queries with various XPath features.

6.1 Introduction

Due to its key advantages of interoperability and extensibility, XML [205] has become the *de facto* for messaging among industries, academias and organizations, in particular for loosely coupled organizations. An increasing amount of information is becoming available in XML form, stored as files or transmitted as streams. This has spawned significant interests in the techniques for searching XML messages.

XPath [193] was recommended by the W3C as a standard language for addressing parts of XML of user's interests and is an integral component of languages for XML processing such as XSLT [53], and lies at the core of the XQuery [197] language for XML messages. Given the central role that XPath plays in the XML stack, efficiently evaluating XPath expressions is essential.

Traditional XML searching implementations on XML messages reflect the XPath specification or XQuery language and usually incur exponential run-time complexity. XPath expressions have been used as a general-purpose mechanism for accessing portions from XML documents, for example, an XPath-based API is provided in DOM 3 [201] for traversing DOM [189] trees. The DOM-based XPath engines such as the one provided with Xalan [13], require that an entire document be in memory before evaluating an XPath expression. For large documents, this approach may result in unacceptable overhead. Furthermore, the XPath engine in Xalan may perform unnecessary traversals of the input document. Consider for example, an expression such as `/descendant::x/ancestor::y`, which selects all `y` ancestors of `x` elements in the document. The Xalan XPath engine evaluates this

expression by using one traversal over the entire document to find all the `x` elements, and for each `x` element, a visit to each of its ancestors to find appropriate `y` elements. As a result, some elements in the document may be visited more than once. Moreover, the requirement of building the in-memory objects of the entire document before query evaluation starts incurs inefficiency of memory usage and introduce large delay. Such delay may result in unacceptable in performance and time critical environments such as low-latency and high-frequency trading systems.

In many environments, it is natural to treat the data source as a stream. Streaming data are available for reading only once and are provided in a fixed order determined by the data source. Applications that use such data cannot seek forwardly or backwardly in the stream, and cannot revisit a data item seen earlier unless they are buffered. However, such a buffering requirement leads to inefficiency of memory space complexity as in the DOM-based approaches. Examples of streaming data include real-time news feeds, stock market data, sensor data, surveillance feeds, and data arriving in from networks. Some data are available in only streaming form because they have a limited lifetime of interest to most consumers. For example, articles in a topical news feed are not likely to retain their value for very long. Moreover, the data source may lack resources to provide non-streaming access. For example, a network router that provides real-time packet counts, error reports, and security alerts is typically unable to fulfill the processing or storage requirements of providing non-streaming access to such data. Therefore, XPath query engines that efficiently evaluate XPath expressions over XML streams are increasingly in need.

The premise of streaming XPath is that in many instances XPath expressions can be evaluated in one depth-first, document-order traversal of an XML document or a stream. The benefits of streaming XPath engines are twofold. First, rather than storing the entire document in memory, only the portion of the document relevant to the evaluation of the XPath is stored, thus resulting in efficient usage of memory space. Second, the streaming algorithms visit each node in the document exactly once, avoiding unnecessary traversals, leading to high-performance.

In the past decade, there have been a number of efforts to build efficient large-scale XML searching systems over XML streams. Most approaches are finite state automata (FSA) based. While most of these automaton-based systems support both structure and value matching to some extent, they have tended to emphasize either the addressing XML structure matching (for example, [11, 40, 77, 78, 150]), or the processing of value-based predicates (for example, [62]).

For structure matching, a useful approach has been to adopt some forms of Finite State Machine (FSM) to present path expressions in which location steps of path expressions are mapped to machine states. Arriving XML messages are then parsed by an *event-based* parser (for example, SAX); the events raised during parsing are used to drive the FSMs through their various transitions. A query is said to match a document if during parsing, an *accept* state for that query is reached. This approach of to XML filtering was first presented in XFilter system [11].

XFilter employs a dynamic index over the states of the query FSMs and includes optimizations that reduce the number of path expressions that must be checked for a given document. In large-scale systems, there is likely to be significant commonality among user interests, which could result in redundant processing in *XFilter*. *CQMC* improved upon XFilter by building an FSM for a set of queries identical in structure. XTrie [40] further supports shared processing of certain common sub-stringd of the path expressions. *YFilter* [57, 58, 59] exploits sharing even more aggressively to reduce the number of automoton states by using a single combined FSM to represent all path expressions. Although it supports both the structure matching and predicate processing, *YFilter* only support a subset of XPath features, for example, it lacks supporting query expressions consisting of

backward axes (reverse axes that contain `ancestor` or `parent`).

Some of these automaton-based streaming XPath evaluation algorithms process XPath queries containing the child axis (`/`), descendant axis (`//`) and wildcard (`*`) [15, 91], however, they do not support predicates since an FSA is memory-less, as observed in [148].

XSQ is pushdown transducers augmented with buffers and auxiliary stacks to minimize the buffering. A notable feature of *XSQ* is that at any point during query processing, the data that is buffered by *XSQ* must necessarily be buffered by any streaming XPath query engine. However, this approach lacks capability of supporting all the XPath feature too.

Automaton-based methods exhibit memory inefficiency due the large number of machine states. Current automaton-based streaming query engines are limited to handle a subset of path expressions. Additionally, some of the XPath features, for example, the arbitrary position predicate function position()=500,cannot be presented efficiently using automata due to the expressive power of automata.

In addition, these implementations typically sit on top of external SAX parsers, thus introducing an extra processing layer of the query processor. Depending on the underlying external SAX parsers such as *expat* [169], these engines lack supporting XML validation. Even the underlying external XML parsers are capable of validating, they are typically disabled in favor of the system performance, because it is well known that validation incurs significant processing overhead in the XML parser at the receiving server [117, 179], in particular for validating XML streams. However, validation is important, because applications that use the XML data typically require the data to be modeled according to a schema.

Furthermore, current streaming query engines build the automata on-the-fly. Regardless of its flexibility, that is, no re-compilation is required when the query expression is changed. The on-the-fly construction of automata does incur significant overhead at runtime.

Lastly, to the best of my knowledge, no current streaming query engine exploits the XML schema information to speedup the performance.

I present a novel approach for efficiently searching XML streams based on my validating Table-Driven XML parser (TDX). Table-Driven validating XPath query processor — called TDXPath. TDXPath supports full features of the core XPath language including all the axes (`child`, `descendent`, `parent`, and `ancestor`), `wildcards`, `predicates`, `position()` and `text()`. It also supports multiple query expressions and XML namespaces. TDXPath integrates XML parsing, validation and query evaluation into a single pass and without backtracking, thus eliminating the overheads introduced by separation of parsing, validation and query evaluation. TDXPath is a schema-specific XPath query engine because it is optimized by schemas at compile-time.

At the core of the TDXPath is the specialized streaming query engine that performs parsing, validation and query evaluation simultaneously by consulting the query table at runtime. TDXPath exploits the XML schemas and encodes the parsing states in a compact table, and marks the parsing table entries to indicate the query states by processing the query expressions at compile-time. Since the query engine only consults the query table at runtime, that is, the engine is independent of the schemas and query expression, the query table is hot-swappable. Thus, when a schema is changed or extended, or when a query expression is changed or a new query expression is added, a new query table can be updated and loaded at runtime. This offers a significant degree of flexibility. Thus TDXPath is suitable for scalable and extensible systems. Consider for example, a publish/subscribe system is usually a long term service to deliver information to its users according to their interests presented by query expression. Such a system must be designed to be scalable, extensible and flexible to accept new users and new expressions and allow updates of the query expressions.

Each XPath query expression selects a single or a set of concrete element node (children), or content texts of element nodes, wildcards are replaced with concrete nodes, attributes or texts and marked in the query table at compile-time. As a result, wildcards evaluation does not burden the query engine at runtime. Backward and forward axes are processed in the same way. Thus different queries matching the same results (equivalent queries thereafter) mark the same query table entries. Thus TDXPath exhibits a consistent high-performance.

Similarly, multiple queries and operator `AND` are treated a single query expression. Equivalent queries in such queries only require to mark the entries once and thus they are evaluated only once.

The aforementioned features offered by TDXPath and the streaming characteristics of the query engine ensure TDXPath offers large scalability.

6.2 Overview of Table-Driven XPath Query Processor

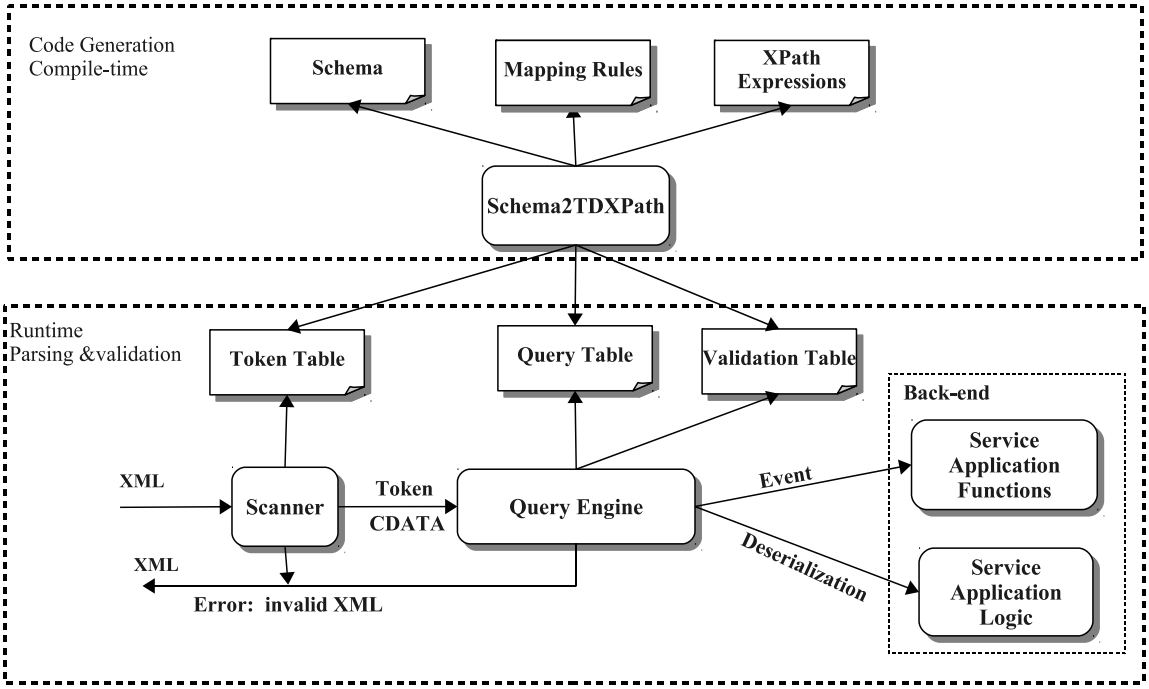


Figure 6.1: System architecture of a table-driven XPath query processor

The TDXPath system architecture is depicted in Figure 6.1. Two phases are required to build a TDXPath streaming XML query processor: compile-time phase and runtime phase. At the compile-time phase, the generator *Schema2TDXPath* first generates a token table, a parsing table and a validation table from a set of schemas by consulting a set of mapping rules. It then marks the parsing table entries for selected grammar productions that match the query patterns to a single or a set of XPath expressions. Functions for verifying the predicates in the query expressions are also generated. Indices of these validation and predicate verification functions are stored in the validation table. The use of indices ensures the query engine is independent of schemas and query expressions, thus

offers high degree of flexibility. When a schema or a query expression is updated, new functions can be generated and loaded dynamically at runtime without re-compilation and re-deployment of the application.

At runtime, the scanner reads from an XML streams, breaks this stream into tokens for the element tags (start-tags and end-tags), attributes, content texts (attributes and elements), namespace prefixes and namespace values by consulting the token table, and feeds these tokens (or content texts if any) to the query engine.

Upon receiving a token, the engine performs parsing, validation and query evaluation by consulting the query table. Because the query table encodes the XML structure constraints imposed by the schema(s), well-formedness checking is integrated and accomplished automatically. XML data-type validation is accomplished by invoking the validation functions indexed in the validation table when special tokens are met.

At the same time, queries are evaluated when the current entry is marked. Predicates are also checked to see whether or not the predicates are satisfied by invoking the predicate functions indexed in the validation table. Flags are set or unset depending on the conditions. If the entry is marked and the predicates are satisfied, the matched results are then delivered to the back-end application logic. If the entry is marked, yet one or more predicates have not been evaluated, the engine must buffer the messages because they are potential candidates for the matched results. This happens only for the axe with forward axes or siblings, either explicitly in the queries or implicitly generated from wildcards.

Like the TDX, there are also two modes for TDXPath: static mode and dynamic mode. The former tends to maximize the performance by trading performance with flexibility. The latter trades flexibility with performance due to initial overheads caused by on-the-fly population of the tables and re-loading the validation and predicate functions at runtime. Note that the overheads only happen one time when a schema or a query is updated or new queries are added.

6.3 Table-Driven XML Stream Query Evaluation

This section describes the table-driven XPath query evaluation (TDXPath) on XML streams which integrates parsing, validation and query evaluation into a single stage. A streaming query evaluation engine performs parsing, validation and query evaluation as an XML stream arrives in by consulting the query table that encodes the parsing and query evaluation states. Details of constructing of the query table are introduced in Section 6.4. In this section, an overview of query processing are given first by examples to illustrate the intuition on how the query engine evaluates queries over an XML stream. Then query engine and algorithms are the introduced.

TDXPath is a schema-specific query processor that exploits XML schema information at compile-time to speedup the parsing, validation and query evaluation. Given a set of XML schemas and XPath query expressions, TDXPath first generates a parsing table from the schemas by consulting the mapping rules (refer to Chapter 4 for details) at compile-time; then converts the parsing table into a query table by preprocessing the query expressions at compile-time (refer to 6.4). The query table encodes the states for the query engine to perform parsing, validation and query evaluation at runtime.

Consider for example, given an XML schema (see A.1), an LL(1) grammar is generated (Table 6.1). In this grammar, upper case letters are used denote nonterminals. Lower case letters such as a denote terminals. Lower case letters with hats such as $\hat{a}$ denote terminals for end-tag element

tokens corresponding to their start-tag tokens. A special character, τ, denotes a special token for text contents.

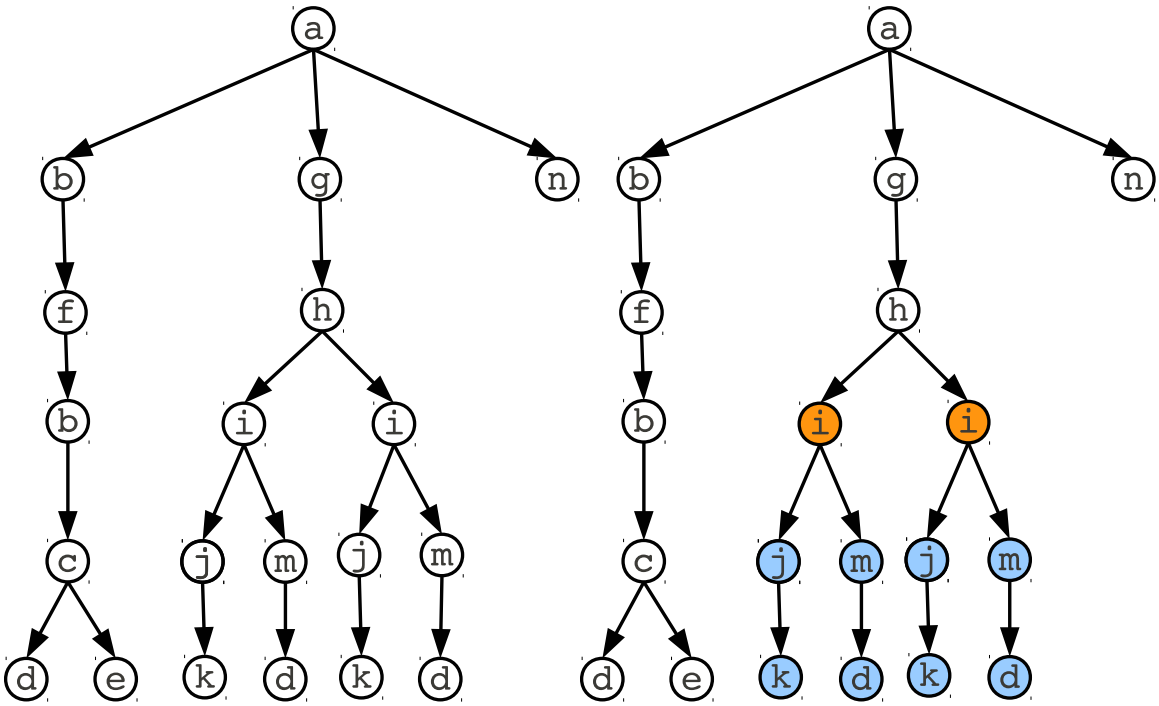
Table 6.1: LL(1) grammar that is generated from the XML schema A.1 (Upper case letters denote nonterminals; lower case letters such as a denote terminals; lower case letters with hats such as $\hat{a}$ denote terminals for end-tag element tokens; Special character, τ, denotes special token for text contents).

Index	Production		
1	A	$\rightarrow$	a B G N $\hat{a}$
2	B	$\rightarrow$	b B' $\hat{b}$
3	B'	$\rightarrow$	C
4	B'	$\rightarrow$	F
5	C	$\rightarrow$	c D E $\hat{c}$
6	D	$\rightarrow$	d τ $\hat{d}$
7	E	$\rightarrow$	e τ $\hat{e}$
8	F	$\rightarrow$	f B $\hat{f}$
9	G	$\rightarrow$	g H $\hat{g}$
10	H	$\rightarrow$	h I $\hat{h}$
11	I	$\rightarrow$	I' I
12	I	$\rightarrow$	ϵ
13	I'	$\rightarrow$	i J M $\hat{i}$
14	J	$\rightarrow$	j K $\hat{j}$
15	K	$\rightarrow$	k τ $\hat{k}$
16	M	$\rightarrow$	m D $\hat{m}$
17	N	$\rightarrow$	n τ $\hat{n}$

Location Steps Query Evaluation. An instance of the schema can be efficiently parsed and validated using TDX (see Chapter 5). In addition to the parsing and validation, TDXPath evaluates the XPath queries by consulting the query table. An XPath query$Q1$: `/a/*/i` selects all the elements `i` including sub-nodes (descendants) of these matched nodes `i`. Figure 6.2 shows an XML instance of the schema (see A.1) and the matched elements of the XML instance on an XPath query expression `/a/*i`. Note that the bottom left element `d` is not a member of the matched results because it is child element of element `c` and it is not a descendant of element `i`.

The query table is constructed by marking the parsing table to encode the query states for query expressions. This is accomplished by consulting the grammar and query expressions. Table 6.2 shows the query table for the query $Q1$: `/a/*/i`. The entry $M[I', i]$ that represents the selected node by the query `/a/*/i` is marked with a check marker (✓). Other marked entries (marked with $\uparrow [I', i]$) are different from this marked entry in that they are descendants of the selected node `i` and are dependent of the selected node only. The marked nodes may depend on some node paths and whether or not constraints imposed by predicates are satisfied. For example, the entry $M[K, k]$ (marked with $\uparrow [I', i]$) indicates that evaluation of this marked entry depends on whether or not the entry $M[I', i]$ is satisfied. In addition to performing evaluation of the marked entries, TDXPath performs parsing and validation.

To evaluate the query `/a/*/i` over the instance in Figure 6.2, when the first start-tag `d` is encountered, because it is a marked entry and depends on the entry $M[I', i]$. The engine checks



(a) An instance of the XML schema A.1 (b) Matched nodes on the query `/a/*/i`

Figure 6.2: An XML instance of the schema A.1 and matched nodes on the XPath query `/a/*/i`

Table 6.2: Query table for the query `/a/*/i` (Some terminals are not shown in the table to reduce the tabe size).

Nonterm	Terminal												
	a	b	c	d	e	f	g	h	i	j	k	m	n
A	1												
B		2											
B'			3			4							
C			5										
D				$6 \uparrow [I', i]$									
E					7								
F						8							
G							9						
H								10					
I									11				12
I'									13✓				
J										$14 \uparrow [I', i]$			
K											$15 \uparrow [I', i]$		
M												$16 \uparrow [I', i]$	
N													17

whether or not the entry $M[I', i]$ is evaluated to *True*. Because the element has not been encountered and the flag has not been set to *True*, it is conclusively determined this first item d is not included in the results. The engine continues to process the incoming tokens. Upon a start-tag token comes in, and it is a marked entry for the selected node. Because no predicates or path constraints is imposed to this entry, the engine determines it is a member of the results; thus sets its flag to *True*and delivers this item to the output results. Because it is determined, as a result, no buffering is required. When the start-tag <j> is met, and again this entry is marked and it depends on the entry $M[I', i]$ which is already set to *True*, this item is determined to be a part of the results. When the element <k> is encountered, and it is a marked entry that depends on the entry $M[I', i]$ which is already evaluated to *True*, and thus it is also a partial result.

It evaluates the element m as exactly the same as does element j. It then evaluates the element d as a start-tag <d> is encountered, however, unlike the first element d, this time its dependent entry $M[I', i]$ is already evaluated to *True*, thus it is determined to be a partial result.

Upon an end-tag is encountered, the flag is set to *False*. The coming element i is evaluated exactly as evaluating the first appearance of the element i.

As this example demonstrates, TDXPath evaluates this query efficiently without buffering even it consists of a wildcard ($\star$). It is also observed the query is evaluated without backtracking.

Table 6.3: Query table on the query /a/$\star$/i[/a//c/e/text()="Hello"] (Some terminals and nonterminals are not shown to reduce the table size).

Nonterm	Terminal										
	a	b	c	d	e	f	i	j	k	m	n
A	1										
B		2									
B'			3			4					
C			5 $\leftarrow [E, e]$								
D				6 $\uparrow [I', i]$							
E					7 $\uparrow [I', i]$ $\rightarrow [C, c]$ $\leftarrow [I', i]$ ϕ						
F						8					
G											
H											
I							11				12
I'							13✓ $\rightarrow [E, e]$				
J								14 $\uparrow [I', i]$			
K									15 $\uparrow [I', i]$		
M										16 $\uparrow [I', i]$	
N											17

Predicates Evaluation. Predicates in a query expression is marked in a similar way as the query location steps. For a query $Q2$: `/a/*/i[/a//c/e/text()="Hello"]`, it selects all the elements `i` that must have a element `e` satisfying the predicate `/a//c/e` and its text value of element `e` must be equal to `"Hello"`. In addition to mark the location step (`/a/*/i`) as shown in Table 6.2, the predicate is also marked in the query table to constraint the path dependencies and satisfactory of the value constraint. An action boolean function is also generated to verify the text value for element `e`. The function index is stored in the action table to be invoked. The query table is listed in Table 6.3.

The entry $M[I', i]$ that represents the node selected by the query $Q2$ is marked with a checker marker ($\checkmark$), and with a dependency marker ($\rightarrow [E, e]$) because the query expression in constrained by a predicate. When a terminal `i` is encountered and the marked entry is evaluated, whether or not this item is a partial result depends on the satisfactory of its dependency entries. The item is a partial result of matched result only when all its dependency entries are satisfied. The entry $M[E, e]$ on which the marked entry depends is marked with a dependency entry ($\rightarrow [C, c,)]$ and a value evaluation boolean function (ϕ). The satisfactory of this entry depends on the evaluation of both the function and its dependency entry. The marker ($\leftarrow [I', i]$) indicates that the entry $M[I', i]$ depends on the evaluation result of this entry. The engine will set the entry $M[I', i]$ of the evaluation result. As a result, when the entry $M[I'i]$ is evaluated, satisfactory can be determined without looking back at the entry $M[E, e]$. This entry is also marked with a sub-selected node marker ($\uparrow [I', i]$) which indicates this entry may be included in the matched results and only depends on the satisfactory of the entry $M[I', i]$.

The table also indicates that the entry $M[C, c]$ is marked with a marker ($\leftarrow [E, e,)]$ indicating the entry $M[, e]$ depends on the satisfactory of this entry. When this entry is evaluated, the evaluation result is set to the entry $M[E, e]$ to speedup the performance.

Specifically, when a start-tag <c> is encountered, the entry $M[C, c]$ is evaluated, the flag of the entry $M[C, c]$ is set, and a counter in the entry $M[E, e]$ is increased by one. A positive counter means the constraints the entry are satisfied. Upon an end-tag </c> is encountered, the counter of the entry $M[D, d]$ is decreased by one if the flag of the entry $M[E, e]$ is set. The flag of the entry $M[C, c]$ is then reset.

Upon arrival of a start-tag token <d>, and the entry $M[D, d]$ is evaluated. This entry depends on the satisfactory of the entry $M[I'i]$. If an element <d> is child of the element <c>, the entry $M[I', i]$ has not been evaluated and it is determined it not a partial result of the matched results. If an element <d> is child of the element <m>, the entry $M[I', i]$ has been evaluated. Depending on the satisfactory of the entry $M[I', i]$, it may or may not be a partial result of the matched results.

When a start-tag token <e> is encountered, the counter is checked and the boolean function is invoked to check if the value constraint is satisfied. The flag is set to *True*or false depending on the satisfactory of the counter and the function. The satisfactory of the marked entry $M[I', i]$ is set accordingly.

When a start-tag token <i> is encountered, the engine checks the flag status of the entry $M[I', i]$ to see if the constraints are already satisfied. If the constraints are satisfied, this item is determined to be included in the matched results. Otherwise, this item is not a partial result of the matched results. Other marked nodes ($\uparrow [I', i]$) are sub-selected nodes that depends only on the satisfactory of the marked entry $M[I', i]$.

Queries Evaluation Requiring Buffering. As illustrated by the aforementioned two examples, neither does TDXPath require buffering potential elements on evaluating location step queries

without predicates. Nor does TDXPath do on evaluating queries with predicates. This is one of the advantages of TDXPath, that is, minimizing the buffering for most commonly used query expressions. Minimizing buffering is a major challenge on query evaluation over XML streams. Current XML stream query algorithms often presents poor performance on the memory usage due to the requirement of buffering potential candidates on evaluating the queries such as `/a/*/i[/a//c/d[text()]]` until their membership of the matched results is determined, that is, until the predicate satisfaction is resolved.

By preprocessing the location step queries at compile-time, TDXPath dose not result in buffering on evaluating location step queries without predicates. Moreover, TDXPath requires no buffering on evaluating many predicates. However, buffering cannot be avoided on evaluating some predicates over XML streams. Consider for example, an XPath query $Q3$: `/a/*/i[/a/n]` over the XML message in Figure 6.2a, which selects all elements `i` that the root element `a` has a child element `n`. Because the element `i` and its descendent elements appear after the element `n`. When the element `i` is encountered, its predicate constraint cannot be determined until the element `n` is met. As a result, TDXPath buffers the these potential candidates until the predicate is determined. As soon as the element `n` is encountered, their membership is determine and the buffer is cleared.

6.3.1 Query Evaluation Engine

TDXPath query evaluation engine has the same architecture as the TDX parsing engine (Section 5.3). The table-driven query evaluation engine consists of an input stream buffer, two stacks, a query table, action tables and an output stream. At the core heart is the parsing, validation and query evaluation driver. The input buffer contains the stream tokens to be parsed, validated and evaluated to determine whether or not the items are included into the matched results. Parsing and validation are performed in the same way the TDX parsing engine does. As soon as the engine has completed parsing and parsing and validation, a query evaluation is then performed if an entry of the query table is marked. Value-based boolean functions are accomplished by invoking the functions indexed in the action tables that are generated from XPath query expressions with predicates. The tables including the action tables that contain indices of validation functions and value-based boolean functions are hot-swappable. The specialized query engine is independent of the schemas and queries. As a result, tables can be populated on-the-fly, and thus offering a mechanism to address the changes of schemas and queries. Figure 6.3 shows the model of a table-driven validating query evaluation engine.

The streaming parsing, validating and query evaluation driver consumes the tokens from an input stream as soon as a token arrives in. As a result, buffering the entire input string is not required. The main stack works in the similar way as the table-driven parser parse and validate the regular LL(1) grammar productions. The extra auxiliary stack is used for parsing *permutation phrase* and *multi-occurrence phrase* productions. The action table stores the type-checking functions and value-based functions for predicates to validate element and attribute value types and value-based boolean functions. To this end, TDXPath integrates parsing, validation and query evaluation in a single stage. The behavior of the driver is driven by the states of the internal stacks, the current token of the input. The states and current token of the input uniquely determine the entry of the query table. It does not depend on XML schemas or query expressions, thus ensuring the independence of the engine. When an entry is marked, query evaluation is performed either to compute the satisfaction of the queries or to determine whether or not the current is item is a member of the matched results.

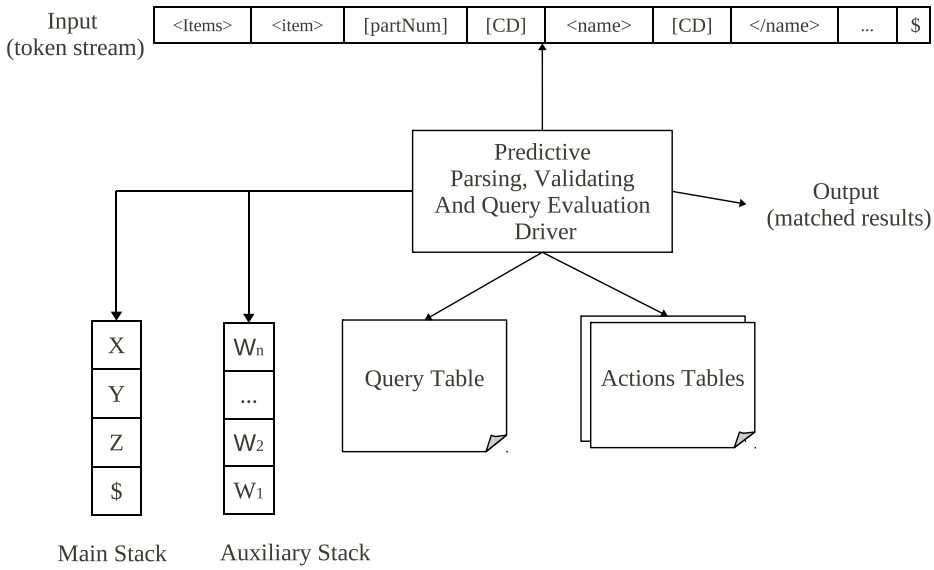


Figure 6.3: Model of a table-driven streaming validation query evaluation engine.

6.3.2 Query Evaluation Algorithm

A query table encodes query evaluation states for the query engine to perform query evaluation. Marked entries of the query table may contain one or more of the following states:

Algorithm 5 Query evaluation.

```
1: procedure EVALUATEENDTAG(M,X,a)   ▷ A query table M, a nontermial X and an
   end-tag terminal a
2:     if is_marked = true then                ▷ Evaluates marked entries only
3:         if a is a start-tag terminal then   ▷ Processing start-tag
4:             EVALUATESTARTTAG(M,X,a)
5:         else
6:             EVALUATEENDTAG(M,X,a)
7:         end if
8:     end if
9: end procedure
```

- A flag that indicates whether or not the entry is marked, denoted by is_marked. It is not necessary and performance inefficient to mark all the entries of the query table.
- Marked selected node, attribute or value-based text content of an element or attribute, denoted by the symbol (✓).
- Marked node that is descendant nodes of the marked selected node, denoted by the symbol ($\uparrow [i, j]$), where $[i, j]$ is the entry of the marked selected node.

Algorithm 6 Evaluating dependent path entry.

Ensure: $M[X, a]$ is marked with $\leftarrow [i, j]$

```
1: procedure EVALUATEDEP(M, X, a, i, j)
2:     if (M[X, a] is marked with φ AND φ= true) OR M[X, a] is NOT marked with φ
   then
3:         ← [X, a] ← true
4:         → [i, j] = true
5:         if M[i, j] is marked with ✓AND its is_met= true then
6:             Eevaluate matched of M[i, j]
7:             if Matched then
8:                 Output the items buffered already
9:             end if
10:            Clear the buffer
11:        end if
12:    else
13:        ← [X, a] ← false
14:    end if
15: end procedure
```

- A matched state for the marked selected node indicating the node is determined to be a partial results. This is used for its descendant nodes of the selected node.
- Dependent node that some nodes are on the path on which the selected node depends, denoted by $\rightarrow [i, j]$, where the current marked entry has a dependence of the entry $[i, j]$.
- Reverse dependent node that indicates the current node depends on, denoted by the symbol ($\leftarrow [i, j]$), where the entry $[i, j]$ has a dependence of the entry marked.
- A set of flags for value-based boolean functions, denoted by symbol (ϕ).
- A stack used for tracking the position of a node, denoted by pos. A simple counter for the position of a node does not work for the nested elements.
- A counter, denoted by $counter$. A marked entry may come from more than one paths. A set of status or flags is required to track satisfaction for different paths. Maintaining and manipulating such a set of status lead to inefficiency. This can be resolved by a counter. A positive counter indicates at least one path is encountered and thus satisfied.
- A counter, denoted by $and_counter$. A marked entry may come depend on multiple value-based boolean functions for multiple paths.
- A flag that indicates whether or not the node on the path is encountered, denoted by is_met.

A terminal representing the current input token and a nonterminal on top of the stack uniquely determines an entry of the query table. When the entry is expanded and then processed, the query

Algorithm 7 Evaluating Selected entry.

Ensure: $M[X, a]$ is marked with ✓ $M[i, j]$

```
1:  procedure EVALUATEQUERY(M,X,a)
2:      if M[X, a] is marked with then
3:          Evaluate matched                    ▷ either true, false, or undetermined
4:          if matched = true then
5:              Output the item as partial result
6:          elsematched = undetermined
7:              Buffer the item
8:          end if
9:          if M[X, a] is marked with ↑ [X, a] then
10:             if matched of M[X, a] = true then
11:                 Output the item as partial result
12:             elsematched of M[X, a] = undetermined
13:                 Bufferthe item
14:             end if
15:         end if
16:     end if
17: end procedure
```

engine first performs the parsing and validation. If the entry is marked for query evaluation, the behavior depends on the status of the marked the entry.

If the entry is marked as an entry that other entries depend on, its state is_met which is initialized to *False* is set to *True*. If there is no predicates required to evaluation or the predicates are satisfied and the terminal is a start-token, the state $counter$ of the entry that depends on the current entry is increased by one. If the terminal is an end-tag, this state $counter$ is decreased by one if the predicates are satisfied. Its is_met state is always to set to *False* when the terminal is an end-tag.

Upon a start-tag token is encountered, if the entry is marked as a selected entry and its $counter$ is positive, and $and_counter$ equals to the total number of the predicates of XPath `AND` operator, the entry is determined to be a member of the matched results. Its matched status $matched$ is set to *True*. Upon its end-tag token is encountered, the matched status is reset to *False*.

When a start-tag is encountered and the entry is marked as an entry whose node is dependent node of the selected nod, the node is included in the results if the status $matched$ is set.

The position state pos is processed using a stack. Upon a start-tag token of the entry is encountered, the top counter on the stack (current level counter) is increased by one, and its current counter is decreased by one upon arrival of an end-tag token. A $push()$ or $pop()$ operation is performed depending on the production of the entry is a recursive or non-recursive rule. No $push()$ or $pop()$ is required for non-recursive productions because it requires initialization only once. For the recursive productions, a $push()$ or $pop()$ operation with the counter initialized to zero is executed upon a start-tag and end-tag of its parent node respectively. The algorithm of the TDXPath driver is shown in Algorithm 5.

Upon evaluating an entry, It first checks whether or not the entry is marked for evaluation. Only marked entries are processed. Depending on the type of a terminal, which is either a start-tag or

Algorithm 8 Evaluating marked entry upon a start-tag terminal.

Ensure: a is a start-tag terminal

procedure EVALUATESTARTTAG(M,X,a) ▷ A query table$M$, a nontermial X and a start-tag terminal a
  $is\_met \leftarrow true$ ▷ Set is_met
  if $M[X, a]$ is marked with pos **then**
    Evaluate pos
  end if
  if $M[X, a]$ is marked with ϕ **then**
    Evaluate ϕ
  end if
  if $M[X, a]$ is marked with $\leftarrow [i, j]$ **then**
    EVALUATEDEP(M, X, a, i, j) ▷ Evaluate dependent path entry
  end if
  if $M[X, a]$ is marked with **then** ▷ Evaluate selected entry
    EVALUATESEL(M,X,a,i,j)
  end if

end procedure

Algorithm 9 Evaluating marked entry upon an end-tag terminal.

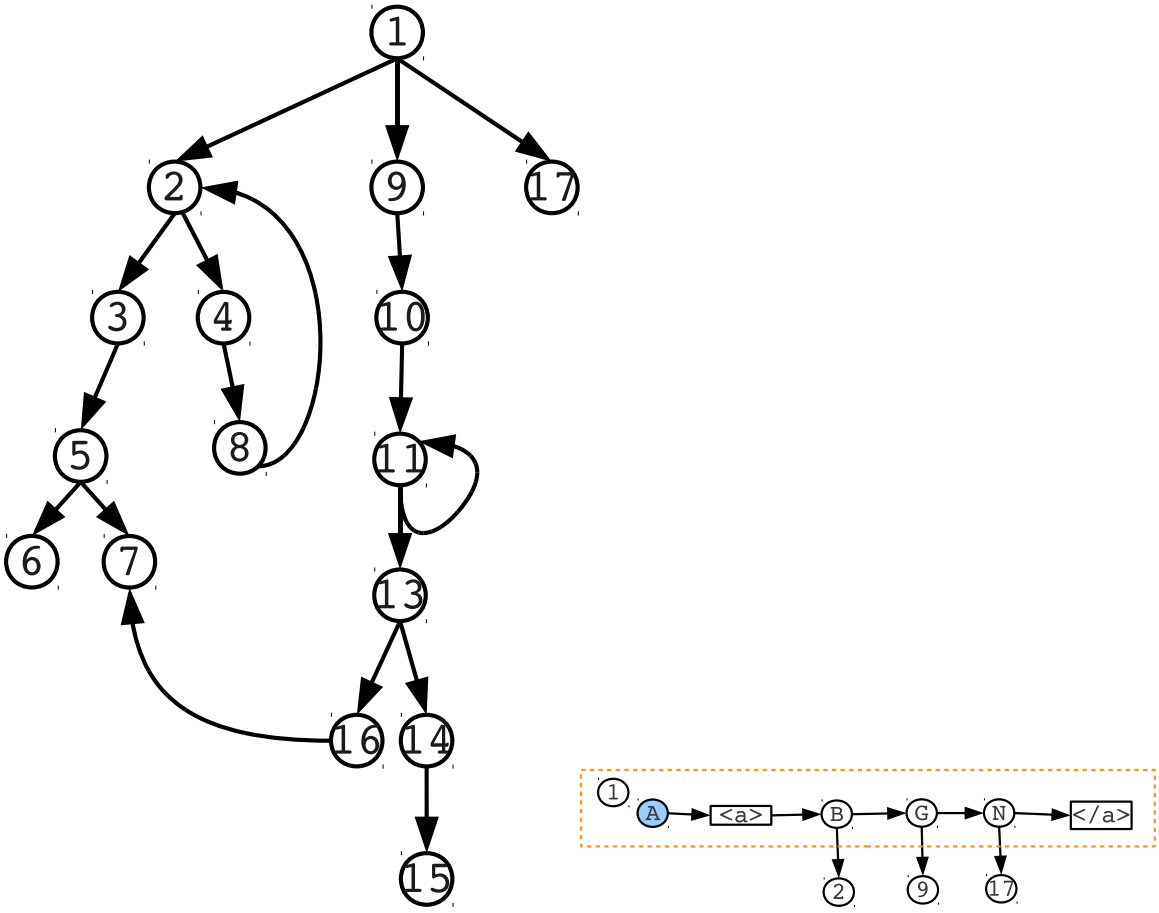
Ensure: a is an end-tag terminal

procedure EVALUATEENDTAG(M,X,a) ▷ A query table$M$, a nontermial X and an end-tag terminal a
  if $M[X, a]$ is marked with ✓ **then**
    $matched = flase$
  end if
  if $M[X, a]$ is marked with $\rightarrow [i, j]$ **then**
    if $is\_met = true$ AND $couter$ of $M[i, j] > 0$ **then**
      $counter = counter - 1$
    end if
  end if
  if $M[X, a]$ is marked with pos **then**
    Process position
  end if
end procedure

an end-tag for an element, a terminal for an attribute, or a special token, the algorithms behaviors differently. It then call different procedures to perform the evaluation. As the algorithm illustrates, buffering is minimized and as soon as the satisfaction is evaluated and the candidates are determined whether or not they are members of the matched results, the buffer is cleared out.

6.4 Construction of a Query Table

This section presents the query table construction. An algorithm to build a graph from an LL(1) grammar is first described and algorithms to mark a parsing table into a query table are then presented.



(a) A graph of the grammar in Table 6.1 (Node represented using production indices for simplicity).

(b) The internal structure of the node 1.

Figure 6.4: A graph of the grammar in Table 6.1 (Parent link are not shown).

6.4.1 LL(1) Grammar Graph

An augmented LL(1) grammar that is generated from one or a set of XML schemas can be viewed as a directed graph $\mathcal{G} = (V, E)$, where V is the set of vertices and E is the set of edges. The vertices correspond to the grammar productions and the edges represent the relationships between the productions. Figure 6.4 shows the graph constructed from the grammar (Table 6.1).

Given an LL(1) grammar $\mathcal{L}$, a graph corresponding to the grammar $\mathcal{G}$ can be constructed. A node representing the start production is created first. For each nonterminal in the right hand side of the production, find all the productions whose left hand side nonterminal is this terminal. If the production does not exist in the graph, create a node as a child of this nonterminal. Otherwise, set this production node as a child of this nonterminal. Repeat this procedure for all the these newly created nodes until no new nodes need to be created or no more links are set. Epsilon productions can be ignored because they are not used for constructing a query table. The algorithm to construct a graph from an LL(1) grammar is shown in Algorithm 10.

Algorithm 10 Build a graph from an LL(1) grammar.

```
1:  procedure MAKEGRAPH(G, L)                    ▷ A graph G and an LL(1) grammar L
2:      Q ← all the nonterminals in the right hand side of the starting production
3:      Create a node for the starting production
4:      while Q is not empty do
5:          N ← Q.deque
6:          for each production P whose left hand side nonterminal is N do
7:              if node P is found then
8:                  set the node as a child of N
9:              else
10:                 create a new node for P as a child of N
11:                 for each nonterminal N' in the right hand side of P do
12:                     Q.enque(N')
13:                 end for
14:             end if
15:         end for
16:     end while
17: end procedure
```

6.4.2 Query Table Construction

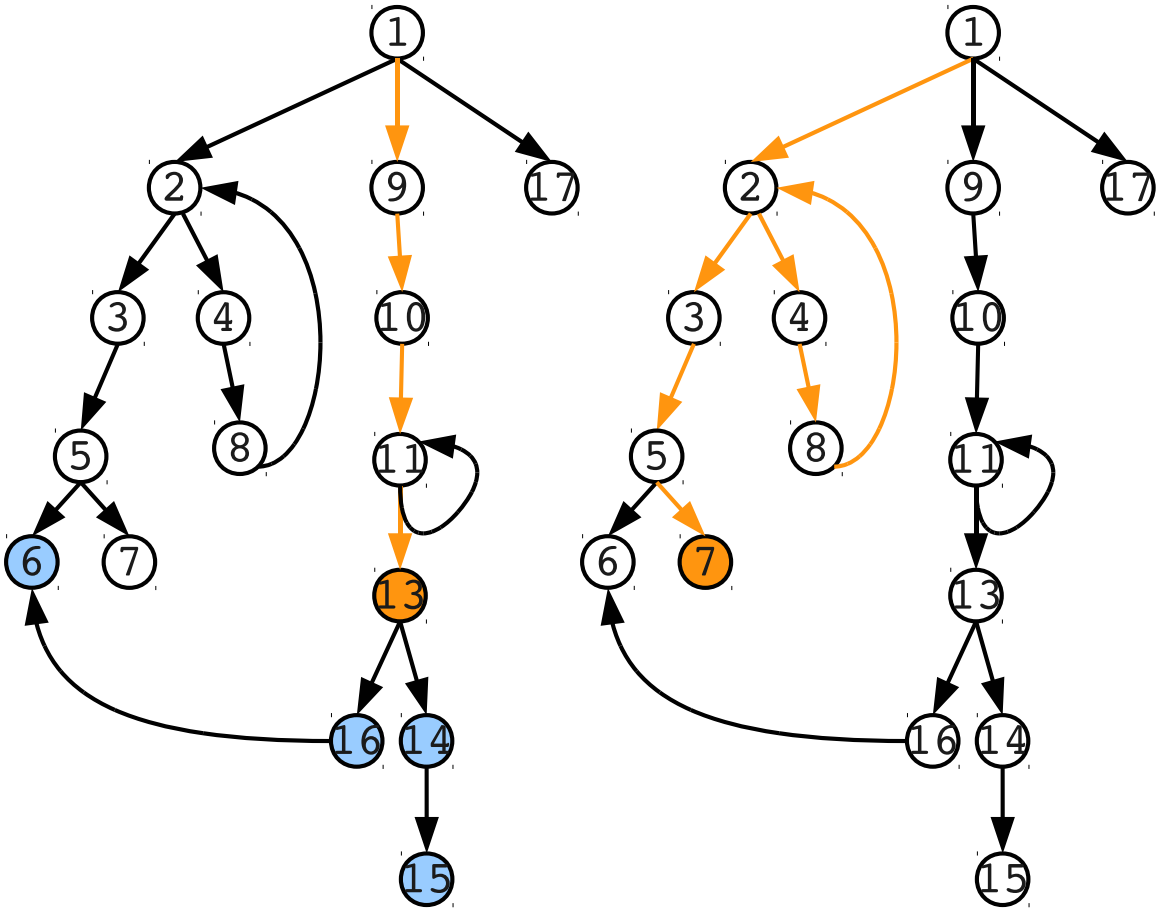
In this section, I present the algorithms to convert a parsing table into a query table for one or a set of queries. An illustrating example is first described to gain the intuition of the algorithms.

Consider for example, the query $Q2$: `/a/*/i[/a//c/e/text()=''Hello"]` used in Section 6.3, find all the paths to the starting production in the graph (Figure 6.4) for the location step part of the query `/a/*/i`. Figure 6.5 shows the paths (highlighted).

The path highlighted in Figure 6.5a implies the dependencies for evaluating the query which selects the element `i` including all its descendants. The query table can be marked for each of the node in the path ($1 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 13$). For example, the node 10 depends on the node 9 which

represent productions $G \rightarrow g\ \tau\ \hat{g}$ and $H \rightarrow h\ \tau\ \hat{h}$, respectively. The entry $M[H, h]$ which depends on the entry $M[G, g]$ marked with a marker ($\rightarrow [G, g]$), and the entry $M[G, g]$ is marked with a marker ($\leftarrow [H, h]$) to indicate that there is an entry $M[H, h]$ which depends on it.

It is observed that only the nodes in the generated augmented LL(1) grammar that directly starts with terminals in its right hand side of the production need to be marked. For instance, although the node 13 depends on the node 11, this node 11 can be ignored because its right hand side consists of all nonterminals I' and I, which doses not start with a terminal. The entry of the query table for the selected production (node 13) is determined by the left side hand symbol of the production and the start-tag token of the element (attribute or special token for character data, i.e., text value), and it is marked with a check marker (✓). In general, the entry need to be marked is uniquely determined by the left hand side symbol of the production and the terminal in the $FIRST$ of the production. In addition, because there is only one path for this selected node 13, all the nodes in this unique path can be eliminated. As a result, table size can be reduced and evaluation performance can optimized.



(a) Paths of the location step `/a/*/i` (b) Paths of predicate `[/a//c/e/text()]`

Figure 6.5: Paths of query `/a/*/i[/a//c/e/text() = ``Hello"]`

The number of the marked entries can be reduced because paths may share a common suffix. Consider for the paths highlighted in Figure 6.5b: $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 7$ and $1 \rightarrow 2 \rightarrow 4 \rightarrow$

$8 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 7$, both denote the path dependencies for the predicate `/a//c/e/text{}`. Rather than marking all the nodes in both paths, they share a common suffix: $2 \rightarrow 3 \rightarrow 5 \rightarrow 7$. It is sufficient to mark the nodes in the common suffix to reserve the path dependencies. The number of marked node can be further reduced. It is observed that there is only one path from the starting production node 1 to the node 7, as a result in this example, the marked path can be further reduced to $5 \rightarrow 7$.

For example, In Figure 6.5a, the two paths $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$ and $1 \rightarrow 3 \rightarrow 7 \rightarrow 2 \rightarrow 4 \rightarrow 5$. Both of the paths share the path $2 \rightarrow 4 \rightarrow 5$. The minimum common suffix $4 \rightarrow 5$. As a result, only these two nodes are required to be marked. An other path for the node 5 is $1 \rightarrow 8 \rightarrow 10 \rightarrow 11 \rightarrow 13 \rightarrow 16 \rightarrow 5$ does not share a common suffix with either of the two paths except with the node 5. However, it is observed that the node 16 is unique in the graph. Thus only the node 16 is required to be marked in the query table. Similarly, for the selected node 13, only the node 11 needs to be marked.

In addition, a boolean function is generated for each operator such as `text()=``Hello"` to verify the satisfaction. Its corresponding entry is marked with ϕ.

Marking Query Table for a Query Path. Given a query path represented as a list $P : 1 \rightarrow 2 \rightarrow \ldots i \cdots \rightarrow N_{n-1} \rightarrow n$, where i is a production index and n is the index of the production to be marked as the selected element node (attribute or text value of a element or an attribute). This path defines a chain of dependencies for the selected node. Let $\mathcal{S}(n)$ be the set of the indices of productions that generate descendant elements of production n. Let $\mathcal{L}(i)$ denote the left hand side symbol of the production i. Let $\mathcal{T}(i)$ be the first terminal in the right side hand of the production i. The algorithm to mark a query table M is shown in Algorithm 11.

The last node in the path P represents the node to be selected. Thus its corresponding entry is marked with $\checkmark$. If this node is an element, then its all descendant nodes' corresponding entries are marked to be selected but only depend on the selected node's corresponding entry. Other nodes are path dependencies and they marked accordingly to indicate constrained dependencies.

Constructing paths for a Single Query Without Predicates. Given an XPath query expression with no predicates, $Q : A_1 N_1 A_i N_i \ldots A_n N_m$ where A_i denotes an axis and N_i presents a node, and a graph $\mathcal{G}$ constructed from an LL(1) grammar using the Algorithm 10. The paths can be generated using the Algorithm .

The last node, N_m, represent the node to be selected. Find all the nodes, $N - n$, in the graph first. For each of such node fond in the graph, traverse the graph starting with this node to generated all the paths. During the traversal of the graph, wildcards ($\star$), ancestors, parents, descendants (`//`) are replace with child axis (`/`).

As aforementioned, the generated paths can be processed to reduce the number of the nodes in the paths. As a result, the marked number of entries will be reduced. This optimization leads to memory space efficiency and runtime evaluation efficiency.

Once the paths are optimized, the query table can be marked for each path using the Algorithm 11.

Constructing Query Table for Predicates. Marking a query table for a query with predicates is break into two stages. A predicate consists of two parts: a query without predicates and a set of predicates that need to be verified. For example, a predicate `/a/b[position()<3]/c/text()="Hello"`, the query without predicates is `/a/b/c` and two predicates are `position()<3` and `text()="Hello"`. Marking the query table for a query with no predicate is accomplished by applying the Algorithm 12.

Algorithm 11 Mark a query table for a path.

```
1: procedure MARKPATH(M, P)                          ▷ A query table M and a path P
2:     for each node i in P do
3:         L ← L(i)                                  ▷ Get the left hand side of the production n
4:         t ← T(i)        ▷ Get the first terminal of the right hand side of the production n
5:         if i is the last node in P then           ▷ The last node is the selected node
6:             Mark M[L, t] with ✓                   ▷ Set the entry marked for the selected node
                 ▷ Get the set of productions for its descendant nodes of the selected node
7:             s ← S(i)
8:             if t is a start-tag of an element then   ▷ The selected node is an element
9:                 for each j in s do          ▷ Mark all its descendant entries with ↑ [L, t]
10:                    N ← L(j)                          ▷ Get nonterminal symbol
11:                    u ← T(j)                          ▷ Get terminal
12:                    Mark M[N, u] with ↑ [L, t]
13:                end for
14:            end if
15:        else                                          ▷ dependent path node
16:            j ← next node in P                        ▷ Nest node in the path
17:            N ← L(j)                                  ▷ Get nonterminal symbol
18:            u ← T(j)                                  ▷ Get terminal
19:            Mark M[L, t] with ← [N, u]
20:            Mark M[N, u] with → [L, t]
21:        end if
22:    end for
23: end procedure
```

Algorithm 12 Mark a query table for a query with no predicates.

```
1: procedure MARKQUERY(M, G, Q)      ▷ A query table M, a graph G and a query Q
2:     n ← L(Q)                  ▷ Get the selected node in the query Q, typically N − m
3:     N be a set of nodes ,n, in the graph, G.
4:     Let P be a set of paths                          ▷ A set for the paths
5:     P ← φ                                            ▷ Initialization
6:     for each node i in N do     ▷ Find all the paths for each of the appearance of node
7:         p be a path ending with node i               ▷ Find a path that ends with i
8:         P ← P ∪ p                                    ▷ Add this path to the set P
9:     end for
10:    Let P′ be the optimized set of paths▷ Eliminate the unnecessary nodes in the paths
11:    for each path p in P′ do
12:        MARKPATH(M,p)                                ▷ Mark each path
13:    end for
14: end procedure
```

For the predicate parts, a boolean function is generated for each predicate, and it is marked in its corresponding entry. The entry is determined by the left side hand symbol of the production that generates the element the predicate belongs to. The terminal is the start-tag of the element that this predicate belongs to. Consider the example the query $Q2$: `/a//c//e/text()="Hello"` that is used in this chapter, the predicate `text()="Hello"` belongs to the element `e`. The production that derives this element is $E \rightarrow e\ \tau\ \hat{e}$. The left hand side nonterminal is E and the start-tag terminal of this production is e. Therefore the entry is $M[E, e]$. A single query expression may contain multiple predicates. Each predicate is processed in turn. The algorithm is shown in Algorithm 13.

Algorithm 13 Mark a query table for a query with predicates.

```
1:  procedure MARKPREDICATE(M, G, P) ▷ A query table M, a graph G and a query Q
2:      Let q be the query of Q without predicates ▷ Extract the query from Q by
    eliminating predicates
3:      MARKQUERY(M,G,q) ▷ Mark the table for query q
4:      Let P be a set of predicates extracted from Q
5:      for each predicate p in P do ▷ Process each predicate
6:          Generate a boolean function for the predicate p
7:          r be the production index that derives the element the predicate belongs to
8:          N ← L(r) ▷ Get the left hand side symbol
9:          t ← T(r) ▷ Get the terminal
10:         Mark entry M[N, t] with φ ▷ Mark the entry to check the boolean function
11:     end for
12: end procedure
```

Constructing Query Table for Multiple queries. Processing multiple queries is no different from processing a single query. Apply the algorithm for query with predicates (Algorithm 13) for each query of the multiple queries.

CHAPTER 7

PERFORMANCE EVALUATION

This chapter presents the performance evaluation of TDX parser and TDX XPath query processor (TDXPath) compared with other parsers and query processors respectively that are in common available. The goals of this experimental study including validating our TDX implementation in C++, characterizing its features and performance, and providing an exploratory description of the features and performance of the systems that are related to our TDX. We stress that our experiments are not designed for a head-to-head benchmark style comparison of the systems. Given the diversity of the systems in goals, supported XML schemas and query languages and features, implementation languages and environments, etc., such a comparison would not be easy for all of the XML parsers and XPath query processors. Moreover, I would like to gain some qualitative insights into the cost of supporting certain XML schema components (such as `xsd:all`) and certain XPath features (such as backward axes and wildcards etc.).

7.1 Experimental Setup

To the best of my knowledge, there are no standard or widely-used XML benchmarks for W3C XML schema validating parsers or for XPath queries. Although a few of XML and XPath benchmarks have existed, none of these is suitable for measuring the full system features that our TDX parser and query processor present. These benchmarks include XMLBench [46], XMark [160], and XParthMark [68]. Therefore, I conducted experimental study using schemas that vary in a variety of features that are likely to influence the performance, like data types, sequential and unordered schema groups. To this end, I augmented the XML schemas, XPath query expressions from those benchmarks by introducing real and synthetic datasets with various data types and more complex schema components like `xsd:all`, `xsd:sequence` and `xsd:choice` and generated XML documents that differ in various size ranging from 1 KB to 10 MB. XML schemas are listed in Appendix A. To the same reason, we conducted XPath Query performance using queries that vary in a variety of features that are likely influence performance, such as query length, number of predicates, different types of axes, wildcards and different queries that match the same set of query results.

My benchmark presents XML schemas that contain various commonly-used different structural data types. I aimed to measure the performance of the parsers and the query processors for different schema structures. To this end, I chose industry-quality XML-based Web services and heuristic XML data that contain different structures and data types for the experiments. XML instances in different sizes for each schema. Each instance consists of a string of UTF-8 XML content stored in continuous

Name	Streaming	validation	Compiled-Schema
TDX	✓	✓	✓
gSOAP	✓	✓	✓
XSD-XML	✓	✓	✓
Libxml2	✓	✓	-
VTD-XML	-	-	-
Expat	✓	-	-

Table 7.1: System features of parsers compared.

memory buffer. File system I/O or network overhead is non-existent. Elements of `xsd:all` are randomly arranged in the message instance for accurate measurements of free-ordered property. For the same reason, attributes in `xsd:attribute` also were placed in a random way. Multiple instances were parsed from separate buffers to avoid any effect possibly caused by high cache hit rates. XML instances large in size were generated from the schema using the XML generator in [160]. The first run was intended to warm up the system and was discarded. Average parsing time of a hundred runs was reported. Time was measured as CPU time using system call `gettime()` except the test for multi-threaded TDX pipelining that were measured in real time elapsed using the same system call. All tests reported here were conducted on a server with 8 Intel ® Xeon® Processors @1.86 GHZ with 8GB of main memory running the Linux kernel 2.6.18. All the parsers and XPath query processors were compiled using g++ version 4.1.2 with option -O2 except the YFilter were run on SUN Java (version 1.5.0). Throughput (MB/S) were calculated by the size of the XML data set divided by parsing or query processing time. No application-specific events were triggered in the measurements, although TDX offers the capability to trigger such events.

7.1.1 Parsers Compared

I compared our TDX parser with three widely-used runtime-based parsers, Xerces [67], Expat [169] and Libxml2 [188]. Xerces is an industry widely used, popular high-performance parser. It supports both validating and non-validating parsing mode with capability of schema caching. We measured performance with validation in SAX mode which is more efficient in both runtime performance and memory consumption than in DOM mode. We chose Xerces with version of 2.7.0 for Linux.

Expat is a non-validating streaming XML parser that only checks well-formedness of the input XML message. It is considered one of the fastest non-validating parsers. The latest version of 2.0.1 for Linux was chosen for the performance comparison. The version of the Expat compared is 2.0.1.

VTD-XML (Virtual Token Descriptor for eXtensible Markup Language) [207] refers to a collection of cross-platform XML processing technologies centered around a non-extractive [206] XML, "document-centric" parsing technique called Virtual Token Descriptor (VTD). It is claimed to be the next generation non-validating XML parser, indexer, editor, slicer, assembler and XPath-engine that goes beyond DOM, SAX and PULL in performance, memory usage, and ease of use. The compared XML non-validating parser was written in C++ of version 2.10.

Name	Streaming	Validating	Multiple	Predicates	Wildcard	Backward Axis
TDXPath	✓	✓	✓	✓	✓	✓
VTD-XML	-	-	✓	✓	✓	✓
TinyXpath	-	-	✓	✓	✓	✓
YFilter	✓	-	✓	✓	✓	-

Table 7.2: System features of XPath query processors compared.

Libxml2 is the XML C parser and toolkit developed for the Gnome project (but usable outside of the Gnome platform). Libxml2 is written in the C programming language, and provides bindings to C/C++, C#, Python and other Pascals, Ruby, and PHP5. It is known to be very portable, the library builds and works without serious troubles on a variety of systems. The version we compared is 2.7.6.

I also compared two compiler-based parsers, and XSD parser [174] and gSOAP [183]. The gSOAP toolkit generates highly optimized and C-based XML validation parsers. Performance comparisons have shown that gSOAP has very fast parsers and deserializers. The presented comparisons are not completely fair for gSOAP whose timings include parsing, validation, and deserialization while other parsers do not deserialize data.

CodeSynthesis XSD is an XML Data Binding compiler for C++ developed by Code Synthesis. A validating parser can be generated from an XML schema to do the validation. The generated parser sits on the top of an external parser either Xerces-c or Expat (In our experiments, we chose the expat as the external parser which is well-known to be faster that xerces-c). The chosen version in our experiments is 3.3.0. The system features that the compared parsers are listed in Table 7.1.

7.1.2 XPath Query Processors Compared

To evaluate our TDX query processor, we compared our TDX query processor with two c++-based XPath query processors, TinyXPath [25] and VTD-XML [207] and a Java-based XPath Filter YFilter [59].

TinyXPath is a small XPath syntax decoder written in C++ based on top of TinyXML parser [178]. It does not support XML namespace. The compared version is 1.3.1.

VTD-XML is an XML parser that enables XPath query expressions to process large XML documents. The compared VTD-XML XPath query engine was written in C++ of version 2.10.

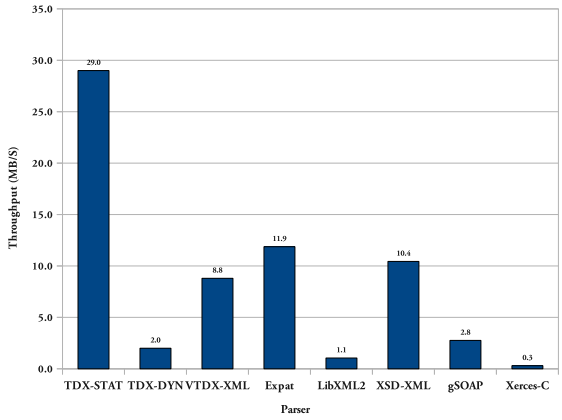
YFilter, a Java-based XML filtering system that provides fast, on-the-fly matching of XML-encoded data to large numbers of query specifications containing constraints on both structure and content. YFilter encodes path expressions using an NFA-based approach that enables highly-efficient, shared processing for large numbers of XPath expressions. Currently only one version 1.0 is available. The system features that the query processor support are listed in Table 7.2.

7.2 Performance Evaluation of TDX Parser

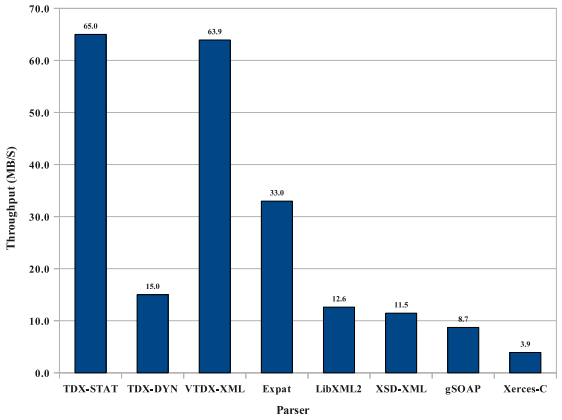
This section describes a performance study that characterizes the performance of our TDX parser compared with other XML parsers including validating vs non-validating, and schema-specific vs

runtime (non-schema-compiled) parsers. We begin in Section 7.2.1 overall performance by comparing the performance of TDX with all the parsers compared to show the overall performance. Then we investigate the characteristics by comparing to the runtime validating parsers in Section 7.2.3 and non-validating parsers in Section 7.2.4. In Section 7.2.5, we examine the implication of parsing and validating unordered elements (`xsd:all` and `xsd:attribute`) in XML datasets that typically present the poor performance in traditional parsers. The scalability performance is presented in Section 7.2.6. The overhead of the dynamic TDX compared to the static TDX is presented in Section 7.2.7. The performance improvement of pipelined TDX is shown in Section 7.2.8.

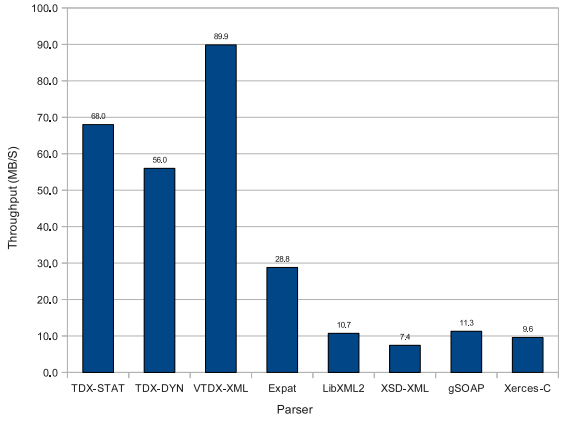
7.2.1 Overall Performance



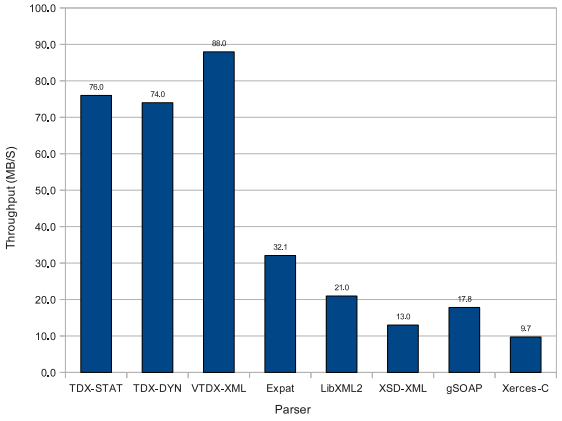
(a) Throughputs over the Structure dataset (1KB)



(b) Throughputs over the PurchaseOrder dataset (10KB)



(c) Throughputs over the Amazon-all dataset (1MB)



(d) Throughputs over the Amazon-seq dataset (10MB)

Figure 7.1: Throughput comparison to validating and non-validating parsers over different XML datasets with various sizes from small (1 KB) to large (10 MB).

Figure 7.1 illustrates the throughputs of all the compared parsers, validating and non-validating, schema-specific and non-schema-specific, over XML messages with different structures of various sizes (1 KB, 10 KB, 1 MB and 10 MB). The results indicate that our static TDX (TDX-STAT) outperforms all the validating parsers in all the test cases. It is even 3x faster than expat, a non-validating streaming XML parser. In our experiments, xerces-c (SAX) is the slowest parser among all the parsers tested. TDX-STAT is between 7x and 96x faster than validating xerces-c (SAX). Specifically, TDX-STAT is 7x faster than xerces-c for parsing and validating the `PurchaseOrder.xml` of size 1 MB (Figure 7.1c). It can be up to 96x faster than xerces-c (SAX) for parsing and validating `Structure.xml` XML messages of 1 KB in size (Figure 7.1a). Xerces-c is a non-schema-specific parser that requires multiple passes of XML messages and access to the schemas at runtime to perform parsing and validation. In contrast, our TDX-STAT pre-compiles the parsing states, stores the states in a tabular form, and pre-compiles the validating functions at compile-time, thus leading no requirement of access to the schemas at runtime. Furthermore, TDX looks up the tables at runtime is constant. As result, high-performance of parsing and validating is achieved.

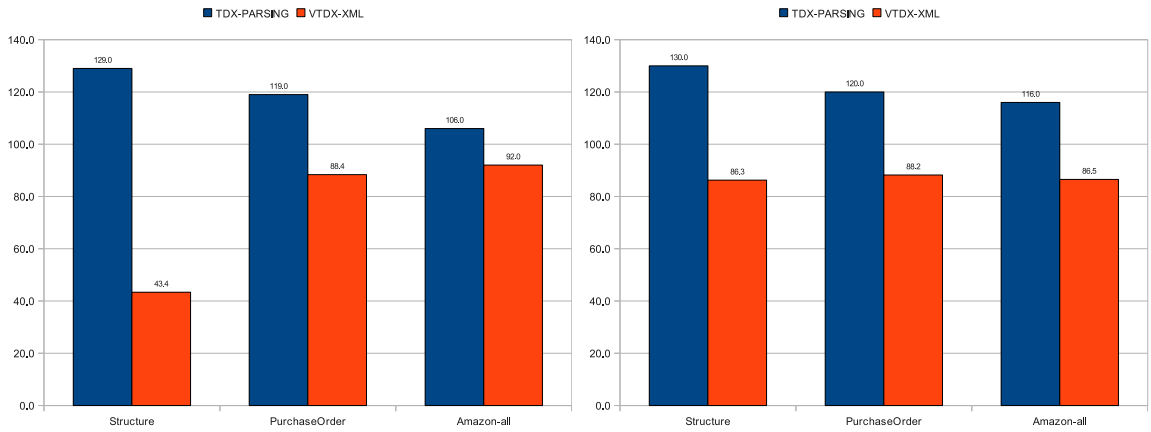
The performance of TDX-STAT is between 6x and 10x faster than schema-specific XML parser gSOAP with namespace support. gSOAP, the DOM-based schema-specific parser, also performs serialization that other parsers reported here do not implement. To this end, it may not be fair to simply compare its throughput to other parsers. However, as presented in our previous paper [209, 208], TDX parser is still several times faster than gSOAP, which was our goal. From the charts, we can also see gSOAP is up to 3x faster than xerces-c in SAX mode, which demonstrates the advantages of schema-specific parsers.

TDX-STAT is between 3x and 10x faster than XSD XML parser. XSD XML parser is very similar to our TDX in the way generating the validating functions from the schemas at compile-time. However, XSD parser sits on top of external parsers (supporting either expat or xerces-c, in our experiments expat was used). This differs from our TDX-STAT in that TDX-STAT combines the validation and parsing into one stage that eliminates the overhead of separation of parsing and validation. As the results indicate, TDX-STAT is 2x faster than Expat, thus XSD parser cannot outweighs TDX-STAT in terms of performance. Moreover, TDX-STAT leverages the schema information at compile-time and encodes the parsing states in a tabular form, leading to an extra gains of performance. Figure 7.1c also indicates that XSD does not optimize performance for parsing and validating un-ordered XML elements such as defined by XML schema components `xsd:all` and `xsd:attribute`. When processing such XML messages, the performance of XSD parser drops significantly, from 13.0 MB/S (Figure 7.1c) to 7.4 MB/S (Figure 7.1c), i.e., the performance drops 43.1%. TDX parser is optimized for permutation phrase parsing and validation by permutation phrase parsing, in contrast.

TDX-STAT runs 6x to 26x faster than LibXML2 parser in our experiments. It is observed that LibXML2 performs better when the XML messages are large in size.

As an integrated validating parser, although TDX-STAT outperforms non-validating streaming parser expat in all of our test cases, validating TDX-STAT still runs 3x faster than VTX-XML parser to XML message `Structure.xml` of 1 KB in size, and achieves almost the same throughput as VTD-XML (65.0 MB/S and 63.9 MB/S respectively) for processing `PurchaseOrder.xml` of 10 KB in size. However, non-validating VTD-XML parser can be up to 1.3x faster than our validating TDX. VTD-XML claims [207] that "XimpleWare's VTD-XML is, far and away, the industry's most advanced and powerful XML processing model for Service Oriented Architecture (SOA) and Cloud Computing". To evaluate our Table-Driven parsing techniques, we conducted experiments to perform

non-validating parsing using our TDX by modifying the validation semantic functions to simply return true. The results illustrate that out non-validating TDX outperforms VTD-XML parser. It can be up to 3x faster than VTD-XML parser (Figure 7.2).



(a) Medium datasets (100KB) (b) Large datasets (10MB)

Figure 7.2: Parsing only.

Unlike TDX-STAT encodes parsing states in a parsing table and compiles the data-type checking functions at compile-time, the dynamic mode TDX (TDX-DYN) populates the parsing table and dynamically loads the data-type checking functions on-the-fly. This incurs initialization overheads at runtime. The smaller the messages to be processed, the more impact the initialization overheads imposes. The large the messages to be processed, the less the impact it applies. As shown in Figure 7.1, TDX-DYN can be up to 15x faster than TDX-DYN when the message is very small in size (1 KB). However, when the message becomes larger, for example, both TDX-DYN and TDX-STAT achieve almost the same throughput (Figure 7.2b). We shall examine the characteristics of TDX-DYN versus TDX-STAT in detail in Section 7.2.7.

Regardless of such initialization overheads introduced by populating the parsing table and dynamically loading the data-type checking libraries in TDX-DYN, however, this mechanism offers a high degree of flexibility to address the schema changes problem that traditional schema-specific parsers cannot resolve. Traditional schema-specific parsers require recompilation and redeployment when the schema where a parer is constructed changes. When changes of a schema are present, the corresponding parsing table and the semantic actions are regenerated and populated to TDX-DYN parser, TDX-DYN only requires to construct parsing tables and reload the libraries on-the-fly. Thus eliminate recompilation and re-deployment. This offers a significant benefit for applications such as Web services that their underlying schemas may evolve and change, in particular when the down time of such a service is intolerant. Further more, such overheads are introduced only once when a schema change is present.

7.2.2 Compared to Schema-specific Validating Parsers

Throughput Performance. Figure 7.3 illustrates the throughputs of TDX relative to the schema-specific validating parsers tested over XML messages, `Structure.xml`, `PurchaseOrder.xml` and `Amazon-all.xml` XML messages in small (1 KB), medium (100 KB) and large (10 MB) sizes. Except with small XML messages (1 KB) where TDX-DYN is slower than XSD XML parser and gSOAP, both TDX-STAT and TDX-DYN are much faster than XSD XML parser and gSOAP. TDX-STAT is between 3x and 7x faster than XSD XML parser, and between 6x and 10x faster than gSOAP. TDX-DYN is between 3x and 5x faster than XSD XML parser, and between 5x and 10x the speed of gSOAP to the medium size XML messages (100 KB). Because the initialization overheads of TDX-DYN are independent of XML message size, the implication will be dropped when XML size increases. As it is shown in Figure 7.3c, TDX-DYN achieves the almost the same throughputs as TDX-STAT for processing XML messages with different structure and data types of 10 MB in size.

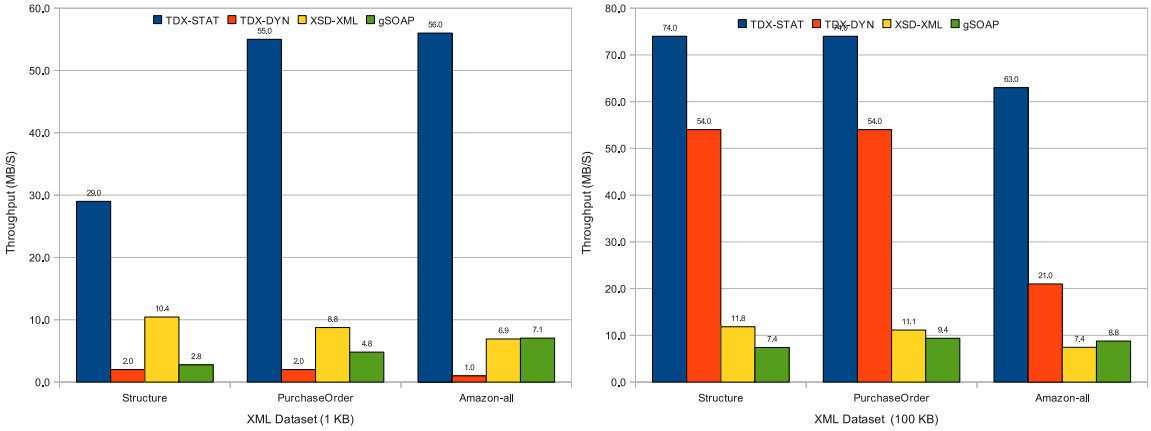
For small XML messages (Figure 7.3a), TDX-STAT is between 3x and 8x faster than XSD XML parser, and between 8x and 12x the speed of gSOAP. However, TDX-DYN is between 4x and 5x slower than XSD XMl parser, and may be up to 4x slower than gSOAP. This can be roughly determined the one-time initialization overhead of TDX-DYN because of the relatively small XML messages (see Section 7.2.7 for details).

Scalability. The scalability of schema-specific validating parsers is illustrated in terms of parsing and validation time. The Figure 7.4 shows the parsing and validating time (ms) with the increased message size from 1 KB to 10^4 KB for each of XML messages tested. The X-axis denotes XML message size, and the Y-axis indicates the parsing and validating time (logarithmic scale). The curves in Figure 7.4 illustrates TDX-STAT scales very well for all the tested XML messages with different structures and data types. When the size increases by an order of magnitude, the parsing and validation time also increases 10 times (Figure 7.4a to Figure 7.4d). Due to the initialization overheads, TDX-DYN does not exhibit good scalability for small XML messages. Because such overheads are independent of XML instances, but dependent of the schemas that define the XML instances, it gets to scale well as the size of the instance increases. The starting size of the massage starting to scale well may vary depending on the size of the parsing table and its corresponding data-type checking libraries that are constructed from the schemas, i.e., it depends on the underlying schemas from which TDX parsers are generated. The figure indicates that the starting sizes are S $10^2, 10^2, 10^3, 10^3$ KB of XML instances `Structure.xml`, `PurchaseOrder.xml`, `Amazon-all.xml`, `Amazon-all.xml`, respectively.

The results also indicate that the initialization overheads introduced by TDX-DYN dominate the processing time when the sizes of the XML instances are small. In Figure 7.4c and Figure 7.4c, the parsing and validation times remain almost constant from 1 KB to 10^2 KB. Thus TDX-DYN trades flexibility with performance for small XML messages in size.

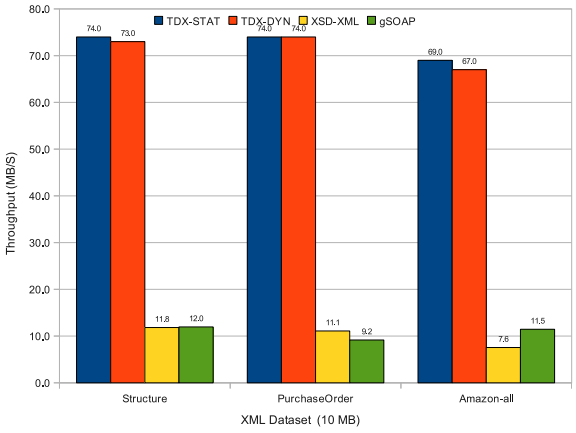
Both XSD parser and gSOAP scales well also. Although XSD outperforms gSOAP, as it is illustrated in Figure 7.4a, gSOAP has the same performance as XSD parser the size of the instance of the schema `Structure.xsd` is larger than 10^3 KB. This indicates that gSOAP may scale better than XSD parser for some XML messages despite that gSOAP is DOM-based XML parser and deserializer.

Alternatively, Figure 7.5 shows the scalability of schema-specific validating parsers in terms of throughputs. Except parsing and validating small XML instances of `Structure.xsd`, the throughputs of TDX-STAT vary slightly.



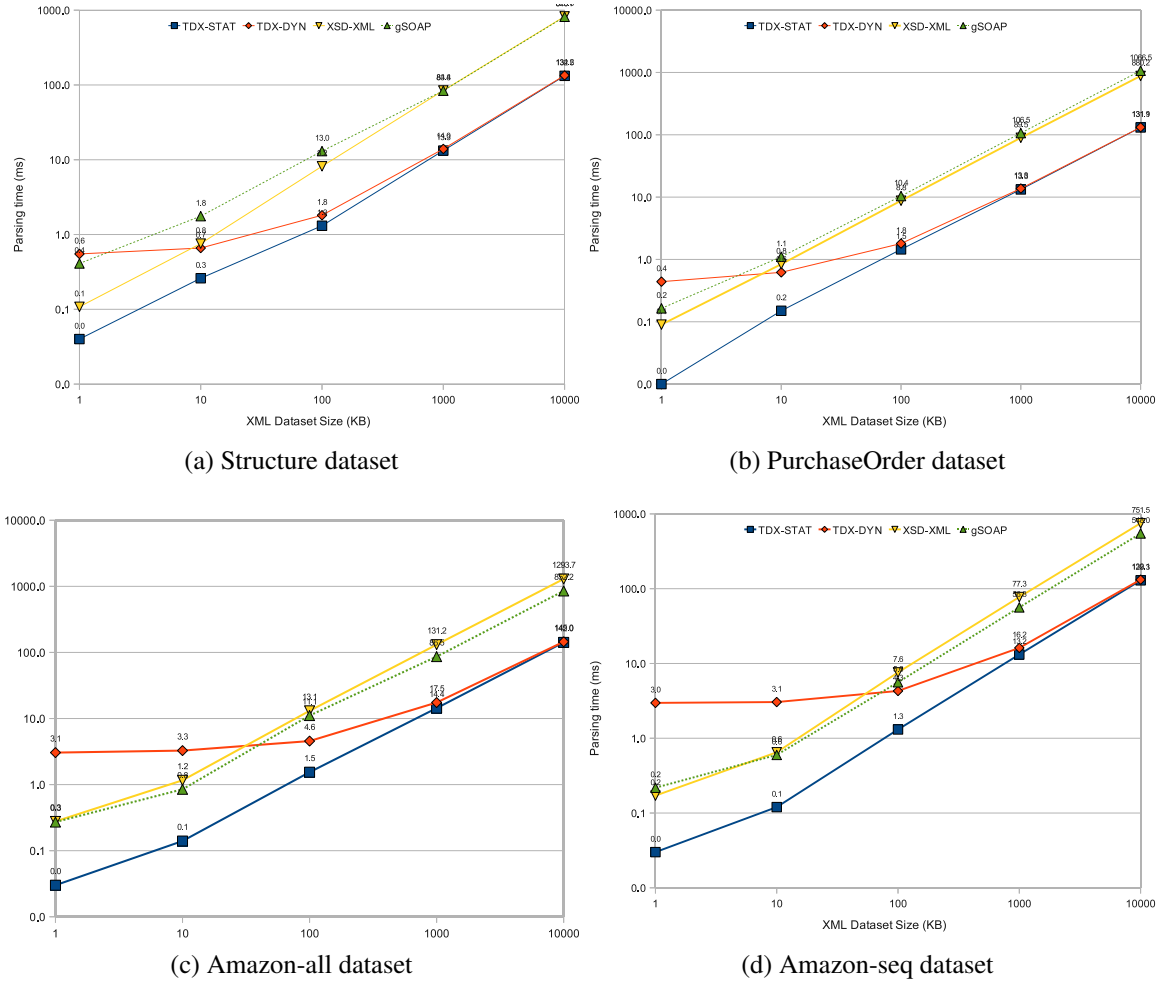
(a) Small dataset size (1KB)

(b) Medium dataset size (100KB)



(c) Large dataset size (10MB)

Figure 7.3: Throughputs comparison to schema-specific validating parsers over different XML dataset Sizes.



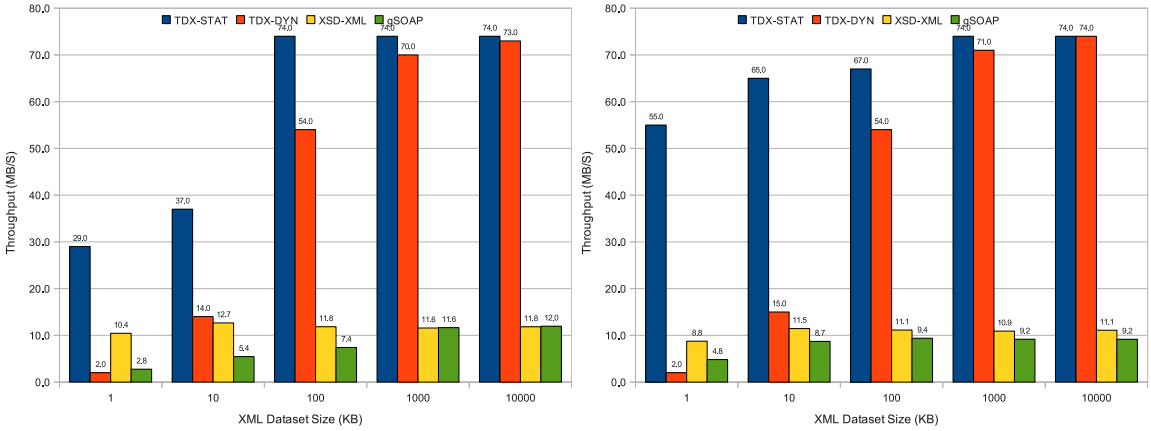
(a) Structure dataset

(b) PurchaseOrder dataset

(c) Amazon-all dataset

(d) Amazon-seq dataset

Figure 7.4: Scalability of schema-specific validating parsers over different XML datasets in terms of parsing and validation time.



(a) Structure dataset

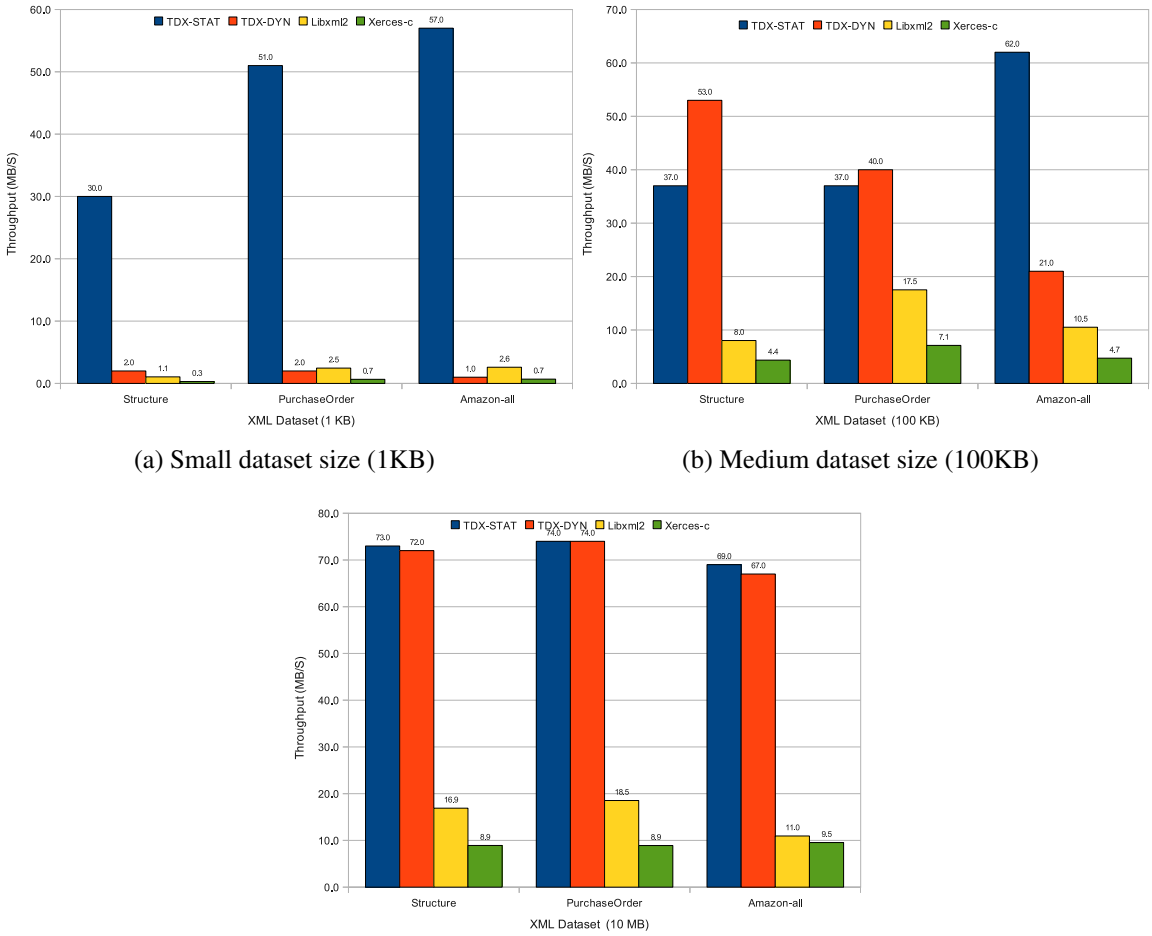
(b) PurchaseOrder dataset

(c) Amazon-all dataset

(d) Amazon-seq dataset

Figure 7.5: Scalability to schema-specific validating parsers over different XML datasets in terms of throughputs.

7.2.3 Compared to Non-Schema-Specific Validating Parsers



(a) Small dataset size (1KB)

(b) Medium dataset size (100KB)

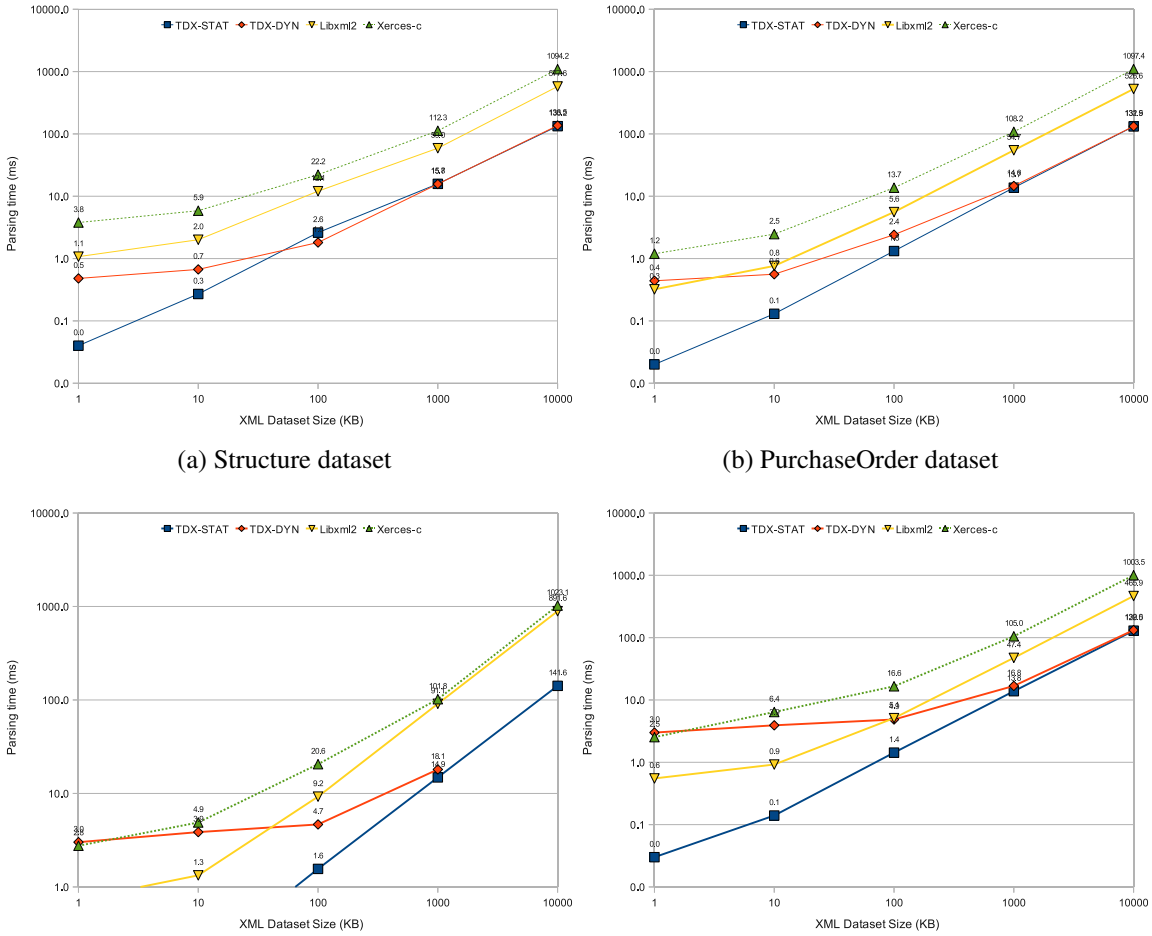
(c) Large dataset size (10MB)

Figure 7.6: Throughputs comparison to non-schema-specific validating parsers over different XML datasets with small, medium and large sizes.

Throughput Performance. Unlike schema-specific parsers that generally construct parsers specialized to a set of schemas, generate parsers, compile the source codes at compile-time, and eliminates the access to the schemas at runtime, non-schema-specific validating parsers typically require access to the schemas to accomplish the validation. As a result, such validating parsers usually exhibit poor performance compared to the schema-specific parsers. The major advantages of such parsers is that they do not require compilation because they read and construct the schema on-the-fly compared to traditional schema-specific peers. However, this limitation does not apply to our dynamic mode TDX (TDX-DYN). As we have described, TDX-DYN combines the advantages of schema-specific parsers and runtime validating parser. Our Table-Driven approach achieves better performance while addresses schema changes in an elegant manner, i.e., populating the parsing table

and dynamically loading the semantic libraries. Figure 7.6 shows the throughputs comparison of TDX to the examined non-schema-specific parsers, XSD parser and gSOAP.

For small XML messages (1 KB), TDX-STAT cab be up to 80x to 100x faster than xerces-c (SAX) for pro parsing and validating `Structure.xml` of size 1 KB (Figure 7.6a. Although TDX-DYN incurs one-time initialization overheads, it is still between 2x and 7x faster than xerces-c (SAX). TDX-STAT is between 10x and 17x faster than xerces-c (SAX), and between 4x and 9x faster than LibXML2 on medium size messages (Figure 7.6b). Figure 7.6c also shows TDX-STAT is 7x to 8x faster than xerces-c (SAX), and is between 4x and 6x faster than LibXML2 on XML instances of size 10 MB. TDX-DYN has the same performance to large messages.



(a) Structure dataset

(b) PurchaseOrder dataset

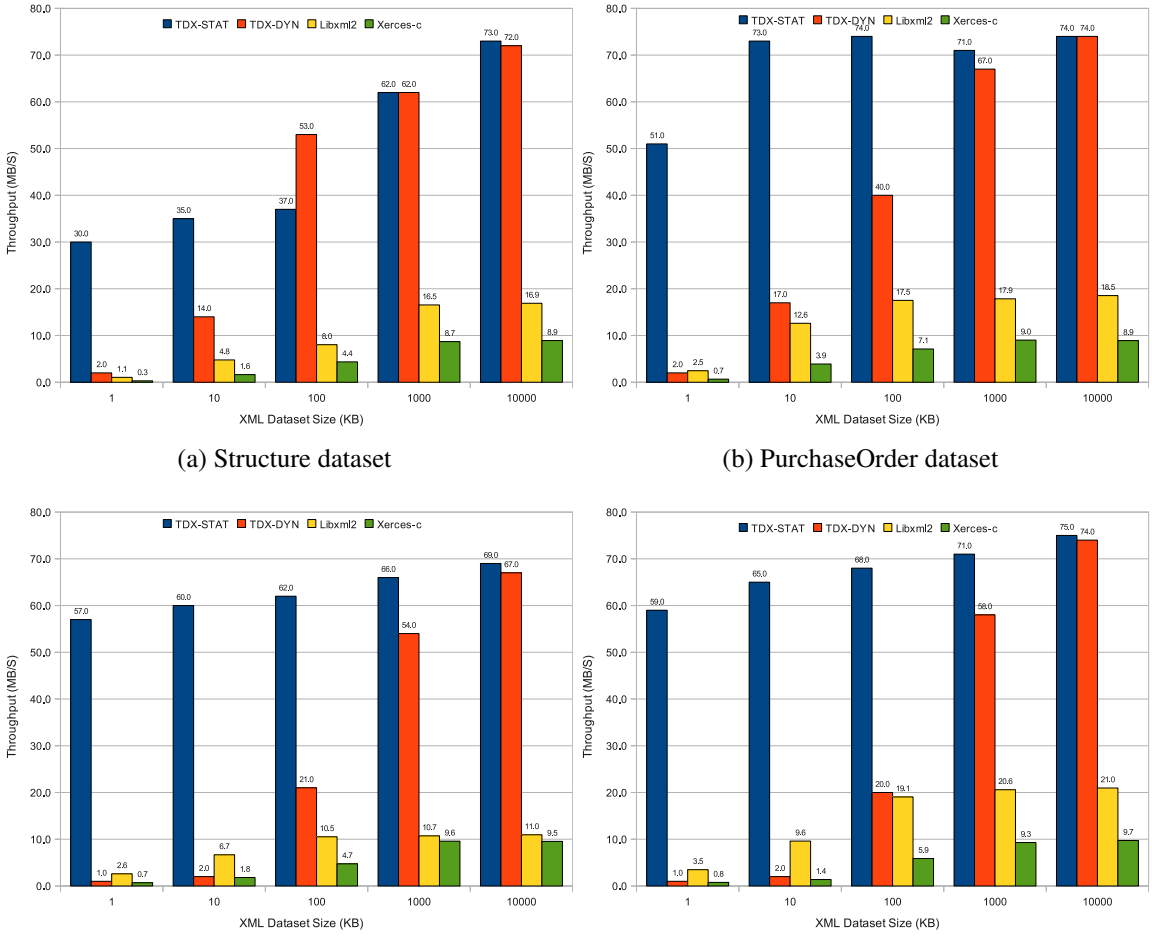
(c) Amazon-all dataset

(d) Amazon-seq dataset

Figure 7.7: Scalability to schema-specific validating parsers over different XML datasets in terms of parsing and validation time.

Scalability. Scalability comparison with non-schema-specific parsers in terms of parsing and validation time is shown in Figure 7.7. The data for both TDX-STAT and TDX-DYn are the same as

in Figure 7.4, thus sharing the same scalability properties. Both LibxML2 and xerces-c scale well while they bear small overheads on messages in small sizes. The alternative scalability comparison in terms of throughputs is shown in 7.8.



(a) Structure dataset

(b) PurchaseOrder dataset

(c) Amazon-all dataset

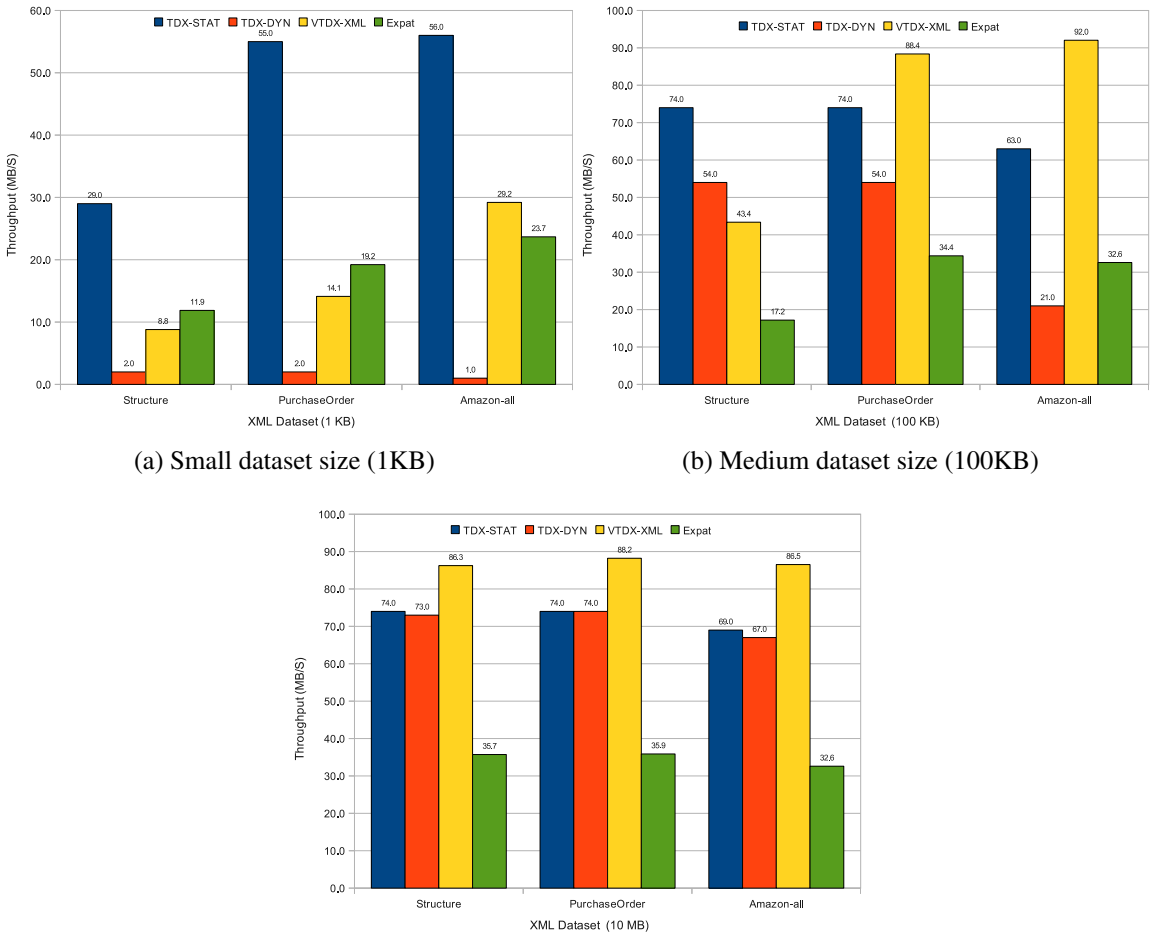
(d) Amazon-seq dataset

Figure 7.8: Scalability to non-schema-specific validating parsers over different XML datasets in terms of throughputs.

7.2.4 Compared to Non-validating Parsers

Non-validating parsers refer to the parsers that perform well-formedness checking of an XML document without performing data types checking or verifying structures of the document against a schema. It is well known that validation incurs significant processing overhead. As a result, non-validating parsers are typically much faster than validating parsers. We compared our TDX, a

validating parser[1], with two widely-used parsers, expat and VTD-XML. The experimental results are shown in Figure 7.9.



(a) Small dataset size (1KB)

(b) Medium dataset size (100KB)

(c) Large dataset size (10MB)

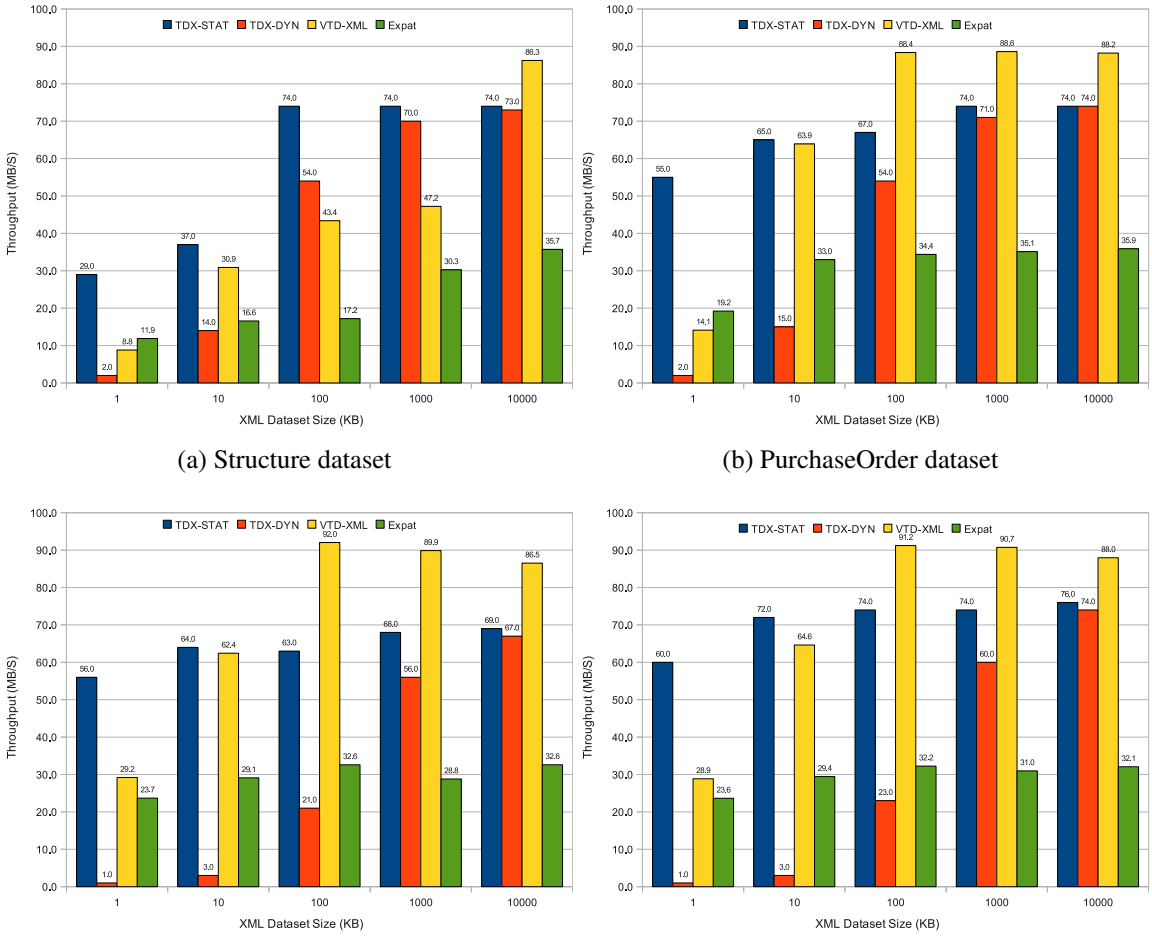
Figure 7.9: Throughputs comparison to non-validating parsers over different XML datasets in small, medium and large sizes.

The results show that our validating TDX-STAT is even 2x to 4x faster than non-validating parser, expat, in all test cases. Specifically, TDX-STAT is 2x faster than expat over processing `Structure.xml` and `Amazon-all.xml`, and approximately 3x faster than expatover processing `PurchaseOrder.xml`, of size 1 KB. On medium size XML messages (100 KB), TDX-STAT is 4x, 2x and 2x faster than expat over XML messages `Structure.xml`, `PurchaseOrder.xml`, and `Amazon-all.xml`, respectively. It is approximately 2.x faster than VTD-XML parser over `Structure.xml`, yet VTD-XML parser is 1x and 1.5x faster than TDX-STAT over `PurchaseOrder.xml` and `Amazon-all.xml`, respectively. On larger size XML messages (10 MB), TDX-STAT is

[1]Parsing and validation are integrated in a single stage in TDX

approximately 2x faster than expat in all the three different XML messages, and is approximately 20% slower than VTD-XML parser. Note that TDX is a validating parser and VTD-XML is a non-validating parser. As we have described in Section 7.2.1, when TDX performs parsing and structural validation by disabling the type-checking validation, TDX-STAT is faster than VTD-XML in all the test cases we conducted in our experiments. This demonstrates advantages of our table-driven approach for XML parsing.

TDX-DYN is slower for XML messages in small size due to the initialization overhead. As the size of the XML message increases, the proportion of initialization overhead will drop down. Therefore, its performance will be approaching the same as TDX-STAT. Figure 7.10 illustrates the trend of STD-DYN in terms of throughputs.



(a) Structure dataset

(b) PurchaseOrder dataset

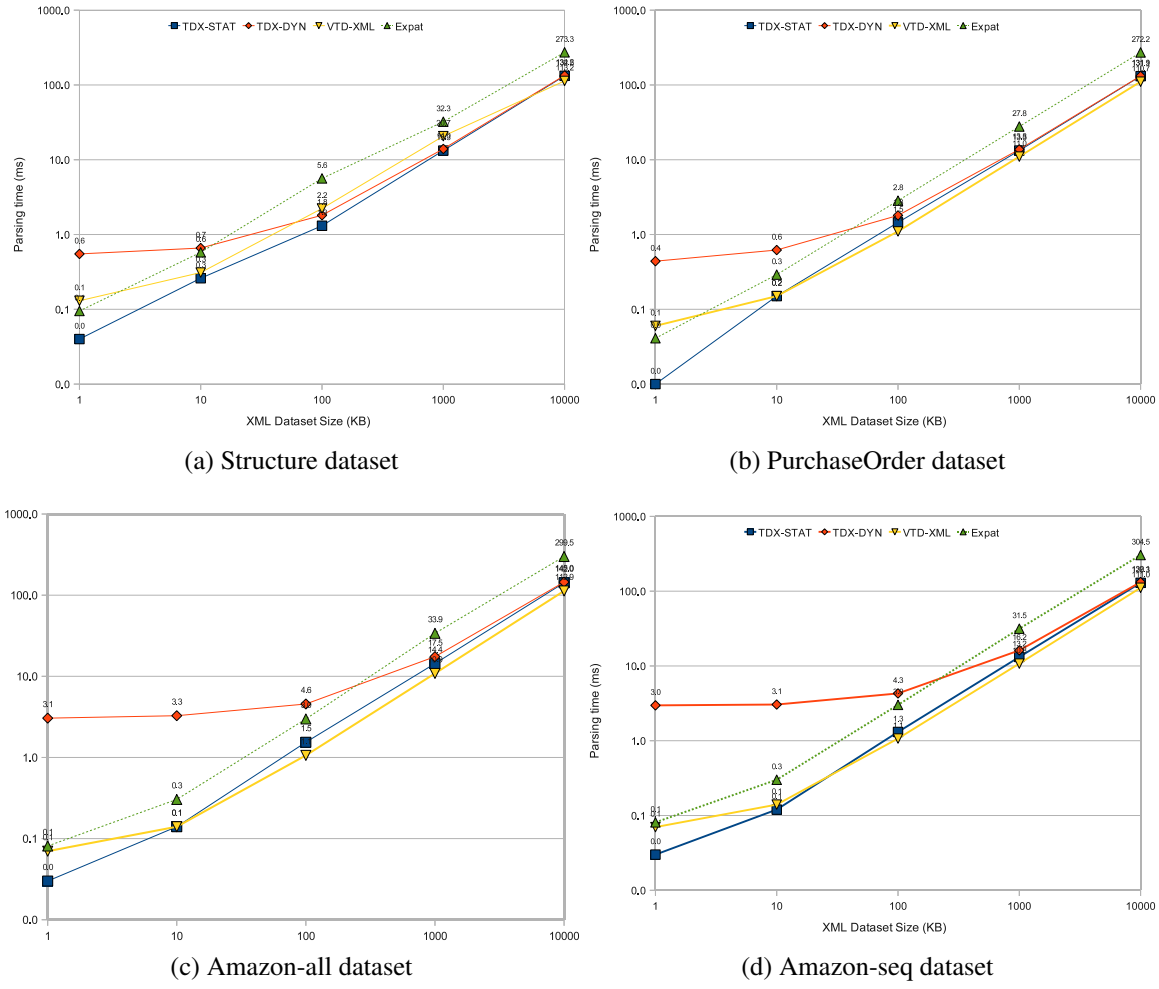
(c) Amazon-all dataset

(d) Amazon-seq dataset

Figure 7.10: Throughputs comparison to non-validating parsers over different XML datasets.

Scalability. Scalability of non-validating parsers compared with our validating TDX parser is shown in Figure 7.11. The non-validating expat parser scales well in all the test cases from 1 KB

to 10^4 KB. This also indicates expat has very small initialization overhead. VTD-XML scales well too starting from 10 KB in all the test cases conducted in our experiments. From the charts, we can see VTD-XML encounters a some initialization overhead. The figure also show that VTD-XML runs faster than expat for parsing messages of size over 10 KB. It is also faster than our validating TDX-STAT parser except parsing messages of small size. This is because TDX-STAT does not incur any initialization overhead.



(a) Structure dataset

(b) PurchaseOrder dataset

(c) Amazon-all dataset

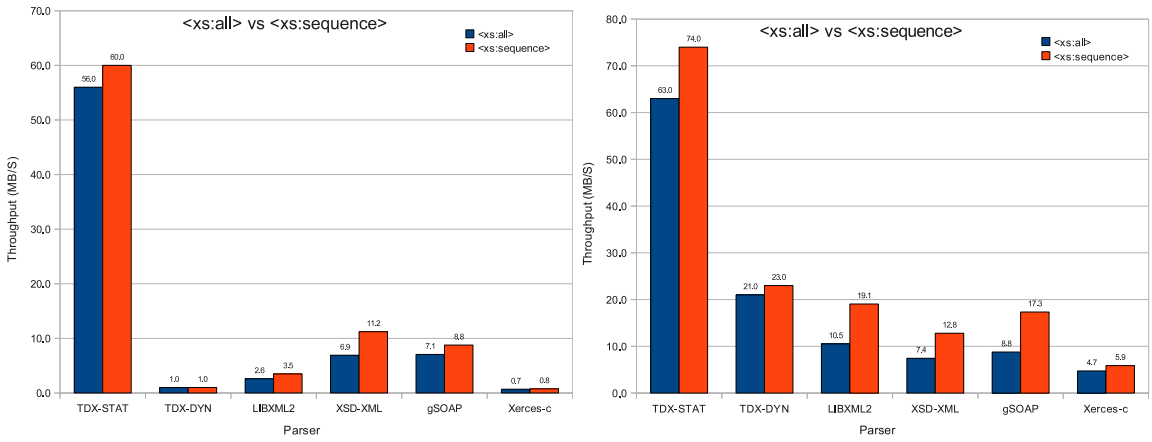
(d) Amazon-seq dataset

Figure 7.11: Parsing and validating time comparison to non-validating parsers over different XML datasets.

TDX-DYN, however, requires constructing of parsing table and dynamically loading type-checking libraries on-the-fly. This introduces one-time initialization cost. As it is shown in the figure, TDX-DYN does not present well-scalability, in particular when the documents are small in size. The starting size may vary depending over the structure of the document. For example, TDX-DYN starts to scale well from 10^2 KB for parsing `Structure.xml` and `PurchaseOrder.xml`. But

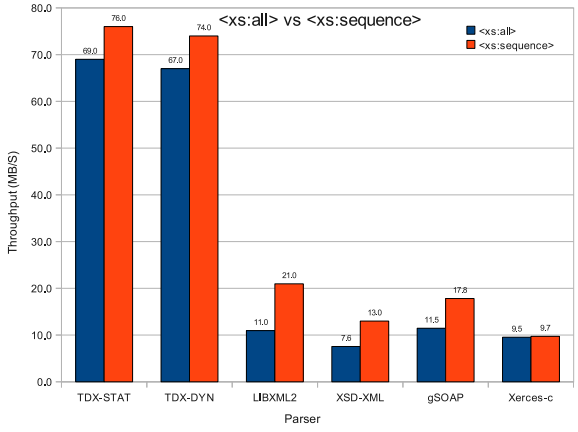
it starts to scale from 10^3 KB for parsing `Amazon-all.xml` and `Amazon-seq.xml`. Because the only difference between TDX-STAT from TDX-DYN is that TDX-DYN incurs initialization overhead, we can conclude TDX-DYN also starts to scale well as the size of XML message increases. The Figure 7.10 also illustrates the same scalability in terms of throughputs.

7.2.5 Performance impact of Validating Unordered XML Elements and Attributes



(a) Small dataset size (1KB)

(b) Medium dataset size (100KB)

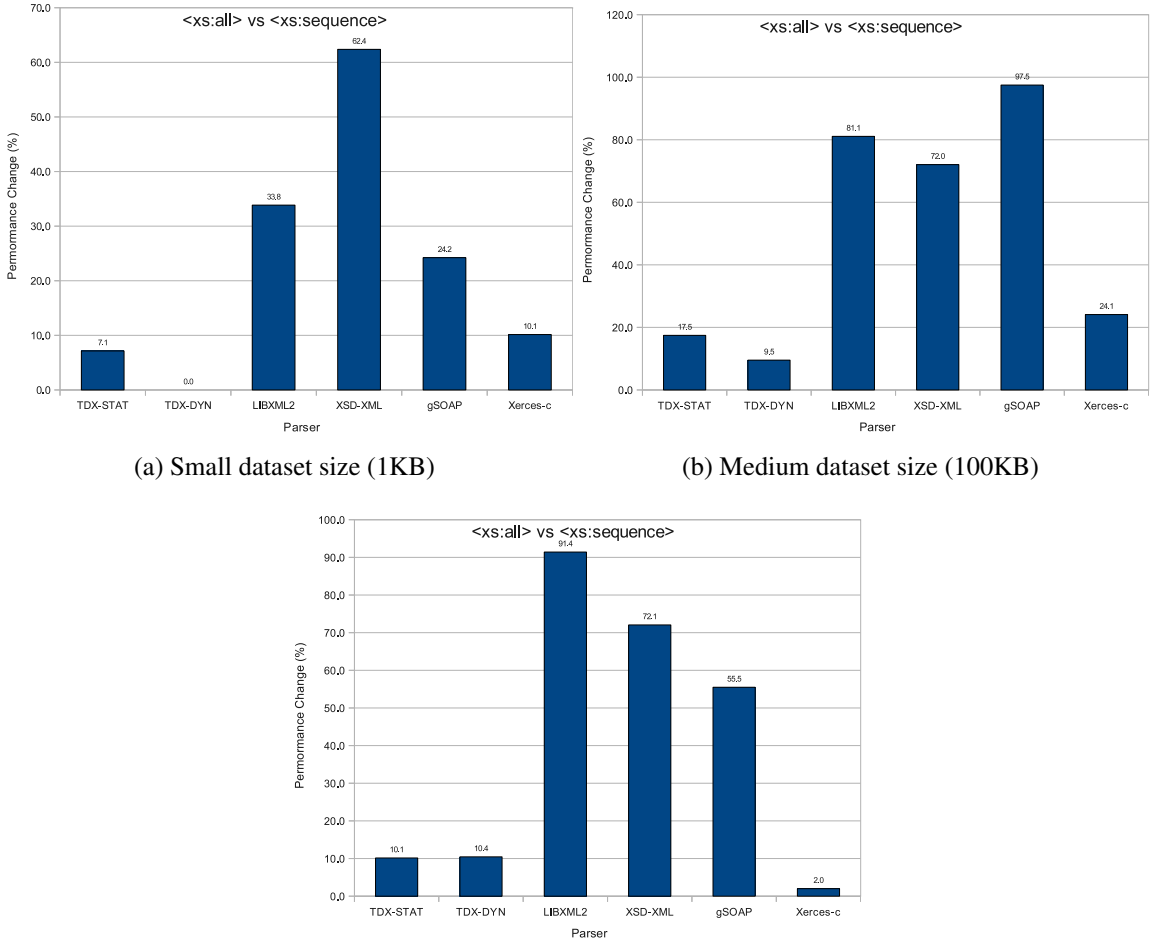


(c) Large dataset size (10MB)

Figure 7.12: Throughputs comparison on validating unordered XML elements against sequenced XML elements on different XML dataset in various sizes (<xs:all> vs. <xs:sequence>).

XML schema `xsd:all` and `xsd:attribute` groups pose challenges for validating unordered elements and attributes. We developed *permutation phrase* grammars that represent the

unordered elements and attributes in an elegant manner (see Chapter 4 for details) and algorithms that efficiently validate such grammars using a two-stack PDA (see Chapter 5 for details).



(a) Small dataset size (1KB)

(b) Medium dataset size (100KB)

(c) Large dataset size (10MB)

Figure 7.13: Percentage of performance dropped on validating unordered XML elements compared to validating sequenced elements of different XML dataset sizes (<xs:all> vs. <xs:sequence>).

To determine the impact of validating unordered elements and attributes, we conducted experiments by creating two schemas `Amazon-all.xsd` and `Amazon-seq.xsd` with `xsd:all` and `xsd:sequence` containing the same number of elements with same data types respectively (each schema contains 50 elements), i.e., they only differ from the schema group models `xsd:all` and `xsd:sequence`. To further break down the impact of the number of elements in the `xsd:all` groups, we created a set of schemas with different numbers (4, 8, 16, 32) of elements based the two schemas. The experimental results are shown in Figure 7.12 - 7.15.

Figure 7.12 illustrates the throughputs comparison of validating parsers over validating unordered elements `xsd:all` against ordered elements `xsd:sequence`. Figure 7.13 shows the percentage of performance increased of those validating parser from validating the unordered elements to validating the ordered elements in the same test cases. From both figures, we can see that TDX achieves best performance for validating unordered elements. The second best is xerces-c (SAX). Other validating parser, XSD-XML, gSOAp and LibXML2, performs worst for document in small, medium and large sizes, respectively.On average, TDX-STAT and TDX-DYN increase 11.6% and 9.9% respectively. Xerces-C (SAX) increases 12.1%. gSOAP increases 59.4%, LibXML2 increases 68.7% and XSD-XML increases 75.5% respectively. This indicates that the last three validating parser are not optimized to perform unordered elements and attributes validation. Both TDX and xerces-c do not incur dramatically performance drops when validating unordered elements and attributes imposed by XML schema groups `xsd:all` and `xsd:attributes`.

Specifically, Figure 7.12a and 7.13a illustrate the performance comparison over validation document in small size (1 KB). The throughput of TDX-STAT increases from 56.0 to 60.0 MB/S, i.e., 7.1% increase. TDX-DYN achieves the same throughput for both ordered and unordered elements in this test case. XSD-XML is the worst among those tested validating parsers for validating unordered elements when the document is small in size. Its throughput increases from 6.9 to 11.2 MB/S, 62.4% increase. Xerces-c (SAX) is more suitable for validation unordered elements than gSOAP and LibXML2. Xerces-c (SAX) increases 10.1% from 0.7 to 0.8 MB/S. gSOAP increases 24.2% MB/S from 7.1 to 8.8 MB/S while LibXML2 increase 33.8% from 2.6 to 3.5 MB/S.

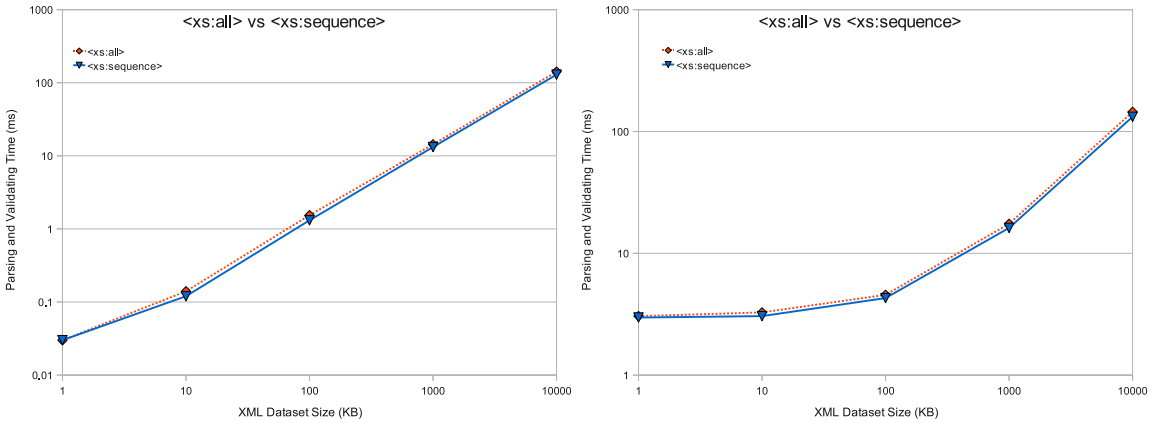
For validating document in medium size (Figure 7.12b and 7.13b), TDX-STAT increased from 63.0 to 74.0 MB/S, i.e., 17.5% increase. TDX-DYN increases 9.5% from 21.0 to 23.0 MB/S. Xerces-c (SAX) increases 24.1% from 4.7 to 5.9 MB/S. XSD-XML increases 72.0% from 7.4 to 12.8 MB/S while LibXML2 increase 81.1% from 10.5 to 19.1 MB/S. gSOAP increases from 97.7% increase from 8.8 to 17.3 MB/S.

For validating document in large size (Figure 7.12c and 7.13c), TDX-STAT increased 10.1% increase from 69.0 to 76.0 MB/S. TDX-DYN increases 10.4% from 67.0 to 74.0 MB/S. Xerces-c (SAX) increases 2.0% from 9.5 to 9.7 MB/S. gSOAP increases from 55.5% increase from 8.8 to 17.3 MB/S. XSD-XML increases 72.1% from 7.4 to 12.8 MB/S while LibXML2 increase 91.4% from 10.5 to 19.1 MB/S.

Scalability Impact of Validating Unordered Elements. Figure 7.14 illustrates the parsing and validation time on unordered XML elements compared to ordered elements over different XML dataset sizes (<xsd:all> vs. <xsd:sequence>). It clearly shows that both TDX-STAT and TDX-DYN introduce very little overhead for parsing and validating instance of `xsd:all`. The difference of parsing and validation time between processing unordered elements (`xsd:all`) and ordered elements (`xsd:sequence`) is very small. It also indicates that TDX-STAT scales linearly when the size of the dataset increases (Figure 7.14a). Due to the initialization overhead, TDX-DYN does not scale well for small dataset in size (Figure 7.14b).

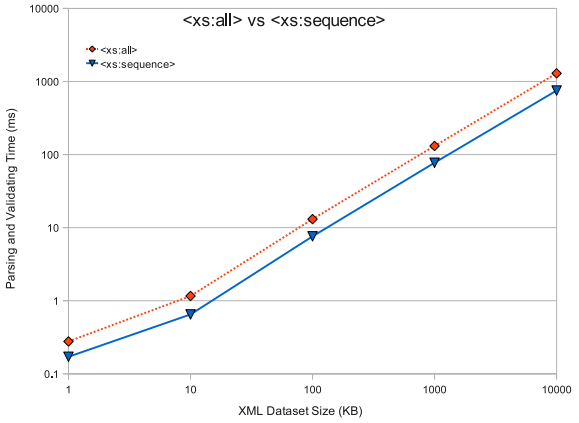
(Figure 7.14f indicates Xerces-c (SAX) does not present linear scalability. In addition,parsing and validation time on unordered and ordered elements may interleave depending on the dataset size. It is also clear that the difference between ordered and unordered elements paring and validation is very small.

XSD-XML exhibits linear-scalability property (Figure 7.14c). However, the large difference between parsing and validating unordered and ordered elements indicates its performance will drop dramatically for XML instances of XML schema `xsd:all` group.

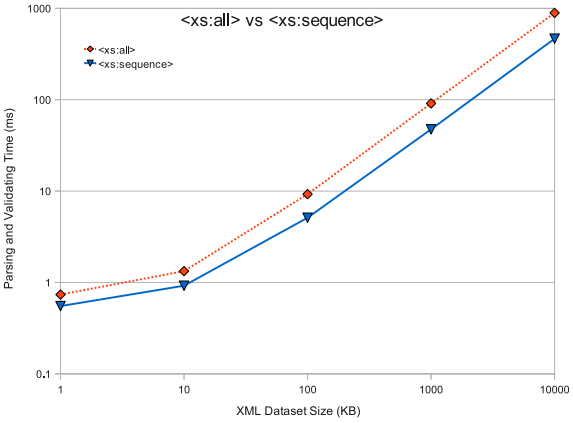


(a) TDX-STAT Parser

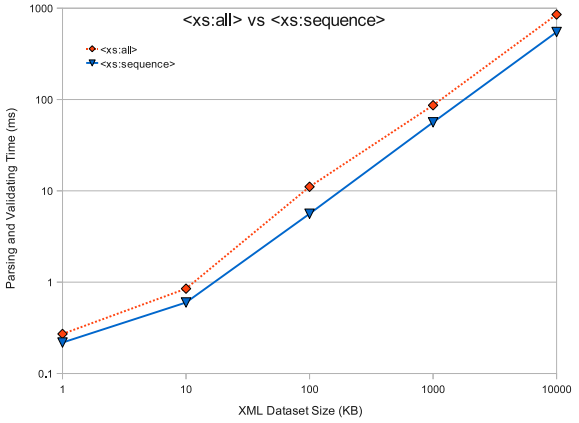
(b) TDX-DYN Parser



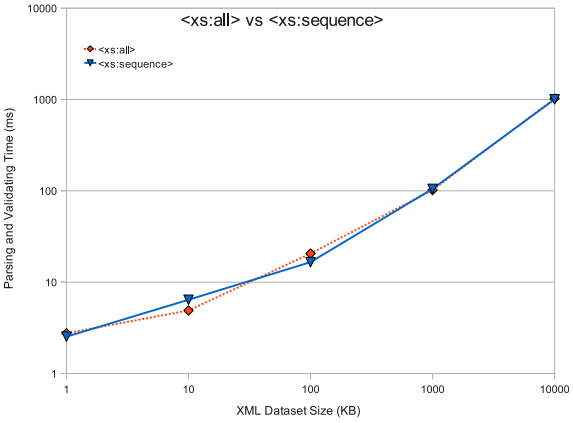
(c) XSD-XML Parser



(d) LibXML2 Parser



(e) gSOAP Parser



(f) Xerces-C Parser

Figure 7.14: Parsing and validation time on parsing and validating unordered XML elements compared to ordered elements over different XML dataset sizes (<xsd:all> vs. <xsd:sequence>).

Both LibXML2 (Figure 7.14d) and LibXML2 (Figure 7.14e) exhibit initialization overheads and scale well except with datasets in very small sizes. Like XSD-XML parser, both parsers are not suitable for parsing and validating XML instances of XML schema `xsd:all` group.

7.2.6 Scalability Impact of Number of Unordered Elements for TDX

Figure 7.15 illustrates the parsing and validation time of our TDX for processing the XML datasets with various different numbers of unordered elements in various sizes. The results demonstrate that the number of `xsd:all` elements (also applicable to `xsd:attribute`) does not incur a significant performance penalty for our TDX. The number of the elements in `xsd:all` makes not much difference to throughputs. As the results show, the parsing and validation time for each of specific dataset remains almost constant although the number of elements in `xsd:all` varies from 4 to 32.

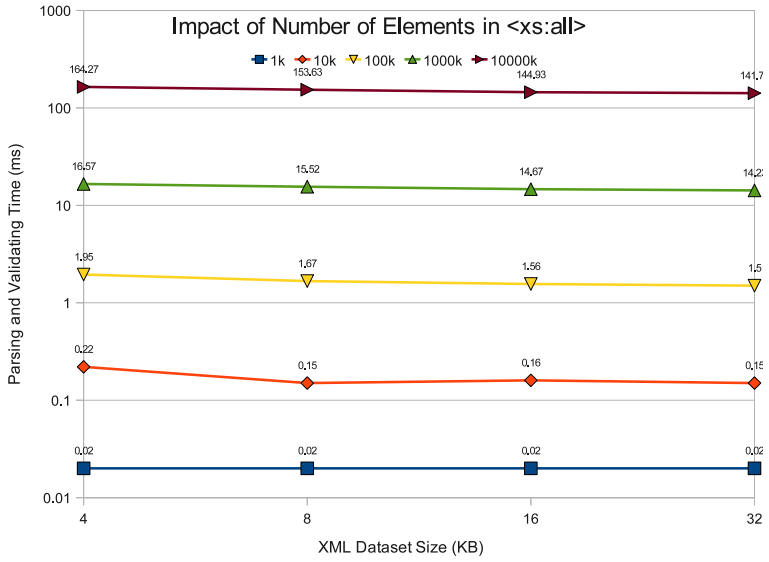
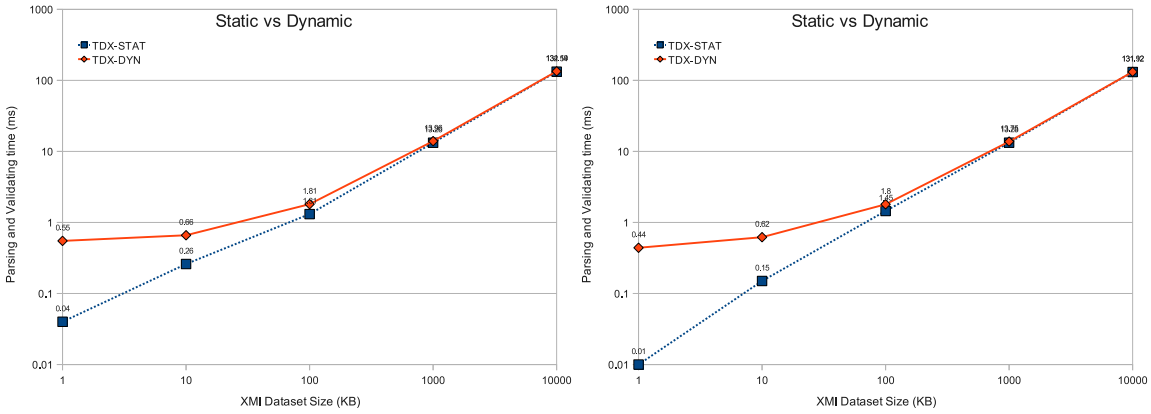


Figure 7.15: Parsing and validation time comparison against the number of unordered elements in <xsd:all>.

7.2.7 Overhead Incurred by Dynamic TDX

Our table-driven approach supports two modes, *static* (TDX-STAT) and *dynamic* (TDX-DYN), for generating a TDX parser. TDX-STAT maximizes the performance by generating the parsing table and data-type checking functions in C/C++ codes. As a result, constructing the table and loading the functions at runtime are eliminated. However, TDX-STAT requires recompilation and redeployment when the schemas from which the parser is generated change or update. On the contrary, TDX-DYN constructs the table and loads the data-type checking libraries on-the-fly. The parsing engine of TDX only depends on the paring table and data-type validation functions. As a result, when the schemas change or update, the parsing table and data checking libraries are regenerated at compile

time manually or using tools and populated to the parser. The parser reconstructs the parsing engine and loads those libraries at on-the-fly. This efficiently addresses the recompilation and redeployment requires TDX-STAT and traditional schema-specific parsers encounter. However, dynamically constructs the parsing table and loads the validation libraries incur some overheads. It is worthy to note that such initialization overheads are only required once. TDX-DYN parses and validates subsequent messages (instances of the schemas) as same as TDX-STAT parser. Figure 7.16 shows the parsing and validation time of static TDX (TDX-STAT) compared to dynamic TDX (TDX-DYN) over different datasets of various sizes.



(a) Structure dataset

(b) PurchaseOrder dataset

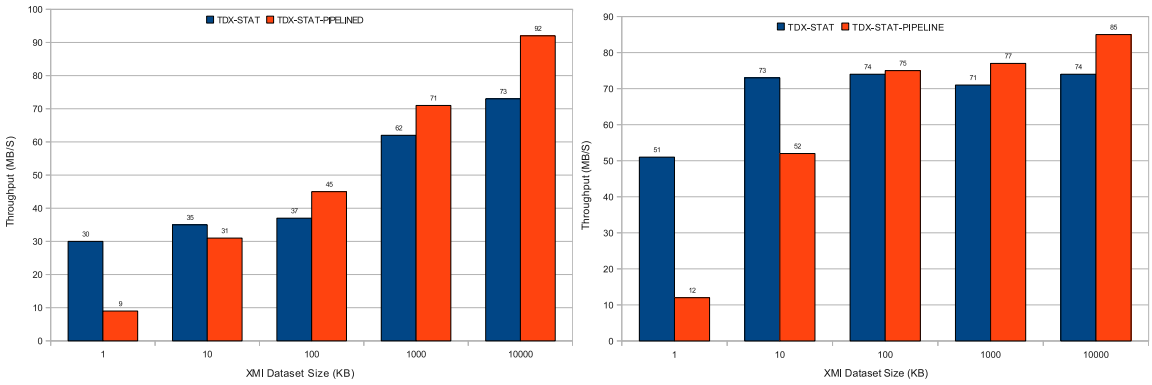
(c) Amazon-all dataset

(d) Amazon-seq dataset

Figure 7.16: Static TDX vs. dynamic TDX over different XML datasets in terms of parsing and validation time.

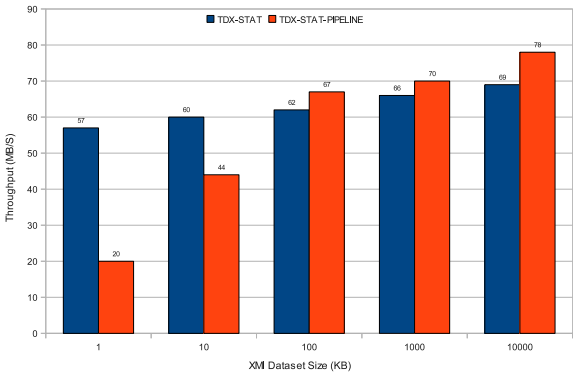
The initialization overheads of TDX-DYN are dominant for XML datasets in small size. As this figure shows, the parsing and validation time for dataset `Structure.xml` of size 1 KB of TDX-DYN is $0.55ms$ while it is only $0.04ms$ in static TDX. The parsing and validating time of TDX-DYN and its corresponding time of TDX-STAT are $0.44ms$ and $0.01ms$, $3.05ms$ and $0.03ms$, and $2.97ms$

and $0.03ms$ for datasets `PurchaseOrder.xml`, `Amazon-all.xml`, and `Amazon-seq.xml`, all of size 1 KB, respectively. Because the initialization overheads are caused by dynamically constructs the parsing table and loads the validation libraries, such costs are determined by the schemas not the XML instances of the schemas (the size of the parsing table and the number of validation functions), the impact of such initialization costs will be dropped for large dataset in size. As the figure illustrates, the performance of TDX-DYN only drops 5.3% compared to TDX-STAT for dataset `Structure.xml` of size 1 MB, 3.5% for dataset `PurchaseOrder.xml` of size 1 MB, 2.2% for dataset `Amazon-all.xml` of size 1 MB, 2.3% for dataset `Amazon-seq.xml` of size 1 MB, respectively.

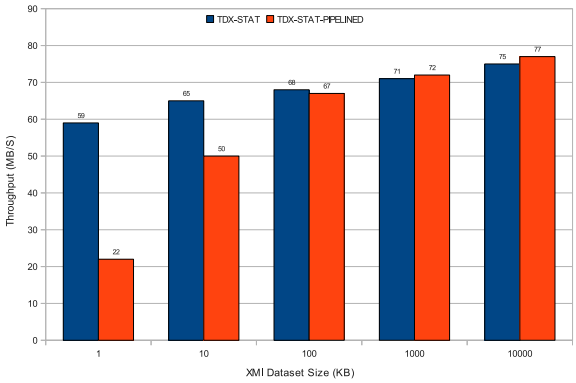


(a) Structure dataset

(b) PurchaseOrder dataset



(c) Amazon-all dataset



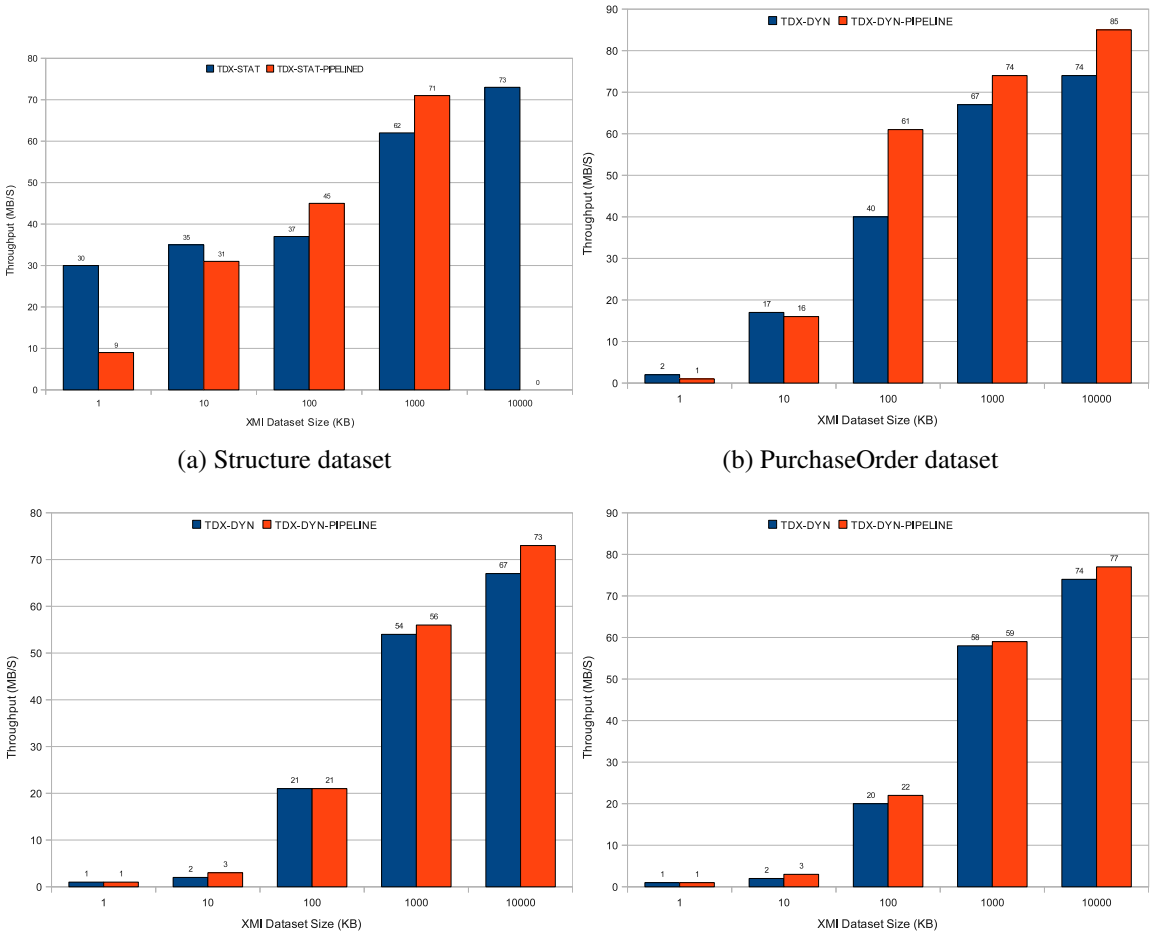
(d) Amazon-seq dataset

Figure 7.17: Throughputs comparison of TDX-STAT by pipelining over different XML datasets of various sizes.

As aforementioned, initialization cost is determined by the schemas, not the XML instances of the schemas. Specifically, the parsing table size, the number and complexity of grammar productions and validation functions determine the initialization costs. In out TDX, unordered elements and attributes are presented as *permutation phrase* . To efficiently parse and validate such grammar productions, a mapping or a hash table is constructed. As a result, TDX-DYN introduces such

initialization cost for `xsd:all` and `xsd:attribute` that TDX-STAT does not. Figure 7.16c and Figure 7.16d illustrate the time for parsing and validation time for two parsers generated from schemas `Amazon-all.xsd` and `Amazon-seq.xsd` respectively. All the elements and attributes are of the same type except that the former is grouped in `xsd:all` and the latter is grouped in `xsd:sequence`. The results shown that TDX-DYN introduces $0.08ms$ (from $2.97ms$ to $3.05ms$) for the datasets of size 1 KB.

7.2.8 Performance Improvements by Pipelining



(a) Structure dataset

(b) PurchaseOrder dataset

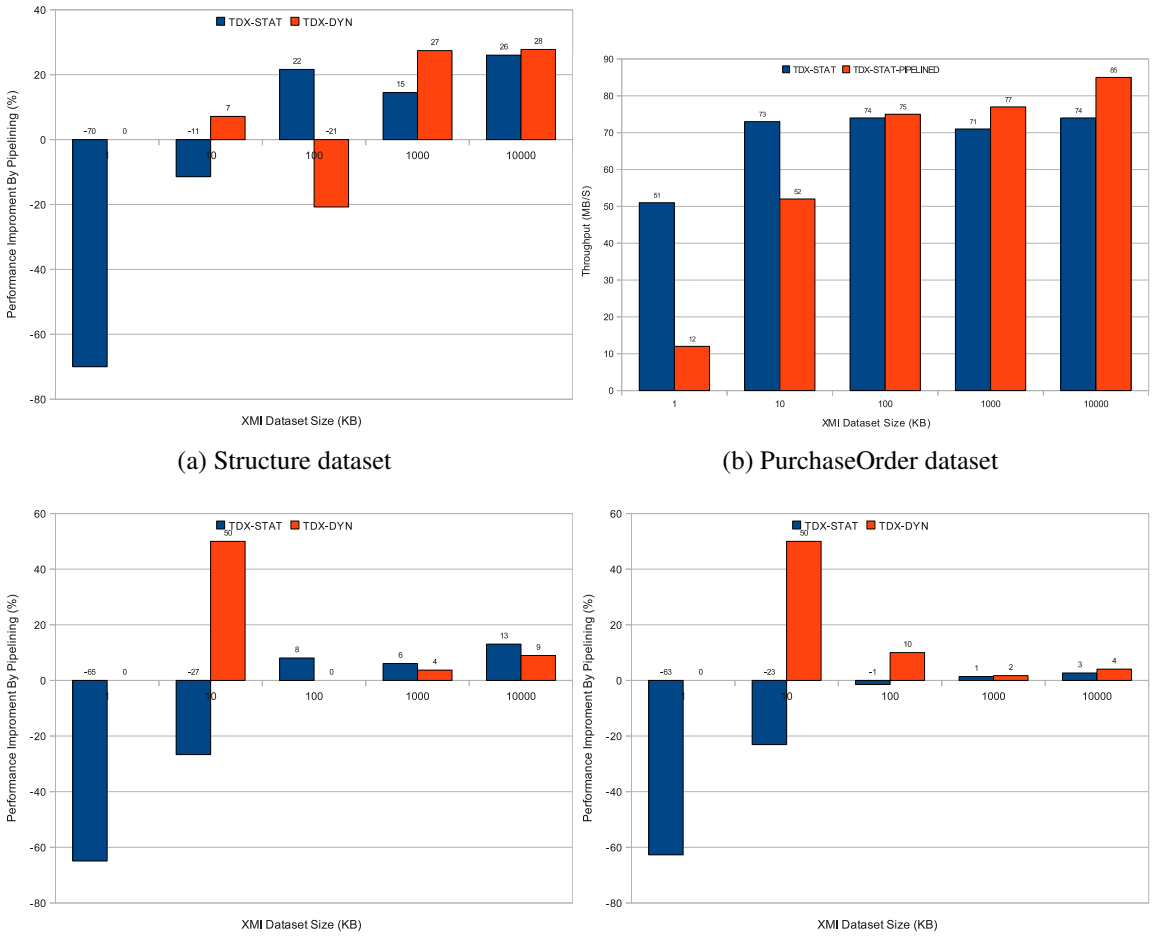
(c) Amazon-all dataset

(d) Amazon-seq dataset

Figure 7.18: Throughputs comparison of TDX-DYN by pipelining over different XML datasets of various sizes.

To determine the speedups improved by pipelining the scanning, tokenization, as well as parsing and validation of our TDX-STAT and TDX-DYN, I conducted the experiments to compare the performance improvements by pipelining. Figure 7.17 and Figure 7.18 illustrate the throughputs

comparison of pipelined TDX-STAT and pipelined TDX-DYN, compared to non-pipelined TDX-STAT and TDX-DYN respectively, over different datasets of various sizes. The results indicate that both TDX-STAT and TDX-DYN can gain performance improvements by pipelining for large XMl datasets in size. TDX-STAT can improve throughput up to 19 MB/S from 73 MB/S to 92 MB/S and TDX-DYN can gain up to 20 MB/S from 72 MB/S to 92 MB/S, on the `Structure.xml` of size 10 MB, on the `Structure.xml` of size 10 MB. TDX-STAT can improve throughputs up to 11 MB/S (74 to 85), 9 MB/S (69 to 78), 2 MB/S (75 to 77) on datasets `PurchaseOrder.xml`, `Amazon-all.xml` and `Amazon-seq.xml` of size 10 MB, respectively. TDX-DYN can improve the throughputs up to 11 MB/S (74 to 85), 7 MB/S (67 to 63), 3 MB/S (74 to 77) on datasets of the same size.



(a) Structure dataset

(b) PurchaseOrder dataset

(c) Amazon-all dataset

(d) Amazon-all dataset

Figure 7.19: Speedups by pipelining.

However, on small datasets in size, pipelining does not not improve the throughputs, on the contrary, it reduces the performance, in particular to TDX-STAT. Throughput may drop up to 21 MB/S of TDX-STAT on the dataset `Structure.xml` of size 1 KB. This is caused by context

switch overheads and communication synchronization overheads among the circular buffers. The smaller the dataset to be processed, the larger impact of such overheads apply. This suggests that pipelined TDX should be used only for processing large datasets. As our table-driven approach was designed for XML stream, which is typically a large XML streaming, performance improvements cal be expected by pipelining.

Figure 7.19 shows the speedups measured for the different datasets of various sizes. The performance may drop up to 70% for small datasets in size, and can achieve up to 28% of *TDX-STAT* for large dataset in size. We can see the trend that the speedup increases as the size increases except the largest speedup gained of *TDX-DYN* appears at medium size.

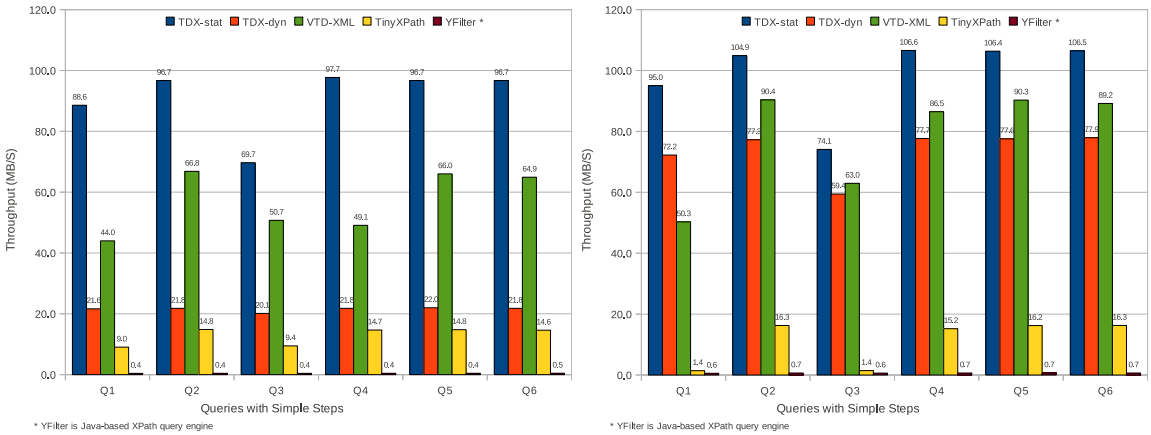
7.3 Performance of TDX Query Processor

This section describes performance of our TDXPath query processor compared with other query processors including *VTD-XML*, *TinyXML*, and *YFilter*. We conducted experiments to characterize its features and performance, gain insights of performance impacts of various XPath features, and provide an exploratory description of the features and performance of systems that are related to TDXPath. We begin in Section 7.3.1 system performance on evaluating queries consisting of only simple step locations without predicates, wildcards, backward axes (also called reverse axes of `axis` and `ancestors`). We then investigate the characteristics on evaluating wildcards queries in Section 7.3.2. Performance on evaluating backward axes queries is examined in Section 7.3.3. Section 7.3.4 describes the characteristics of performance on evaluating various queries that match the same results. We then examine the performance on evaluating different numbers of queries in Section 7.3.5. In Section 7.3.6, the system scalability to XML dataset sizes is exhibited.

7.3.1 Performance of Queries with Simple Step Queries

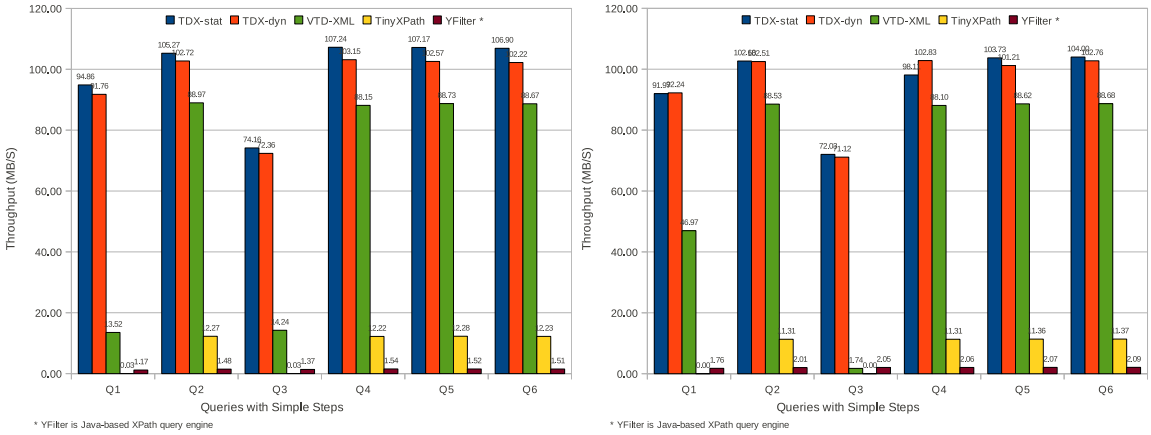
Performance comparison on simple step queries is shown in Figure 7.20. These simple step queries select nodes, attributes and text contents of nodes without predicates, backward axes, or children axes more than one distance. Because such simple step queries are the basic of features that all the XPath query engines support, the performance of simple step queries can be used to evaluate general performance of query methods implemented in different query engines.

Figure 7.20 shows the performance comparison on simple step queries over the `PurchaseOrder` datasets of different sizes, small ($10KB$), medium ($100KB$), large ($1MB$) and very large ($10MB$). The results show that our validating TDXPath query processor outperforms all other query processors measured on all the queries over all the different dataset sizes. The throughputs of TDXPath (static mode) range from $88.6MB/S$ to $107.2MB/S$. TDXPath (static mode) is around 1.5 to 2 times faster than the second fasted query processor *VTD-XML*, 6.6 to 67.9 times faster than *TinyXML*, and 50 to 220 times faster than *YFilter*.



(a) Small size dataset (10 KB)

(b) Medium size dataset (100 KB)



(c) Large size dataset (1 MB)

(d) Very large size dataset (10 MB)

Q1: /purchaseOrder/items/item/productName/text()
Q2: /purchaseOrder/billTo/@country
Q3: /purchaseOrder/items/item/@partNum
Q4: /purchaseOrder/shipTo/zip
Q5: /purchaseOrder/billTo/name
Q6: /purchaseOrder/billTo/name/text()

(e) Queries

Figure 7.20: Throughputs for different queries with simple step queries over PurchaseOrder datasets in various sizes.

It is observed that the throughputs of TDXPath remain almost constant on all the queries over all the dataset sizes. However, the throughputs of *VTD-XML* and *TinyXML* change significantly when the queries or dataset sizes change. For example, *VTD-XML* achieves $44.0MB/S$ on query $Q1$ over the dataset size $10KB$ while its throughput drops to $13.5MB/S$ on the same query over the dataset

size $1MB$. Similarly, over the same dataset size ($1MB$), its throughput achieves $88.5MB/S$ on query $Q2$ while it achieves $13.5MB/S$ on query $Q1$. This indicates TDXPath scales very well on both the queries and dataset sizes. TDXPath does not require buffering the potential matched results on the simple queries. Furthermore, the table-driven approach used in TDXPath pre-encodes the query expressions at compile time and performs checks at run-time linearly without backtracking. *TinyXML* achieves $9.0MB/S$ on query $Q1$ over the dataset size $10KB$ while its throughput drops to $0.03MB/S$ on the same query over the dataset size $1MB$. Similarly, over the same dataset size ($100KB$), its throughput achieves $1.4MB/S$ on query $Q1$ while it achieves $16.3MB/S$ on query $Q6$. This indicates TDXPath scales very well on both the queries and dataset sizes while *VTD-XML* and *TinyXML* do not. TDXPath does not require buffering the potential matched results on the simple queries. Furthermore, the table-driven approach used in TDXPath pre-encodes the query expressions at compile time and performs checks at run-time without backtracking.

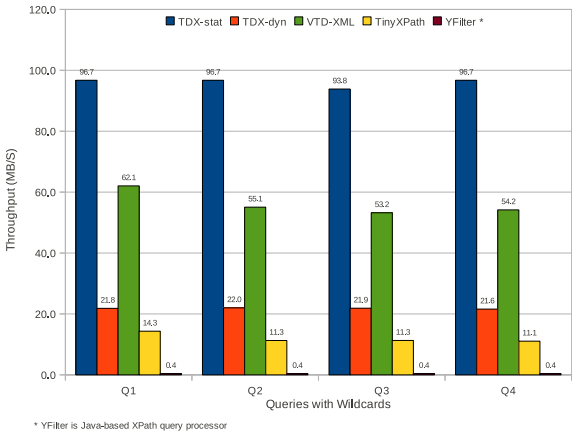
VTD-XML is considered one of world's the most fastest XPath 1.0 implementation due to the non-extractive tokenization. The non-extractive feature allows to keep the source text intact in memory, and use offsets and lengths to represent the XML document tokens. A VTD record uses a 64-bit integer to encode the the offset, length, token type and nesting depth of a token in an XML document. As a result, *VTD-XML* achieves memory efficiency and high throughput compared to other DOM based SAX based query engines. As the author claimed [207]: *"VTD-XML typically outperforms DOM parsers by* 5 *to* 10 *times, and it typically outperforms SAX parsers with null content handlers by about* 100%*."* Our experimental results in Figure 7.20 confirms this point. The results indicate the throughputs of *VTD-XML* range from $1.74MB/S$ to $89.0MB/S$, and it is several times faster than *TinyXML*, and up to several orders of magnitude faster than *YFilter*. Please note the *YFilter* is implemented in Java while all other query engines are implemented in C++. It may not be fair to do the direct performance comparison due to the overhead of Java VM (Virtual Machine) though, for one reason, *YFilter* is an FSA based XML filter that is considered a very fast compared to other automaton based alternatives. Another reason is that it provides overall performance comparison for application developers to choose right tools, in particular for performance-critical applications.

TinyXML is a simple, small, minimal, C++ XML and XPath parser focusing on the memory efficiency. It builds in-memory presentation of the XML document being parsed. Like other DOM based parsers and XPath query engines, it is not suitable to large XML documents or XML streams. The results indicate it is orders of magnitude slower than TDXPath.

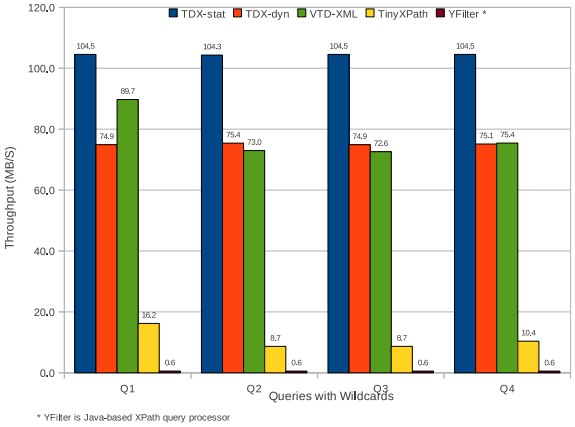
YFilter is an XML filtering system that provides fast, on-the-fly matching of XML- encoded data to large numbers of query specifications containing constraints on both structure and content. *YFilter* encodes path expressions using an NFA-based approach that enables highly-efficient, shared processing for large numbers of XPath expressions to reduce the number of machine states. Note that the *YFilter* is implemented in Java. Our experimental results suggest it is several orders slower tha TDXPath. Taking into the impact of Java VM, we can conclude it is still orders of magnitude slower than TDXPath, and its performance may be comparable to *TinyXML*, but still slower than *VTD-XML*.

Like our TDX in dynamic mode, TDXPath in dynamic mode suffers the one-time initialization overheads. This one-time costs impose performance impacts for the XML messages in small sizes. TDXPath-dyn is slower than *VTD-XML* over the dataset of size $10KB$ (Figure 7.20a). But as the dataset size increases, impact caused by such initial overheads will decreases quickly. Our experimental results show that TDXPath-dyn starts to outperform *VTD-XML* with the size of $100KB$ (Figure 7.20b. Starting with $1MB$, TDXPath-dyn achieves almost the same throughputs as the TDXPath-stat (Figure 7.20c).

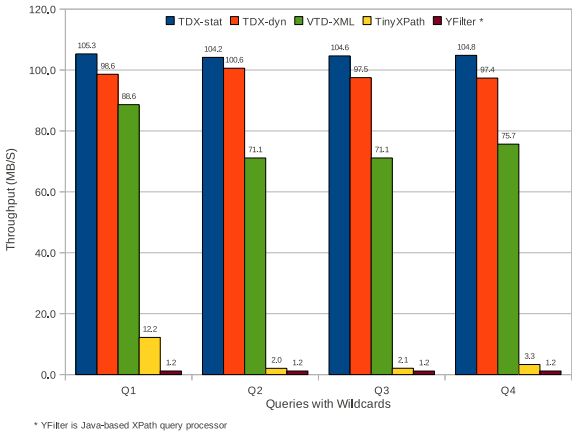
7.3.2 Performance of Queries with Wildcards



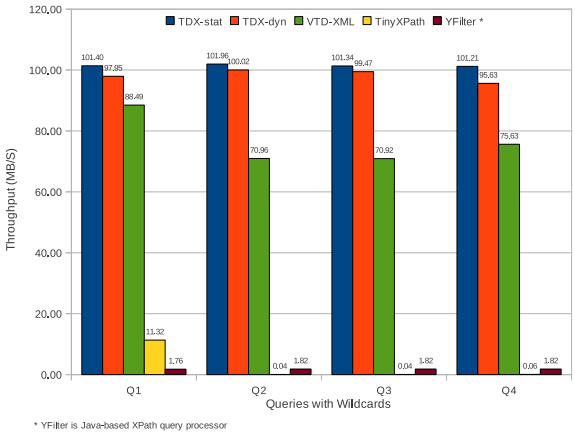
(a) Small size dataset (10 KB)



(b) Medium size dataset (100 KB)



(c) Large dataset (1 MB)



(d) Very large dataset (10 MB)

Q1: //*//*[position()=1]/productName/text()
Q2: items/item[position()=1]/productName/text()
Q3: item[position()=1]/productName/text()
Q4: /*/*/*[position()=1]/productName/text()

(e) Queries

Figure 7.21: Throughputs for different queries with wildcards over PurchaseOrder datasets in various sizes.

Wildcars ($\star$) selects nodes, attributes, texts regardless their positions in the query expressions. Different engines may have different ways to evaluate the wildcards in the queries. Figure 7.21 shows the performance comparison on the queries with wildcards over the datasets of various sizes, small, medium, large and very large. The results indicate our validating query processor outperform all

other query processor measured here. TDXPath achieves throughputs ranging from $93.8MB/S$ to $105.3MB/S$. It is up to 1.5 times faster than *VTD-XML*, 7 to 10 times faster than *TinyXMl* and 56 to 100 times faster than *YFilter*.

Similar to the performance on simple step queries, the throughputs of TDXPath remain almost constant on all the queries over all the datasets tested. TDXPath pre-processes the wildcards at compile time and marks the paring and query table using the concrete productions in the table instead. No wildcard is present in the table, and the query engine does not process any wildcard at all. As a result, queries with wildcards do not introduce any runtime cost to the TDXPath query engine. Because wildcards can be easily presented and processed in finite state machines, wildcards in automaton-based query processor do not introduce the overheads. As expected, the throughputs of *YFilter* remain roughly constant on the datasets with the same size. Due to the sizes of XML documents have impacts on the *YFilter*, the throughputs on different XML document sizes vary. Although the throughputs of *VTD-XML* vary as the figure shows, the margin is not significant. This indicates *VTD-XML* is also suitable to evaluate queries with wildcards well, yet not as good as TDXPath.

However, as the figure shows, *TinyXML* may have large impacts in evaluating queries with wildcards, in particular over the large XML document in size. In addition, the results also indicate *TinyXML* is not suitable to large XML document in size due to its DOM-based feature that requires build in-memory objects before query evaluation starts.

TDXPath in dynamic mode exhibits the same behavior on evaluating wildcards queries as on evaluating simple step queries. The initial costs impose large impact only over the small datasets in size. As the size increases, susch impacts drop dramatically.

7.3.3 Performance of Queries with Backward Axes

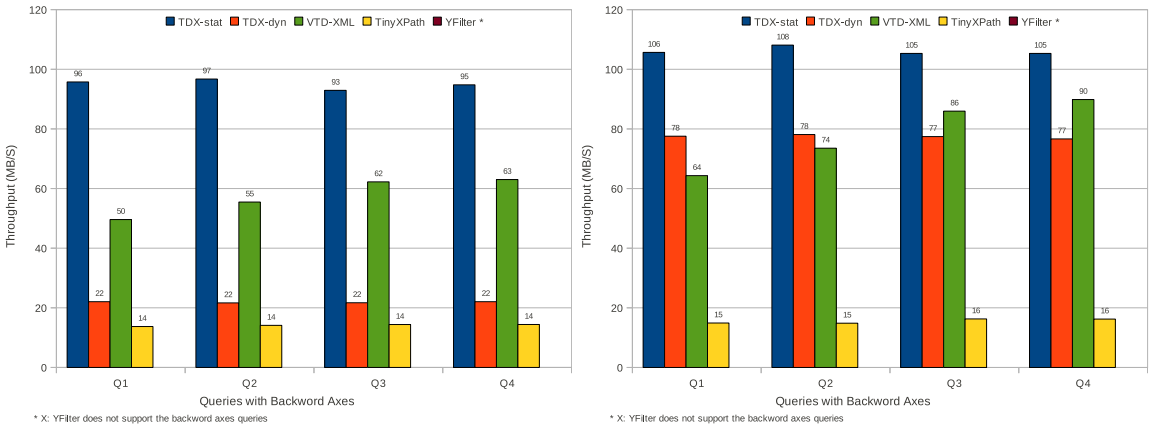
Backward axes impose challenges on query evaluation. Because backward axes are common used in query expressions, efficiently evaluating queries with backward axes is an important feature of XPath query processors. Some XPath query engines require buffering the messages and checking the predicates backwardly. We examined the performance of evaluating backward axes in TDXPathcompared with other query engines.

Figure 7.22 show the throughputs comparison on the queries with backward axes (also called reverse axes) consisting of `ancestor` and `parent` over various data sizes from small size of $10KB$ (Figure 7.22a) to very large data size $10MB$ (Figure 7.22d). The results indicate TDXPath achieves highest throughputs ranging from $93MB$ to $106MB$. The second highest throughputs achieved is *VTD-XML* which range from $50MB/S$ to $90MB/S$. This illustrates that our validating TDXPath is up to 1.5 times faster than non-validating *VTD-XML*. From the figure, we can see that TDXPath is 6 to 50 times faster than *TinyXML*, which achieves throughputs ranging from $2MB/S$ to $16MB/S$.

TDXPath pre-processes the queries at compile-time and marks the query table accordingly, that is, query table is independent of axes either backward or forward (`child` and `descendent`). As a result backward axes do not impose performance impacts. The results demonstrates again: the margin of the throughputs of TDXPath on various queries is pretty small.

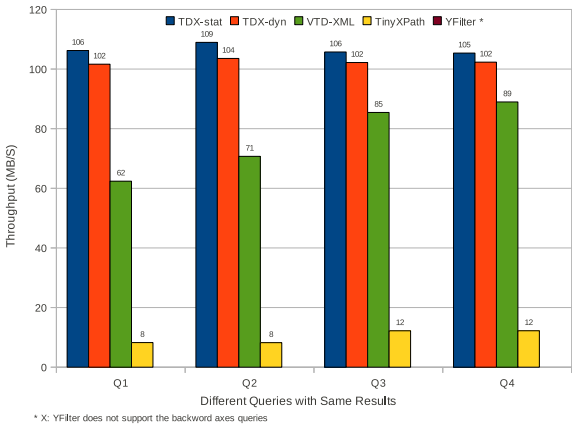
Because the current implementation of *YFilter* does not support backward axes feature, no throughputs of *YFilter* is shown in the figure.

Note that TDXPath in dynamic mode exhibits the same behavior on evaluating backward axes as on evaluating simple step queries and queries with wildcards.

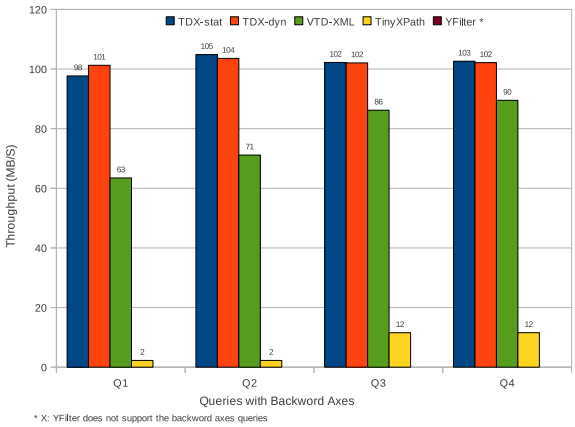


(a) Small dataset(10 KB)

(b) Medium dataset(100 KB)



(c) Large dataset(1 MB)



(d) Very large dataset(10 MB)

Q1: purchaseOrder/items/item/parent::*[position()=2]/quantity
Q2: purchaseOrder/items/item/ancestor::billTo
Q3: purchaseOrder/items/ancestor::*/item[position()=3]/shipDate
Q4: purchaseOrder/shipTo/parent::items/item[position()=3]/shipDate

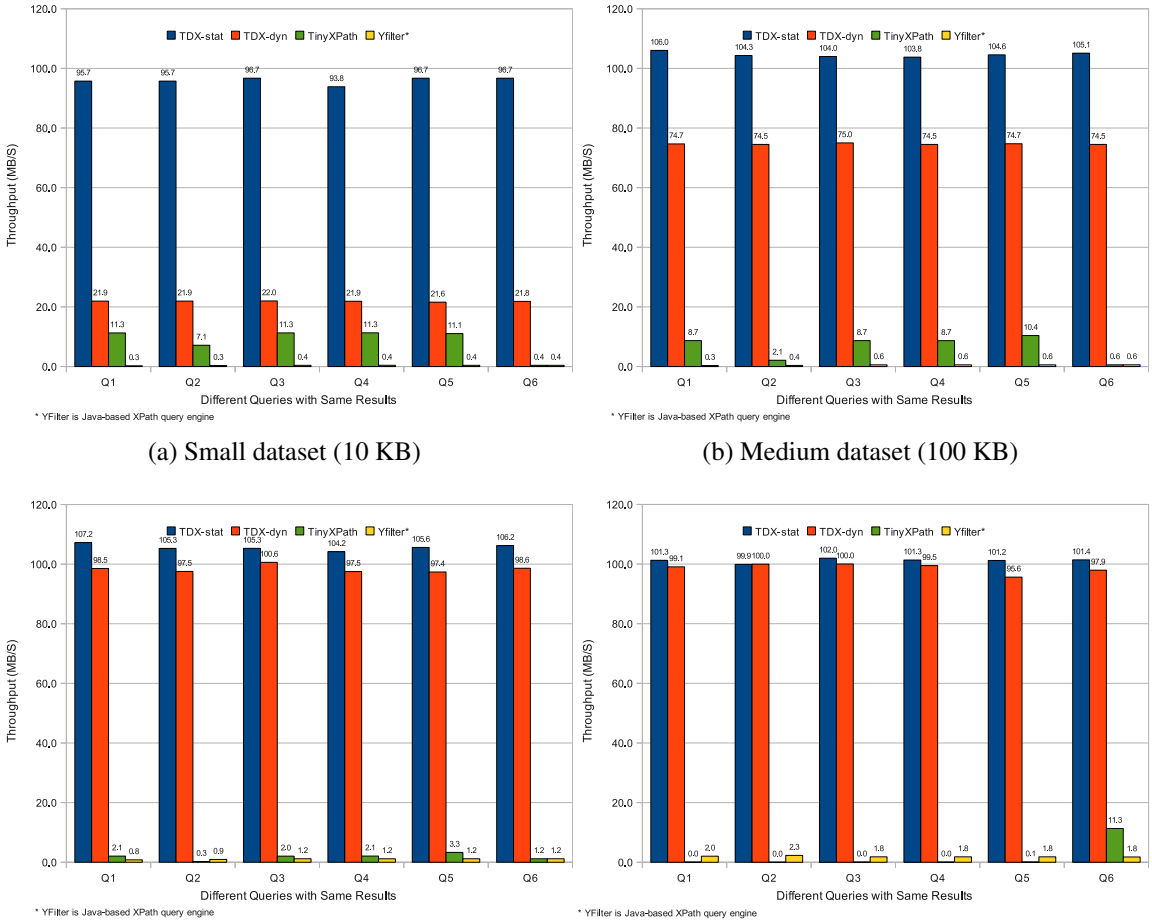
(e) Queries

Figure 7.22: Throughputs for different queries with backward axes over PurchaseOrder datasets in various sizes.

7.3.4 Performance of Equivalent Queries

To further examine the characteristics of different query processors processing the queries with different XPath features, we conducted experiments on evaluating queries that match the same results, we call them equivalent queries. The various queries $Q1$ ~$Q6$ matches the same results. Figure 7.23

illustrates the throughputs comparison on such equivalent queries over the dataset in various sizes from small to very large (Figure 7.23a ~Figure 7.23d).



(a) Small dataset (10 KB)

(b) Medium dataset (100 KB)

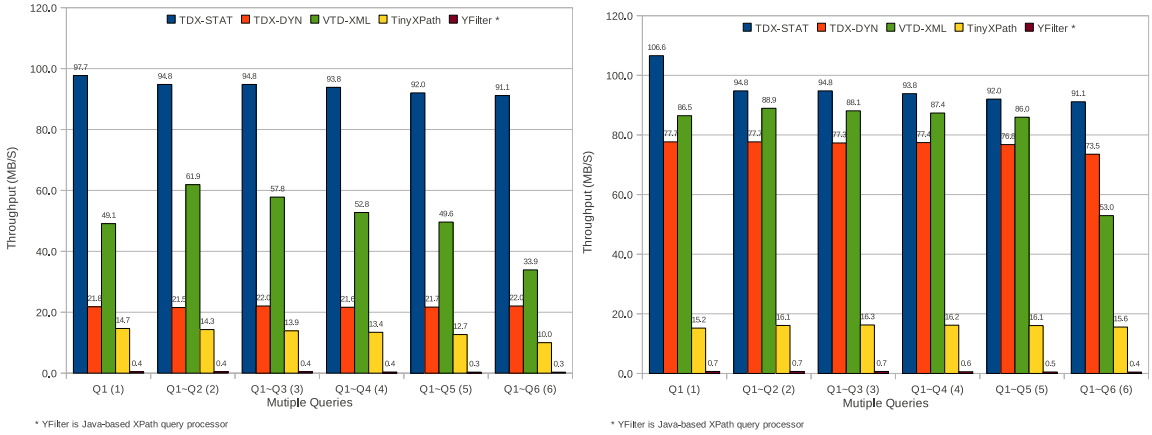
(c) Large dataset (1 MB)

(d) Very large dataset (10 MB)

Q1: purchaseOrder/items/item[position()=1]/productName/text()
Q2: items/item[position()=1]/productName/text()
Q3: items/item[position()=1]/productName/text()
Q4: item[position()=1]/productName/text()
Q5: /*/*/*[position()=1]/productName/text()
Q6: //*//*[position()=1]/productName/text()

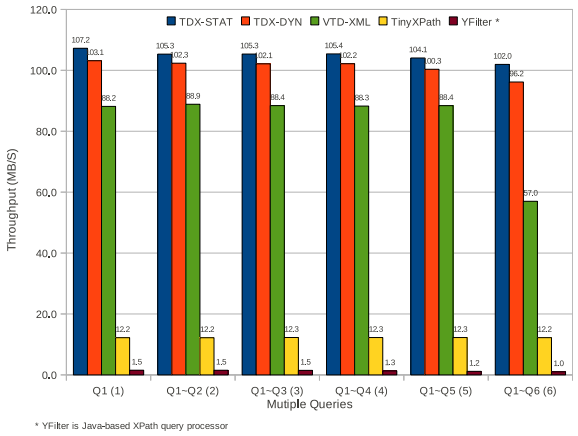
(e) Queries

Figure 7.23: Throughputs for different queries with the same results over PurchaseOrder datasets in various sizes.

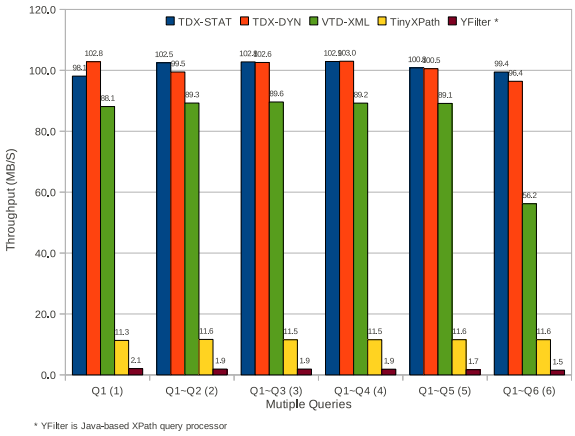


(a) Small dataset (10 KB)

(b) Medium dataset (100 KB)



(c) Large dataset (1 MB)



(d) Very large dataset (10 MB)

Q1: purchaseOrder/items/item[position()=1]/productName/text()
Q2: items/item[position()=1]/productName/text()
Q3: items/item[position()=1]/productName/text()
Q4: item[position()=1]/productName/text()
Q5: /*/*/*[position()=1]/productName/text()
Q6: //*//*[position()=1]/productName/text()

(e) Queries

Figure 7.24: Throughputs for multiple queries of PurchaseOrder datasets in various sizes.

The results indicate TDXPath outperforms all other measured query engines significantly. It is even at least 4 times faster than the query engine *VTD-XML* that achieves the second highest throughputs. It also shows that TDXPath is orders of magnitude faster than *TinyXML* and *YFilter*. Not only does TDXPath achieve very high throughputs ($93.8MB/S$ ~$107.2MB/S$) on all the queries over various dataset sizes, the its throughputs remain almost constant. Unlike XPath query

engines that optimizes some XPath query expression by sacrificing other features, TDXPath engine exhibits the independence feature on evaluating XPath features. This suggest the table-driven approach for evaluating XPath queries exhibits consistence performance on evaluating any XPath query expressions.

7.3.5 Performance of Multiple Queries

TDXPath merges multiple queries into a single query table at compile-time. We conducted experiments to examine the performance on evaluating different number of (from one to six queries). Figure 7.24 shows the throughputs comparison on evaluating these multiple queries over that datasets in different sizes (Figure 7.24a ~Figure 7.24d). The number in the parentheses are the number of queries, for example, the number three in the parentheses in Figure 7.24a means three queries are evaluated including $Q1$ through $Q3$ while the number six means six queries (queries $Q1$ through $Q6$) are evaluated in the experiment. These six queries consist of various XPath features including forward and backward axes, wildcards, text contents, attributes and predicates.

From the figure, we can see that TDXPath and *TinyXML* sare the top two engines that scale best to the number of queries among all the query engines measured. The throughputs of TDXPath drop 6.75%, 14.7%, 4.85% and 3.31% over the small, medium, larger and very large datasets respectively. The throughputs of *TinyXML* decrease up to 31.97% from valuating $Q1$ to evaluating $Q1$ through $Q6$ over the dataset of $10KB$ in size. However, its throughputs remain almost constant from evaluating one through six queries on very large dataset in size (Figure 7.24d).

YFilter comes to the third place on evaluating multiple queries. Its throughputs decrease 25.0%, 42.86%, 33.33% and 28.58% over the small, medium, larger and very large datasets respectively. The worst query engine among all the query engines measured is *VTDX-XML*. Its throughputs drop 45.23%, 40.08%, 42.54% and 44.3% over the small, medium, larger and very large datasets respectively.

7.3.6 Scalability

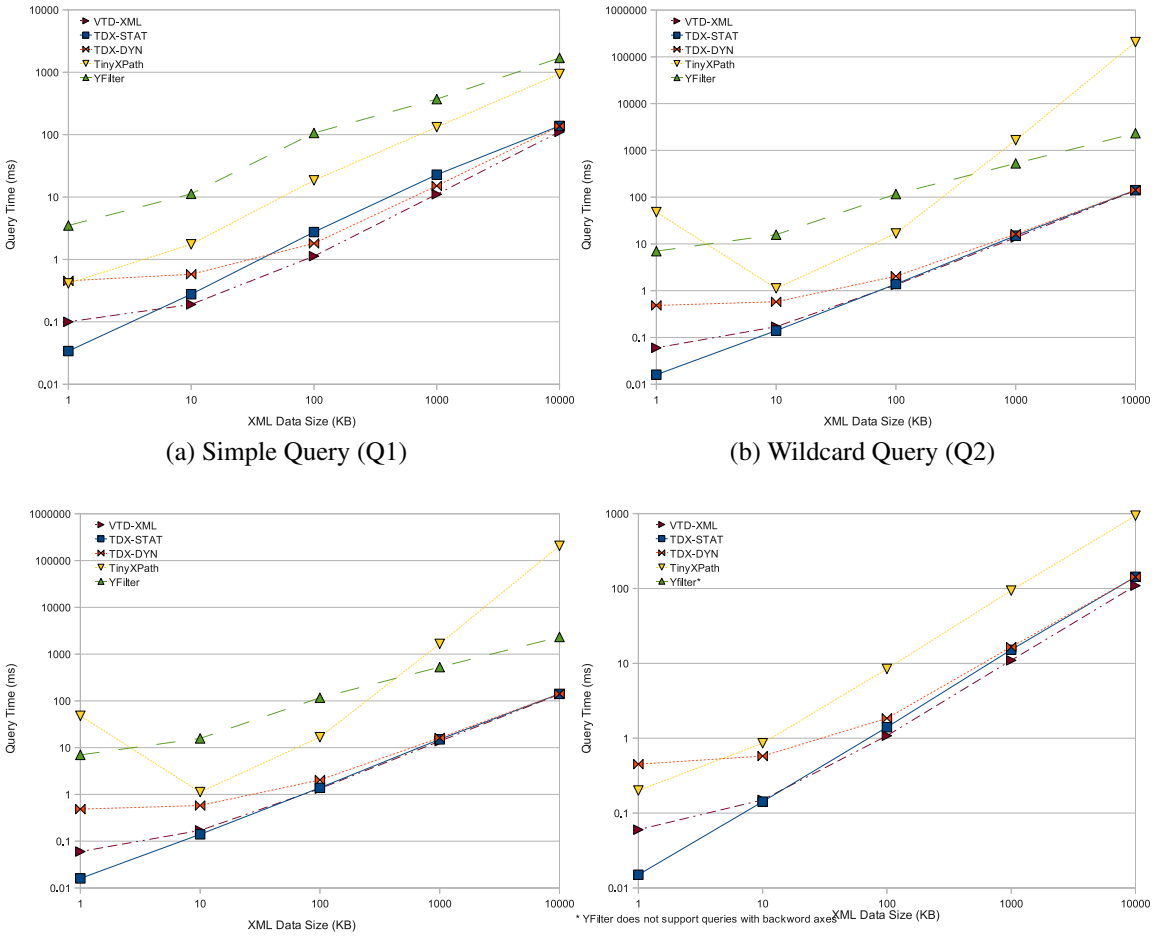
Scalability to the XML document size is measured in terms of evaluating time. Figure 7.25 illustrates the scalability on evaluating four different queries: simple step query ($Q1$), wildcard query ($Q2$), predicate query ($Q3$) and backward axis query ($Q4$). The X-axis in the dataset sizes in logarithm and the Y-axis denotes the evaluating time in logarithm.

The linear lines demonstrate the perfect scalability of TDXPath to the data sizes on evaluating all the four different queries $Q1$ through $Q4$. TDXPath in dynamic mode also scales very well as TDXPath in static mode except for the very small dataset in size (less than $100KB$ in our experiments) due to the initial overheads for populating the parsing and query table.

The curves indicate *VTD-XML* scales well starting with dataset size of $10KB$. Although *VTD-XML* keeps the entire XML document in memory, unlike DOM-based query engine, its fixed 64-bit integer token presentation ensures the scalability.

Figure 7.25a, Figure 7.25b and Figure 7.25c *YFilter* exhibits a fair scalability on evaluating simple step queries, queries with wildcards and predicates. Because *YFilter* does not support backward axes, the scalability is not available in Figure 7.25d.

The poor scalability of *TinyXML* is caused by its DOM-based feature. The curves show *TinyXML* exhibits inconsistent scalability. While it shows good scalability on evaluating simple step queries



(a) Simple Query (Q1)

(b) Wildcard Query (Q2)

(c) Predicate Query (Q3)

(d) Backward Axis Query (Q4)

Q1: /purchaseOrder/shipTo/zip
Q2: /*/*/*[position()=1]/productName/text()
Q3: /purchaseOrder/items/item[position()=1]/productName/text()
Q4: /purchaseOrder/shipTo/parent::items/item[position()=3]/shipDate

(e) Queries

Figure 7.25: Scalability to dataset size in terms of evaluating time.

(Figure 7.25a) and backward axis queries (Figure 7.25d), it does not scale on evaluating wildcard queries (Figure 7.25b) and predicate queries (Figure 7.25c).

7.4 Discussion

Our experimental results indicate TDXPath outperforms significantly all the query engines measured on all the queries with different XPath features. Wildcards, predicates, forward and backward axes, and multiple queries do not impose performance impacts. Throughputs of TDXPath are not affected by the queries with different XPath characteristics. It scales very well on these features.

Unlike query engines that specialize in some features (for example, forward and backward axes [20]), replace expensive queries with less expensive alternatives by re-writing rules, or build some data structures on-the-fly to improve the engine performance, TDXPath pre-processes the queries and marks the query table entries to encode the query states at compile-time. Different queries that match the same results have the same entries marked in the query table. Duplicate entries by multiple queries are marked only once. Thus no overhead is incurred on-the-fly at runtime. As a result, TDXPath achieves high-performance on all the XPath query characteristics.

TDXPath is a streaming query processor requiring no in-memory object presentation of the XML messages, and the buffering for potential matching results. As a result, TDXPath offers good scalability. As the performance results illustrate, DOM-like methods (for example, *TinyXML*) do not scale well due to the requirement of building in-memory object presentation for the entire XML documents. Thus is not suitable to large XML documents or streaming XML messages.

Queries are evaluated at runtime and depending on the different methods used to evaluate the queries, different queries matching the same results may affect the overall performance. This was demonstrated by our experimental results.

In addition, our TDXPath query processor integrates parsing, validation and query on XML streams. Streaming XML query engine typically sits on top of SAX parsers. Validating is typically either not supported depending on the underlaid parsers, or disabled in favor of the performance.

CHAPTER 8

CONCLUSIONS AND FUTURE WORKS

We have presented a novel framework for efficiently parsing, validating, and searching XML streams using compiler techniques. The generated XML parser expedites schema information and encodes such information in compact tabular forms at compile time, and utilizes an efficient runtime streaming parsing engine based on a two-stack pushdown automaton. The tabular tables are constructed from a set of augmented LL(1) grammars, which are generated from the schemas using a set of mapping rules. Type checking validation is accomplished by semantic actions associated with the grammar productions. Thus parsing and validation are performed simultaneously. Our experimental results show an order of magnitude performance improvements compared to widely used XML parsers. It even runs several times faster that non-validating XML parsers like the Expat [169].

We developed a set of mapping rules for efficiently translating XML schema content models into augmented LL(1) grammar. The arbitrary finite occurrence constraints and `xsd:all` groups of XML Schema pose challenges to traditional automaton based approaches. Arbitrary finite occurrence constraints can lead to an explosive growth in the number of states for simple automaton approaches. Models of `xsd:all` group and unordered `xsd:attribute` content cannot be represented in any standard regular expression syntax, and thus require significant augmentation of the automaton model. If translated directly into a standard automaton model, an `xsd:all` group results in an expansion of states that is combinatorial in the number of members of the group. We extended the LL(1) grammar by introducing two new grammar productions: *permutation phrase* grammar and *multi-occurrence* grammar. The former grammar encodes unordered `xsd:attribute` and `xsd:all` content models in an compact manner while the latter encodes finite arbitrary occurrences constraints in an efficient way. We developed algorithms for efficiently paring these extensions. The proposed table-driven based approach presented here implements a top-down, nonrecursive parser that we believe achieves most performance gains though, these augmented LL(1) grammar can be applied to a wide range of compiler techniques with the proposed algorithms. The mappings rule support full expressiveness of the XML Schema content models, and can be applied to other XML schemas such as Relax-NG [54].

The table-driven parsing engine only consults pre-generated tables at runtime, thus it is independent of any XML schema. This allows the associated tables to be populated on-the-fly to the parsing engine. Therefore, unlike other compile-based XML parsers (for example [183, 52, 151, 97]) that require recompilation when the schema from which they are generated, the table-driven approach does require recompilation when the schema is updated.

One of the properties that affects XML parsing inefficiency is the plain text format in XML. Comparisons on strings and conversions between string to in-memory representations are typically

very expensive. Our approach only scans the input once and breaks XML tag names into tokens that are represented as integers. Thus subsequent comparisons operate on integers rather than on strings, thus achieving a significant performance improvement.

By associating XPath expressions with the parsing table, the parsing table can be turned into a validating query processor for XML messages. The XML query processor can exploit all the benefits of the table-driven method.

The LL(1) grammar based table-driven method allows a symbol to associate user-provided application-specific actions with productions, thus offering a high-level flexibility.

In a word, the presented table-driven approach can be used for implementing standalone validating parsers and validating query processors. It can also be applied to many application domains, like Web services, subscription systems, content based routing to name a few.

APPENDIX A

LIST OF XML SCHEMAS USED IN PERFORMANCE EXPERIMENTS

Listing A.1: A sample XML Schema

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="
       http://ww2.cs.fsu.edu/~wzhang/tdx/sample.xsd" targetNamespace="
       http://ww2.cs.fsu.edu/~wzhang/tdx/sample.xsd">
3      <xsd:element name="a" type="AType"/>
4      <xsd:complexType name="AType">
5        <xsd:sequence>
6          <xsd:element name="b" type="BType"/>
7          <xsd:element name="g" type="GType"/>
8          <xsd:element name="n" type="NType"/>
9        </xsd:sequence>
10     </xsd:complexType>
11     <xsd:complexType name="BType">
12       <xsd:choice>
13         <xsd:element name="c" type="CType"/>
14         <xsd:element name="f" type="FType"/>
15       </xsd:choice>
16     </xsd:complexType>
17     <xsd:complexType name="CType">
18       <xsd:sequence>
19         <xsd:element name="d" type="xsd:string"/>
20         <xsd:element name="e" type="xsd:string"/>
21       </xsd:sequence>
22     </xsd:complexType>
23     <xsd:complexType name="FType">
24       <xsd:sequence>
25         <xsd:element name="f" type="BType"/>
26       </xsd:sequence>
27     </xsd:complexType>
28     <xsd:complexType name="GType">
29       <xsd:sequence>
30         <xsd:element name="h" type="HType"/>
31       </xsd:sequence>
```

```
32      </xsd:complexType>
33      <xsd:complexType name="HType">
34        <xsd:sequence>
35          <xsd:element name="i" type="IType" minOccurs="0" maxOccurs=
                "unbounded"/>
36        </xsd:sequence>
37      </xsd:complexType>
38      <xsd:complexType name="IType">
39        <xsd:sequence>
40          <xsd:element name="j" type="JType"/>
41          <xsd:element name="m" type="MType"/>
42        </xsd:sequence>
43      </xsd:complexType>
44      <xsd:complexType name="JType">
45        <xsd:sequence>
46          <xsd:element name="k" type="xsd:string"/>
47        </xsd:sequence>
48      </xsd:complexType>
49      <xsd:complexType name="MType">
50        <xsd:sequence>
51          <xsd:element name="d" type="xsd:string"/>
52        </xsd:sequence>
53      </xsd:complexType>
54      <xsd:complexType name="NType">
55        <xsd:sequence>
56          <xsd:element name="n" type="xsd:string"/>
57        </xsd:sequence>
58      </xsd:complexType>
59  </xsd:schema>
```

Listing A.2: XML Schema of Structure

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3    xmlns="http://ww2.cs.fsu.edu/~wzhang/tdx/Strcture.xsd"
         targetNamespace="http://ww2.cs.fsu.edu/~wzhang/tdx/Strcture.
         xsd"
4    xmlns:tns="http://ww2.cs.fsu.edu/~wzhang/tdx/Strcture.xsd"
5    elementFormDefault="qualified">
6
7    <xsd:simpleType name="EnumerationType">
8      <xsd:restriction base="xsd:string">
9        <xsd:enumeration value="optional" />
10       <xsd:enumeration value="required" />
11       <xsd:enumeration value="prohibited" />
12     </xsd:restriction>
13   </xsd:simpleType>
14
15   <xsd:simpleType name="PhoneType">
16     <xsd:restriction base="xsd:string">
17       <xsd:pattern value="[1-9][0-9]{2}-[1-9][0-9]{2}-[0-9]{4}" />
18     </xsd:restriction>
```

```
19  </xsd:simpleType>
20
21  <xsd:complexType name="SequenceType">
22    <xsd:sequence>
23      <xsd:element name="Name" type="xsd:string" />
24      <xsd:element name="Title" type="xsd:string" minOccurs="0" />
25      <xsd:element name="Author" type="xsd:string" minOccurs="0"
26        maxOccurs="1" />
27      <xsd:element name="Reference" type="xsd:string" minOccurs="0"
28        maxOccurs="unbounded" />
29      <xsd:element name="Number" type="xsd:int" />
30      <xsd:element name="Amount" type="xsd:double" />
31      <xsd:element name="Year" type="xsd:date" />
32      <xsd:element name="Phone" type="PhoneType" />
33      <xsd:element name="List" type="EnumerationType" />
34
35    </xsd:sequence>
36    <xsd:attribute name="ISBN" type="xsd:string" use="required" />
37    <xsd:attribute name="Published" type="xsd:boolean" use="
        required" />
38  </xsd:complexType>
39
40  <xsd:complexType name="AllType">
41    <xsd:all>
42      <xsd:element name="Name" type="xsd:string" />
43      <xsd:element name="Title" type="xsd:string" minOccurs="0" />
44      <xsd:element name="Author" type="xsd:string" minOccurs="0"
45        maxOccurs="1" />
46      <xsd:element name="Reference" type="xsd:string" minOccurs="0"
47        maxOccurs="1" />
48      <xsd:element name="Number" type="xsd:int" />
49      <xsd:element name="Amount" type="xsd:double" />
50      <xsd:element name="Year" type="xsd:date" />
51      <xsd:element name="Phone" type="PhoneType" />
52      <xsd:element name="List" type="EnumerationType" />
53
54    </xsd:all>
55    <xsd:attribute name="ISBN" type="xsd:string" use="required" />
56    <xsd:attribute name="Published" type="xsd:boolean" use="
        required" />
57  </xsd:complexType>
58
59  <xsd:complexType name="ChoiceType">
60    <xsd:choice>
61      <xsd:element name="Name" type="xsd:string" />
62      <xsd:element name="Title" type="xsd:string" minOccurs="0" />
63      <xsd:element name="Author" type="xsd:string" minOccurs="0"
64        maxOccurs="1" />
65      <xsd:element name="Reference" type="xsd:string" minOccurs="0"
66        maxOccurs="1" />
67      <xsd:element name="Number" type="xsd:int" />
```

```
68        <xsd:element name="Amount" type="xsd:double" />
69        <xsd:element name="Year" type="xsd:date" />
70        <xsd:element name="Phone" type="PhoneType" />
71        <xsd:element name="List" type="EnumerationType" />
72      </xsd:choice>
73      <xsd:attribute name="ISBN" type="xsd:string" use="required" />
74      <xsd:attribute name="Published" type="xsd:boolean" use="
          required" />
75    </xsd:complexType>
76
77    <xsd:complexType name="StructureType">
78      <xsd:sequence>
79        <xsd:element name="Sequence" type="SequenceType" />
80        <xsd:element name="All" type="AllType" />
81        <xsd:element name="Choice" type="ChoiceType" />
82      </xsd:sequence>
83    </xsd:complexType>
84
85    <xsd:complexType name="StructuresType">
86      <xsd:sequence>
87        <xsd:element name="Structure" type="StructureType"
88          minOccurs="0" maxOccurs="unbounded" />
89      </xsd:sequence>
90    </xsd:complexType>
91
92    <xsd:element name="Structures" type="StructuresType" />
93
94  </xsd:schema>
```

Listing A.3: XML Schema of PurchaseOrder

```
1  <xsd:schema xmlns:xsd="http://www.w4.org/2001/XMLSchema"
       targetNamespace="http://tempuri.org/po.xsd"
2  xmlns="http://tempuri.org/po.xsd" elementFormDefault="qualified">
3
4    <xsd:element name="purchaseOrder" type="PurchaseOrderType" />
5    <xsd:element name="comment" type="xsd:string" />
6
7    <xsd:complexType name="PurchaseOrderType">
8      <xsd:sequence>
9        <xsd:element name="shipTo" type="USAddress" />
10       <xsd:element name="billTo" type="USAddress" />
11       <xsd:element ref="comment" minOccurs="0" />
12       <xsd:element name="items" type="Items" />
13     </xsd:sequence>
14     <xsd:attribute name="orderDate" type="xsd:date" />
15   </xsd:complexType>
16
17   <xsd:complexType name="USAddress">
18
19     <xsd:sequence>
20       <xsd:element name="name" type="xsd:string" />
```

```
      <xsd:element name="street" type="xsd:string" />
      <xsd:element name="city" type="xsd:string" />
      <xsd:element name="state" type="xsd:string" />
      <xsd:element name="zip" type="xsd:decimal" />
    </xsd:sequence>
    <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US" />
  </xsd:complexType>

  <xsd:complexType name="Items">
    <xsd:sequence>
      <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="productName" type="xsd:string" />
            <xsd:element name="quantity">
              <xsd:simpleType>
                <xsd:restriction base="xsd:positiveInteger">
                  <xsd:maxExclusive value="100" />
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="USPrice" type="xsd:decimal" />
            <xsd:element ref="comment" minOccurs="0" />
            <xsd:element name="shipDate" type="xsd:date" minOccurs=
               "0" />
          </xsd:sequence>
          <xsd:attribute name="partNum" type="SKU" use="required" /
             >
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>

  <!-- Stock Keeping Unit, a code for identifying products -->
  <xsd:simpleType name="SKU">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{3}-[A-Z]{2}" />
    </xsd:restriction>
  </xsd:simpleType>

</xsd:schema>
```

Listing A.4: XML Schema of AmazonSearch-all

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.amazon.com/AmazonSearch.xsd"
xmlns="http://www.amazon.com/AmazonSearch.xsd" elementFormDefault="
   qualified">

  <xsd:complexType name="ProductType">
    <xsd:all>
      <xsd:element name="AgeGroup" type="xsd:string" />
```

```
      <xsd:element name="Accessories" type="xsd:string" />
      <xsd:element name="Artists" type="xsd:string" />
      <xsd:element name="Asin" type="xsd:string" />
      <xsd:element name="Authors" type="xsd:string" />
      <xsd:element name="Availability" type="xsd:boolean" />
      <xsd:element name="BrowseList" type="xsd:string" />
      <xsd:element name="Catalog" type="xsd:string" />
      <xsd:element name="DeweyNumber" type="xsd:string" />
      <xsd:element name="CollectiblePrice" type="xsd:double" />
      <xsd:element name="Encoding" type="xsd:string" />
      <xsd:element name="EsrbRating" type="xsd:string" />
      <xsd:element name="Distributor" type="xsd:string" />
      <xsd:element name="Features" type="xsd:string" />
      <xsd:element name="IssuesPerYear" type="xsd:string" />
      <xsd:element name="Isbn" type="xsd:string" />
      <xsd:element name="Lists" type="xsd:string" />
      <xsd:element name="ImageUrlSmall" type="xsd:string" />
      <xsd:element name="ImageUrlMedium" type="xsd:string" />
      <xsd:element name="ImageUrlLarge" type="xsd:string" />
      <xsd:element name="ListPrice" type="xsd:double" />
      <xsd:element name="ProductName" type="xsd:string" />
      <xsd:element name="KeyPhrases" type="xsd:string" />
      <xsd:element name="Mpn" type="xsd:string" />
      <xsd:element name="Starring" type="xsd:string" />
      <xsd:element name="Directors" type="xsd:string" />
      <xsd:element name="TheatricalReleaseDate" type="xsd:string" /
          >
      <xsd:element name="ReleaseDate" type="xsd:string" />
      <xsd:element name="Manufacturer" type="xsd:string" />
      <xsd:element name="OurPrice" type="xsd:double" />
      <xsd:element name="UsedPrice" type="xsd:double" />
      <xsd:element name="RefurbishedPrice" type="xsd:double" />
      <xsd:element name="ThirdPartyNewPrice" type="xsd:double" />
      <xsd:element name="SalesRank" type="xsd:string" />
      <xsd:element name="Media" type="xsd:string" />
      <xsd:element name="ReadingLevel" type="xsd:string" />
      <xsd:element name="NumberOfPages" type="xsd:string" />
      <xsd:element name="NumberOfIssues" type="xsd:string" />
      <xsd:element name="SubscriptionLength" type="xsd:string" />
      <xsd:element name="RunningTime" type="xsd:int" />
      <xsd:element name="Publisher" type="xsd:string" />
      <xsd:element name="NumMedia" type="xsd:string" />
      <xsd:element name="MpaaRating" type="xsd:string" />
      <xsd:element name="Upc" type="xsd:string" />
      <xsd:element name="Tracks" type="xsd:string" />
      <xsd:element name="Platforms" type="xsd:string" />
      <xsd:element name="Reviews" type="xsd:string" />
      <xsd:element name="SimilarProducts" type="xsd:string" />
      <xsd:element name="Status" type="xsd:string" />
      <xsd:element name="Url" type="xsd:string" />
    </xsd:all>
```

```
  </xsd:complexType>

  <xsd:element name="ProductInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Product" type="ProductType"
          maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing A.5: XML Schema of AmazonSearch-sequence

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.amazon.com/AmazonSearch.xsd"
xmlns="http://www.amazon.com/AmazonSearch.xsd" elementFormDefault="
    qualified">

  <xsd:complexType name="ProductType">
    <xsd:sequence>
      <xsd:element name="AgeGroup" type="xsd:string" />
      <xsd:element name="Accessories" type="xsd:string" />
      <xsd:element name="Artists" type="xsd:string" />
      <xsd:element name="Asin" type="xsd:string" />
      <xsd:element name="Authors" type="xsd:string" />
      <xsd:element name="Availability" type="xsd:boolean" />
      <xsd:element name="BrowseList" type="xsd:string" />
      <xsd:element name="Catalog" type="xsd:string" />
      <xsd:element name="DeweyNumber" type="xsd:string" />
      <xsd:element name="CollectiblePrice" type="xsd:double" />
      <xsd:element name="Encoding" type="xsd:string" />
      <xsd:element name="EsrbRating" type="xsd:string" />
      <xsd:element name="Distributor" type="xsd:string" />
      <xsd:element name="Features" type="xsd:string" />
      <xsd:element name="IssuesPerYear" type="xsd:string" />
      <xsd:element name="Isbn" type="xsd:string" />
      <xsd:element name="Lists" type="xsd:string" />
      <xsd:element name="ImageUrlSmall" type="xsd:string" />
      <xsd:element name="ImageUrlMedium" type="xsd:string" />
      <xsd:element name="ImageUrlLarge" type="xsd:string" />
      <xsd:element name="ListPrice" type="xsd:double" />
      <xsd:element name="ProductName" type="xsd:string" />
      <xsd:element name="KeyPhrases" type="xsd:string" />
      <xsd:element name="Mpn" type="xsd:string" />
      <xsd:element name="Starring" type="xsd:string" />
      <xsd:element name="Directors" type="xsd:string" />
      <xsd:element name="TheatricalReleaseDate" type="xsd:string" /
          >
      <xsd:element name="ReleaseDate" type="xsd:string" />
      <xsd:element name="Manufacturer" type="xsd:string" />
      <xsd:element name="OurPrice" type="xsd:double" />
```

```
      <xsd:element name="UsedPrice" type="xsd:double" />
      <xsd:element name="RefurbishedPrice" type="xsd:double" />
      <xsd:element name="ThirdPartyNewPrice" type="xsd:double" />
      <xsd:element name="SalesRank" type="xsd:string" />
      <xsd:element name="Media" type="xsd:string" />
      <xsd:element name="ReadingLevel" type="xsd:string" />
      <xsd:element name="NumberOfPages" type="xsd:string" />
      <xsd:element name="NumberOfIssues" type="xsd:string" />
      <xsd:element name="SubscriptionLength" type="xsd:string" />
      <xsd:element name="RunningTime" type="xsd:int" />
      <xsd:element name="Publisher" type="xsd:string" />
      <xsd:element name="NumMedia" type="xsd:string" />
      <xsd:element name="MpaaRating" type="xsd:string" />
      <xsd:element name="Upc" type="xsd:string" />
      <xsd:element name="Tracks" type="xsd:string" />
      <xsd:element name="Platforms" type="xsd:string" />
      <xsd:element name="Reviews" type="xsd:string" />
      <xsd:element name="SimilarProducts" type="xsd:string" />
      <xsd:element name="Status" type="xsd:string" />
      <xsd:element name="Url" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:element name="ProductInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Product" type="ProductType"
          maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

BIBLIOGRAPHY

[1] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental maintenance for materialized views over semistructured data. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 38–49, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[2] S. Abraham. Some questions of phrase-structure grammars. *Computational linguistics*, 4:61–70, 1965.

[3] N. Abu-Ghazaleh and M. J. Lewis. Differential checkpointing for reducing memory requirements in optimized SOAP deserialization. In *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 250–255, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Nayef Abu-Ghazaleh and Michael J. Lewis. Differential deserialization for optimized SOAP performance. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 21, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Nayef Abu-Ghazaleh and Michael J. Lewis. Lightweight checkpointing for faster SOAP deserialization. In *Proceedings of IEEE International Conference on Web Services (ICWS '06)*, Chicago IL, September 18-20, 2006.

[6] Nayef Abu-Ghazaleh, Michael J. Lewis, and Madhusudhan Govindaraju. Performance of dynamic resizing of message fields for differential serialization of SOAP messages. In *Proceedings of the International Symposium on Web Services and Applications*, pages 783–789, June 2004.

[7] Joaquín Adiego, Gonzalo Navarro, and Pablo de la Fuente. Using structural contexts to compress semistructured text collections. *Information Processing Management*, 43(3):769–790, 2007.

[8] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, Reading MA, 1985.

[9] Alfred V. Aho. Indexed grammars—an extension of context-free grammars. *J. ACM*, 15(4):647–671, 1968.

[10] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.

[11] Mehmet Altinel and Michael J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 53–64, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[12] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of tree pattern queries. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 497–508, New York, NY, USA, 2001. ACM.

[13] Apache.org. Xalan xslt stylesheet processor. http://xml.apache.org.

[14] Andrei Arion, Angela Bonifati, Gianni Costa, Sandra D'Aguanno, Ioana Manolescu, and Andrea Pugliese. XQueC: pushing queries to compressed xml data. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 1065–1068. VLDB Endowment, 2003.

[15] Iliana Avila-Campillo, Todd J. Green, Ashish Gupta, Makoto Onizuka, Demian Raven, , and Dan Suciu. XMLTK: An XML toolkit for scalable XML stream processing. In *Programming Language Technologies for XML(PLAN-X)*, Pittsburgh, PA, October 2002.

[16] Andrey Balmin, Fatma, Özcan, Kevin S. Beyer, Roberta J. Cochrane, and Hamid Pirahesh. A framework for using materialized xpath views in xml query, 2004.

[17] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. Incremental validation of xml documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004.

[18] Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. Buffering in query evaluation over xml streams. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 216–227, New York, NY, USA, 2005. ACM.

[19] Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. On the memory requirements of xpath evaluation over xml streams. *J. Comput. Syst. Sci.*, 73(3):391–441, 2007.

[20] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming xpath processing with forward and backward axes. In *Proceedings of the 19th International Conference on Data Engineering (ICDE2003)*, pages 455–466, Bangalore, India, March 2003.

[21] Peter Baumgartner, Ulrich Furbach, Margret Groß-Hardt, and Thomas Kleemann. Optimizing the evaluation of xpath using description logics. In *INAP/WLP*, pages 1–15, 2004.

[22] R. J. Bayardo, D. Gruhl, V. Josifovski, and J. Myllymaki. An evaluation of binary xml encoding optimizations for fast stream based xml processing. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 345–354, New York, NY, USA, 2004. ACM.

[23] Michael Benedikt, Glenn Bruns, Julie Gibson, Robin Kuss, and Amy Ng. Automated update management for XML integrity constraints. In *PLan-X'02: Workshop on Programming Languages for XML*, 2002.

[24] Michael Benedikt and Christoph Koch. XPath leashed. *ACM Comput. Surv.*, 41(1):1–54, 2008.

[25] Yves Berquin. Tinyxpath, April 2011. http://tinyxpath.sourceforge.net/.

[26] Henrik Björklund, Wouter Gelade, Marcel Marquardt, and Wim Martens. Incremental XPath evaluation. In *ICDT '09: Proceedings of the 12th International Conference on Database Theory*, pages 162–173, New York, NY, USA, 2009. ACM.

[27] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[28] Klemens Bohm, Kathrin Gayer, Karl Aberer, and M. Tamer. Query optimization for structured documents based on knowledge on the document type definition. In *Proc. of the Advances in Digital Libraries Conference*, pages 196–205, 1998.

[29] Rajesh Bordawekar, Lipyeow Lim, and Oded Shmueli. Parallelization of XPath queries using multi-core processors: challenges and experiences. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 180–191, New York, NY, USA, 2009. ACM.

[30] Stefan Böttcher. Schema-based query optimization for xquery queries. In *9th East-European Conference on Advances in Databases and Information Systems (ADBIS 2005)*, 2005.

[31] Béatrice Bouchou and Mirian Halfeld Ferrari Alves. Updates and incremental validation of xml documents. In *Eigth Int'l Symp. Database programming Languages*, pages 216–232, 2003.

[32] M. Brantner, C.-C. Kanne, G. Moerkotte, and S. Helmer. Algebraic optimization of nested XPath expressions. In *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, pages 128–128, April 2006.

[33] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Full-fledged algebraic XPath processing in natix. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 705–716, Washington, DC, USA, 2005. IEEE Computer Society.

[34] F. Bry, F. Coskun, S. Durmaz, T. Furche, D. Olteanu, and M. Spannagel. The xml stream query processor spex. In *Proceedings of 21st International Conference on Data Engineering*, pages 1120–1121, April 2005.

[35] Fabian E. Bustamante, Greg Eisenhauer, Karsten Schwan, and Patrick Widener. Efficient wire formats for high performance computing. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 39, Washington, DC, USA, 2000. IEEE Computer Society.

[36] Robert D. Cameron. Extending context-free grammars with permutation phrases. *ACM Letters on Program Languages and Systems*, 2(1-4):85–94, 1993.

[37] Robert D. Cameron. A case study in SIMD text processing with parallel bit streams: Utf-8 to utf-16 transcoding. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 91–98, New York, NY, USA, 2008. ACM.

[38] K. Selçuk Candan, Wang-Pin Hsiung, Songting Chen, Junichi Tatemura, and Divyakant Agrawal. AFilter: adaptable XML filtering with prefix-caching suffix-clustering. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 559–570. VLDB Endowment, 2006.

[39] Julien Carme, Joachim Niehren, and Marc Tommasi. Querying unranked trees with stepwise tree automata. In Vincent van Oostrom, editor, *International Conference on Rewriting Techniques and Applications, Aachen*, volume 3091 of *Lecture Notes in Computer Science*, pages 105–118. Springer, June 2004.

[40] Chee-Yong Chan, Pascal Felber, Minos Garofalakis, and RajeevRastogi. Efficient filtering of xml documents with xpath expressions. *The VLDB Journal*, 11(4):354–379, 2002.

[41] Yi Chen, Susuan B. Davidson, and Yifeng Zhang. An efficient XPath query processor for XML streams. In *Proceedings of the 22nd International Conference on Data Engineering*, pages 79–90, 2006.

[42] James Cheney. Compressing XML with multiplexed hierarchical PPM models. In *DCC '01: Proceedings of the Data Compression Conference*, page 163, Washington, DC, USA, 2001. IEEE Computer Society.

[43] James Cheney. Compressing XML with multiplexed hierarchical PPM models. In *DCC '01: Proceedings of the Data Compression Conference*, page 163, Washington, DC, USA, 2001. IEEE Computer Society.

[44] James Cheng and Wilfred Ng. XQzip: Querying compressed XML using structural indexing. In *EDBT*, pages 219–236, 2004.

[45] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient structural joins on indexed XML documents. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 263–274. VLDB Endowment, 2002.

[46] Suren A. Chilingaryan. XML benchmark. http://xmlbench.sourceforge.net/.

[47] Cristiana Chitic and Daniela Rosu. On validation of XML streams using finite state machines. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 85–90, New York, NY, USA, 2004. ACM.

[48] Ai-Ti Chiu and Jia-Lien Hsu. An automaton-based filtering system for streaming musicxml. In *SWWS*, pages 177–178, 2006.

[49] Kenneth Chiu. A compiler-based approach to schema-specific XML parsing. Technical Report Computer Science Technical Report 592, Indiana University, 2003.

[50] Kenneth Chiu, Tharaka Devadithya, Wei Lu, and Aleksander Slominski. A binary XML for scientific applications. In *E-SCIENCE '05: Proceedings of the First International Conference on e-Science and Grid Computing*, pages 336–343, Washington, DC, USA, 2005. IEEE Computer Society.

[51] Kenneth Chiu, Madhusudhan Govindaraju, and Randall Bramley. Investigating the limits of soap performance for scientific computing. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 246, Washington, DC, USA, 2002. IEEE Computer Society.

[52] Kenneth Chiu and Wei Lu. A compiler-based approach to schema-specific XML parsing. In *Proceedings of The First International Workshop on High Performance XML Processing*, 2004.

[53] James Clark. Xsl transformations (xslt) version 1.0. W3C Recommendation, November 16 1999. http:/www.w3.org/TR/xslt.

[54] James Clark and MURATA Makoto. Relax NG specification, November 2001. http://relaxng.org/spec-20011203.html.

[55] Intel Corporation. Intel xml software suite 1.0. http://www.intel.com/cd/software/products/asmo-na/eng/366637.htm.

[56] Dan Davis and Manish Parashar. Latency performance of SOAP implementations. In *proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid*, Berlin, Germany, May 2002.

[57] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.

[58] Yanlei Diao, Peter Fischer, Michael J. Franklin, and Raymond To. YFilter: Efficient and scalable filtering of XML documents. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, page 341, Washington, DC, USA, 2002. IEEE Computer Society.

[59] Yanlei Diao and Michael J. Franklin. High-performance XML filtering: An overview of YFilter. *IEEE Data Eng. Bull.*, 26(1):41–48, 2003.

[60] W3C XML Specification DTD. XML metadata interchange (XMI) specifications. Available from http://www.omg.org/.

[61] Craig Bruce (ed.). Binary-xml encoding specification, version 0.0.8. May 2003. http://www.opengis.org/techno/discussions/03-002r8.pdf.

[62] Françoise Fabret, H. Arno Jacobsen, François Llirbat, Joăo Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD Rec.*, 30:115–126, May 2001.

[63] Mary F. Fernandez and Dan Suciu. Optimizing regular path expressions using graph schemas. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pages 14–23, Washington, DC, USA, 1998. IEEE Computer Society.

[64] Thorsten Fiebig and Guido Moerkotte. Evaluating queries on structure with extended access support relations. In *Selected papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases*, pages 125–136, London, UK, 2001. Springer-Verlag.

[65] M. Fisher. *Grammars with macrolike productions*. PhD thesis, Harvard University, Cambridge, MA, 1968.

[66] Daniela Florescu. Managing semi-structured data. *ACM Queue*, 3(8):18–24, 2005.

[67] Apache Foundation. Xerces xml parser. Ghttp://xerces.apache.org/.

[68] Massimo Franceschet. XPathMark: Functional and performance tests for XPath. In Peter A. Boncz, Torsten Grust, Jérôme Siméon, and Maurice van Keulen, editors, *XQuery Implementation Paradigms*, number 06472 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[69] Markus Frick, Martin Grohe, and Christoph Koch. Query evaluation on compressed trees (extended abstract). In *LICS '03: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, page 188, Washington, DC, USA, 2003. IEEE Computer Society.

[70] Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Efficient static analysis of XML paths and types. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 342–351, New York, NY, USA, 2007. ACM.

[71] Pierre Genevès and Jean-Yves Vion-Dury. Logic-based XPath optimization. In *DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering*, pages 211–219, New York, NY, USA, 2004. ACM.

[72] Georg, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing xpath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

[73] Marc Girardot and Neel Sundaresan. Millau: an encoding format for efficient representation and exchange of XML over the Web. *Comput. Netw.*, 33(1-6):747–765, 2000.

[74] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 95–106. VLDB Endowment, 2002.

[75] Georg Gottlob, Christoph Koch, and Reinhard Pichler. The complexity of xpath query evaluation. In *PODS '03: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 179–190, New York, NY, USA, 2003. ACM.

[76] Madhusudhan Govindaraju, Aleksander Slominski, Venkatesh Choppella, Randall Bramley, and Dennis Gannon. Requirements for and evaluation of RMI protocols for scientific

computing. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, pages 61–86, Washington, DC, USA, 2000. IEEE Computer Society.

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

[78] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata. In *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, pages 173–189, London, UK, 2002. Springer-Verlag.

[79] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 328–339, New York, NY, USA, 1995. ACM.

[80] Nils Grimsmo. Faster path indexes for search in xml data. In *ADC '08: Proceedings of the nineteenth conference on Australasian database*, pages 127–135, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[81] Sven Groppe and Stefan Böttcher. Schema-based query optimization for xquery queries. In *ADBIS Research Communications*, 2005.

[82] Torsten Grust. Accelerating XPath location steps. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 109–120, New York, NY, USA, 2002. ACM.

[83] Torsten Grust, Maurice Van Keulen, and Jens Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Trans. Database Syst.*, 29(1):91–131, 2004.

[84] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase join: teach a relational dbms to watch its (axis) steps. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 524–535. VLDB Endowment, 2003.

[85] Martin Gudgin, Noah Mendelsohn, Mark Nottingham, and Hervé Ruellan (ed.). XML-binary optimized packaging. http://www.w3.org/TR/xop10/.

[86] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 157–166, New York, NY, USA, 1993. ACM.

[87] Ashish Kumar Gupta and Dan Suciu. Stream processing of XPath queries with predicates. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 419–430, New York, NY, USA, 2003. ACM.

[88] Beda Christoph Hammerschmidt, Christian Werner, Ylva Brandt andVolker Linnemann, Sven Groppe, and Stefan Fischer. Incremental validation of string-based XML data in databases, file systems, and streams. In *ADBIS*, pages 314–329, 2007.

[89] AgileDelta Inc. Theory, benefits and requirements for efficient encoding of XML documents. http://www.agiledelta.com/EfficientXMLEncoding.htm.

[90] International Characters Inc. Parabix project. http://parabix.costar.sfu.ca/.

[91] Zachary G. Ives, A. Y. Halevy, and D. S. Weld. An XML query engine for network-bound data. *The VLDB Journal*, 11(4):380–402, 2002.

[92] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-tree: indexing XML data for efficient structural joins. In *Proceedings of the 19th International Conference on Data Engineering*, pages 253–264, March 2003.

[93] Vanja Josifovski, Marcus Fontoura, and Attila Barta. Querying XML streams. *The VLDB Journal*, 14(2):197–210, 2005.

[94] Bintou Kane, Hong Su, and Elke A. Rundensteiner. Consistently updating XML documents using incremental constraint check queries. In *WIDM '02: Proceedings of the 4th international workshop on Web information and data management*, pages 1–8, New York, NY, USA, 2002. ACM.

[95] Jaakko Kangasharju, Sasu Tarkoma, and Tancred Lindholm. Xebu: A binary format with schema-based optimizations for XML data. In *WISE*, pages 528–535, 2005.

[96] Christoph Koch. Efficient processing of expressive node-selecting queries on XML data in secondary storage: a tree automata-based approach. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 249–260. VLDB Endowment, 2003.

[97] Margaret G. Kostoulas, Morris Matsa, Noah Mendelsohn, Eric Perkins, Abraham Heifets, and Martha Mercaldi. Xml screamer: an integrated approach to high performance xml parsing, validation and deserialization. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 93–102, New York, NY, USA, 2006. ACM.

[98] Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Visibly pushdown automata for streaming xml. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 1053–1062, New York, NY, USA, 2007. ACM.

[99] April Kwong and Michael Gertz. Schema-based optimization of XPath expressions. Technical report, Department of Computer Science, University of California, Davis, CA, 2002.

[100] Yongnian Le. Schema validation with Intel streaming SIMD extensions 4 (Intel SSE4). White Paper, Intel Corporation, April 2008. http://softwarecommunity.intel.com/isn/downloads/intelavx/Schema Validation w Intel SSE4_WP.pdf.

[101] Christopher League and Kenjone Eng. Schema-based compression of XML data with Relax NG. *Journal of Computers*, 2(10):9–17, 2007.

[102] Daewook Lee, Joonho Kwon, Weidong Yang, Hyoseop Shin, Jae min Kwak, and Sukho Lee. Schema-aware XPath filtering on XML document streams. *Journal of Intelligent Manufacturing*, 20(3):273–282, 2009.

[103] Daewook Lee, Hyoseop Shin, Joonho Kwon, Weidong Yang, and Sukho Lee. SFilter: Schema based filtering system for XML streams. In *Multimedia and Ubiquitous Engineering, 2007. MUE '07. International Conference on*, pages 266–271, April 2007.

[104] Zhai Lei. XML parsing accelerator with Intel streaming SIMD extensions 4 (Intel SSE4). White Paper, Intel Corporation, April 2008. http://softwarecommunity.intel.com/isn/downloads/intelavx/Intel XML Parsing Accelerator w Intel SSE4_WP.pdf.

[105] Gregory Leighton, Jim Diamond, and Tomasz Muldner. AXECHOP: A grammar-based compressor for XML. In *DCC '05: Proceedings of the Data Compression Conference*, pages 467–467, Washington, DC, USA, 2005. IEEE Computer Society.

[106] Gregory Leighton, Tomasz Müldner, and James Diamond. TREECHOP: A tree-based queryable compressor for XML. Technical report, Jodrey School of Computer Science, Acadia University, 2005.

[107] Stefan Letz, Roland Seiffert, JAn van Lunteren, and Paul Herrmann. System architecture for XML offload to a Cell processor-based workstation. In *Proceedings of XML 2005*, 2005.

[108] Stefan Letz, Michel Zedler, Tobias Thierer, Matthias Schütz, Jochen Roth, and Roland Seiffert. XML offload and acceleration with Cell broadband engine. In *Proceedings of XTech 2006*, 2006.

[109] Mark Levene and Peter Wood. XML structure compression. In *In Proc. 2nd Int. Workshop on Web Dynamics*, 2002.

[110] Michael Leventhal. Random access XML programming assisted with XML hardware. In *XML 2004*, Washington, D.C., U.S.A, November 15-19 2004.

[111] Xiaogang Li and Gagan Agrawal. Parallelizing xquery in a cluster environment. In *IDEAS*, pages 291–294, 2006.

[112] Hartmut Liefke and Susan B. Davidson. View maintenance for hierarchical semistructured data. In *DaWaK 2000: Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery*, pages 114–125, London, UK, 2000. Springer-Verlag.

[113] Hartmut Liefke and Dan Suciu. XMill: an efficient compressor for XML data. *SIGMOD Rec.*, 29(2):153–164, 2000.

[114] Yongjing Lin, Youtao Zhang, Quanzhong Li, and Jun Yang. Supporting efficient query processing on compressed XML files. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 660–665, New York, NY, USA, 2005. ACM.

[115] Le Liu, Jianhua Feng, Guoliang Li, Qian Qian, and Jianhui Li. Parallel structural join algorithm on shared-memory multi-core systems. In *The Ninth International Conference on Web-Age Information Management (WAIM'08)*, pages 70–77, 2008.

[116] Jean loup Gailly and Mark Adler. Gzip compressor. http://www.gzip.org.

[117] W. Lowe, M.L. Noga, and T. Gaul. Foundations of fast communication via XML. *Annals of Software Engineering*, 13:357–379, 2002.

[118] Wei Lu, Kenneth Chiu, and Yinfei Pan. A parallel approach to XML parsing. In *Proceedings of The 7th IEEE/ACM International Conference on Grid Computing (Grid2006)*, Barcelona, Spain, September 2006.

[119] Wei Lu and Dennis Gannon. Parallel XML processing by work stealing. In *SOCP '07: Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*, pages 31–38, New York, NY, USA, 2007. ACM Press.

[120] Wei Lu and Dennis Gannon. ParaXML: a parallel XML processing model on multicore CPUs. Technical Report TR662, Indiana University, Bloomington, Indiana, USA, 2008.

[121] Imam Machdi, Toshiyuki Amagasa, and Hiroyuki Kitagawa. Gmx: an xml data partitioning scheme for holistic twig joins. In *iiWAS '08: Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*, pages 137–146, New York, NY, USA, 2008. ACM.

[122] Imam Machdi, Toshiyuki Amagasa, and Hiroyuki Kitagawa. XML data partitioning strategies to improve parallelism in parallel holistic twig joins. In *ICUIMC '09: Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication*, pages 471–480, New York, NY, USA, 2009. ACM.

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

[124] Gerard Marks and Mark Roantree. Pattern based processing of XPath queries. In *IDEAS '08: Proceedings of the 2008 international symposium on Database engineering & applications*, pages 179–188, New York, NY, USA, 2008. ACM.

[125] Morris Matsa, Eric Perkins, Abraham Heifets, Margaret Gaitatzes Kostoulas, Daniel Silva, Noah Mendelsohn, and Michelle Leger. A high-performance interpretive approach to schema-directed parsing. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 1093–1102, New York, NY, USA, 2007. ACM.

[126] Norman May, Matthias Brantner, Alexander Böhm, Carl-Christian Kanne, and Guido Moerkotte. Index vs. navigation in xpath evaluation. In *XSym'06: Database and XML Technologies, Proceedings of the 4th International XML Database Symposium*, pages 16–30, 2006.

[127] Jason McHugh and Jennifer Widom. Query optimization for XML. In *VLDB*, pages 315–326, 1999.

[128] David Megginson and David Brownell. The simple API for XML. http://www.saxproject.org/.

[129] Gerome Miklau and Dan Suciu. Containment and equivalence for an XPath fragment. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 65–76, New York, NY, USA, 2002. ACM.

[130] Jun-Ki Min, Myung-Jae Park, and Chin-Wan Chung. XPRESS: a queriable compression for XML data. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 122–133, New York, NY, USA, 2003. ACM.

[131] Yasuhiko Minamide and Akihiko Tozawa. XML validation for context-free grammars. In *APLAS*, pages 357–373, 2006.

[132] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, 2001.

[133] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Interet Technol.*, 5(4):660–704, 2005.

[134] Benedek Nagy. Permutation languages in formal linguistics. In *IWANN '09: Proceedings of the 10th International Work-Conference on Artificial Neural Networks*, pages 504–511, Berlin, Heidelberg, 2009. Springer-Verlag.

[135] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3), 2002.

[136] Wilfred Ng, Wai-Yeung Lam, Peter T. Wood, and Mark Levene. XCQ: A queriable XML compression system. *Knowl. Inf. Syst.*, 10(4):421–452, 2006.

[137] Matthias Nicola and Jasmi John. XML parsing: a threat to database performance. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 175–178, New York, NY, USA, 2003. ACM.

[138] D. Olteanu, T. Kiesling, and F. Bry. An evaluation of regular path expressions with qualifiers against XML streams. In *Proceedings. 19th International Conference on Data Engineering*, pages 702–704, March 2003.

[139] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking forward. In *EDBT '02: Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers*, pages 109–127, London, UK, 2002. Springer-Verlag.

[140] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. Ordpaths: insert-friendly XML node labels. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 903–908, New York, NY, USA, 2004. ACM.

[141] Makoto Onizuka. Light-weight XPath processing of XML stream with deterministic automata. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 342–349, New York, NY, USA, 2003. ACM.

[142] Luca Padovani and Stefano Zacchiroli. Stream processing of XML documents made easy with LALR(1) parser generators. Technical report, University of Bologna, 2007.

[143] Yinfei Pan, Wei Lu, Ying Zhang, and Kenneth Chiu. A static load-balancing scheme for parallel XML parsing on multicore CPUs. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 351–362, Washington, DC, USA, 2007. IEEE Computer Society.

[144] Yinfei Pan, Ying Zhang, Kenneth Chiu, and Wei Lu. Parallel XML parsing using Meta-DFAs. In *E-SCIENCE '07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing*, pages 237–244, Washington, DC, USA, 2007. IEEE Computer Society.

[145] Yannis Papakonstantinou and Vasilis Vassalos. Query rewriting for semistructured data. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 455–466, New York, NY, USA, 1999. ACM.

[146] Stelios Paparizos, Jignesh M. Patel, and H. V. Jagadish. Sigopt: Using schema to optimize XML query processing. In *ICDE*, pages 1456–1460, 2007.

[147] Stelios Paparizos, Yuqing Wu, Laks V. S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 71–82, New York, NY, USA, 2004. ACM.

[148] Feng Peng and Sudarshan S. Chawathe. XPath queries on streaming data. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 431–442, New York, NY, USA, 2003. ACM.

[149] Feng Peng and Sudarshan S. Chawathe. XSQ: A streaming XPath engine. *ACM Trans. Database Syst.*, 30(2):577–623, 2005.

[150] João Pereira, Françoise Fabret, Hans-Arno Jacobsen, François Llirbat, and Dennis Shasha. Webfilter: A high-throughput xml-based publish and subscribe system. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 723–724, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[151] Eric Perkins, Morris Matsa, Margaret G. Kostoulas, Abraham Heifets, and Noah Mendelsohn. Generation of efficient parsers through direct compilation of xml schema grammars. *IBM Syst. J.*, 45(2):225–244, 2006.

[152] Dac Pham, Hans-Werner Anderson, Erwin Behnen, Mark Bolliger, Sanjay Gupta, Peter Hofstee, Paul Harvey, Charles Johns, Jim Kahle, Atsushi Kameyama, John Keaty, Bob Le, Sang Lee, Tuyen Nguyen, John Petrovick, Mydung Pham, Juergen Pille, Stephen Posluszny, Mack Riley, Joseph Verock, James Warnock, Steve Weitzel, and Dieter Wendel. Key features of the design methodology enabling a multi-core soc implementation of a first-generation cell processor. In *ASP-DAC '06: Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 871–878, Piscataway, NJ, USA, 2006. IEEE Press.

[153] Luping Quan, Li Chen, and Elke A. Rudensteiner. Argos: Efficient refresh in an xql-based web caching system. In *Selected papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases*, pages 78–91, London, UK, 2001. Springer-Verlag.

[154] Mukund Raghavachari and Oded Shmueli. Efficient revalidation of XML documents. *IEEE Trans. on Knowl. and Data Eng.*, 19(4):554–567, 2007.

[155] F. Reuter. Cardinality automata: A core technology for efficient schema-aware parsers, 2003. http://www.swarms.de/publications/cca.pdf.

[156] Daniel J. Rosenkrantz. Programmed grammars and classes of formal languages. *J. ACM*, 16(1):107–131, 1969.

[157] David Salomon. *Data Compression: The Complete Reference*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[158] Arsany Sawires, Junichi Tatemura, Oliver Po, Divyakant Agrawal, and K. SelÇuk Candan. Incremental maintenance of path-expression views. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 443–454, New York, NY, USA, 2005. ACM.

[159] Arsany Sawires, Junichi Tatemura, Oliver Po, Divyakant Agrawal, Amr El Abbadi, and K. Selçuk Candan. Maintaining XPath views in loosely coupled systems. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 583–594. VLDB Endowment, 2006.

[160] Albrecht Schmidt. XMark – an XML benchmark project. http://www.xml-benchmark.org/.

[161] Luc Segoufin and Victor Vianu. Validating streaming xml documents. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 53–64, New York, NY, USA, 2002. ACM.

[162] Julian Seward. Bzip compressor. http://www.bzip.org.

[163] Scott Tyler Shafer. Datapower delivers XML acceleration device, 2002. http://www.infoworld.com/articles/hn/xml/02/08/27/020827hndatapower.html.

[164] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 302–314, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[165] Panu Silvasti, Seppo Sippu, and Eljas Soisalon-Soininen. XML-document-filtering automaton. *Proc. VLDB Endow.*, 1(2):1666–1671, 2008.

[166] Panu Silvasti, Seppo Sippu, and Eljas Soisalon-Soininen. Schema-conscious filtering of XML documents. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 970–981, New York, NY, USA, 2009. ACM.

[167] Przemyslaw Skibinski. XWRT compressor. http://sourceforge.net/projects/xwrt.

[168] Przemyslaw Skibinski and Jakub Swacha. Combining efficient XML compression with query processing. In *ADBIS*, pages 330–342, 2007.

[169] SourceForge.net. http://expat.sourceforge.net.

[170] Hariharan Subramanian and Priti Shankar. Compressing xml documents using recursive finite state automata. In *Implementation and Application of Automata, ser. LNCS*, pages 282–293. Springer, 2006.

[171] Sun Microsystems. Fast Infoset Project. Available from https://fi.dev.java.net/.

[172] Toyotaro Suzumura, Satoshi Makino, and Naohiko Uramoto. Optimizing differential XML processing by leveraging schema and statistics. In Asit Dan and Winfried Lamersdorf, editors, *ICSOC*, volume 4294 of *Lecture Notes in Computer Science*, pages 264–276. Springer, 2006.

[173] Toyotaro Suzumura, Toshiro Takase, and Michiaki Tatsubori. Optimizing Web services performance by differential deserialization. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, pages 185–192, Washington, DC, USA, 2005. IEEE Computer Society.

[174] Code Synthesis. Codesynthesis xsd. http://www.codesynthesis.com/products/xsd/download.xhtml.

[175] Toshiro Takase, Hisashi Miyashita, Toyotaro Suzumura, and Michiaki Tatsubori. An adaptive, fast, and safe XML parser based on byte sequences memorization. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 692–701, New York, NY, USA, 2005. ACM.

[176] Hajime Takekawa and Hiroshi Ishikawa. Incrementally-updatable stream processors for XPath queries based on merging automata via ordered hash-keys. In *DEXA '07: Proceedings of the 18th International Conference on Database and Expert Systems Applications*, pages 40–44, Washington, DC, USA, 2007. IEEE Computer Society.

[177] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 204–215, New York, NY, USA, 2002. ACM.

[178] Lee Thomason. TinyXML, June 2011. http://sourceforge.net/projects/tinyxml/.

[179] Henry S. Thompson and Richard Tobin. Using finite state automata to implement W3C XML schema contentmodel validation and restriction checking. In *Proceedings of XML Europe*, 2003.

[180] P.M. Tolani and J.R. Haritsa. XGrind: a query-friendly XML compressor. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 225–234, 2002.

[181] R.A. van Engelen and K.A. Gallivan. The gSOAP toolkit for web services and peer-to-peer computing networks. In *proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid*, pages 128–135, Berlin, Germany, May 2002.

[182] R.A. van Engelen, K.A. Gallivan, and B. Walsh. Tight timing estimation with the Newton-Gregory formulae. In *proceedings of CPC 2003*, pages 321–330, Amsterdam, Netherlands, January 2003.

[183] Robert van Engelen. The gSOAP toolkit for C and C++ Web services. Available from http://gsoap2.sourceforge.net.

[184] Robert van Engelen. Constructing finite state automata for high performance XML Web services. In *proceedings of the International Symposium on Web Services (ISWS)*, 2004.

[185] Robert van Engelena, Madhusudhan Govindaraju, and Wei Zhang. Exploring remote object coherence in XML web services. In *Proceedings of IEEE International Conference on Web Services (ICWS)*, pages 249–256, Chicargo, IL, USA, September 18-22 2006.

[186] Jan van Lunteren. High-performance pattern-matching for intrusion detection. In *INFOCOM'06: Proceedings of the 25th IEEE International Conference on Computer Communications*, pages 1–13, 2006.

[187] Jan van Lunteren, Ton Engbersen, Joe Bostian, Bill Carey, and Chris Larsson. XML accelerator engine. In *Proceedings of The First International Workshop on High Performance XML Processing*, 2004.

[188] D. Veillard. Libxml2 project web page, 2004. http://xmlsoft.org.

[189] W3C. Document object model (DOM). http://www.w3.org/DOM/.

[190] W3c. Efficient XML interchange (EXI) format 1.0. http://www.w3.org/XML/EXI/.

[191] W3C. Namespaces in XML specification. Available from http://www.w3.org/TR/xml-names/.

[192] W3C. Wap binary xml content format. Available from http://www.w3.org/TR/wbxml/.

[193] W3C. XML XPath specification. http://www.w3.org/TR/xpath.

[194] W3C. XML information set (second edition), 2003. http://www.w3.org/TR/xml-infoset.

[195] W3C. XML schema, W3C recommendation, October 2004. Part1:http://www/w3c.org/TR/xmlschema-1/, Part2:http://www/w3c.org/TR/xmlschema-2/, Primer:http://www/w3c.org/TR/xmlschema-0/.

[196] W3C. Efficient xml interchange measurements note, May 2007. http://www.w3.org/TR/2007/WD-exi-measurements-20070725/.

[197] W3C. An xml query language, 2007. http://www.w3.org/TR/xquery/.

[198] Guoren Wang, , Guoren Wang, Mengchi Liu, Bing Sun, Ge Yu, and Jianhua Lv. Effective schema-based XML query optimization techniques. In *Proceedings of IDEAS*, pages 230–235. IEEE Computer Society, 2003.

[199] Hongzhi Wang, Jianzhong Li, Jizhou Luo, and Zhenying He. XCpaqs: Compression of XML document with XPath query support. In *ITCC '04: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 2*, page 354, Washington, DC, USA, 2004. IEEE Computer Society.

[200] Felix Weigel, Klaus U. Schulz, and Holger Meuss. The bird numbering scheme for xml and tree databases - deciding and reconstructing tree relations using efficient arithmetic operations. In *Proceedings of The International XML Database Symposium (XSym)*, pages 49–67, 2005.

[201] R. Whitmer. Document object model (dom) level 3 xpath specification version 1.0. Technical report, W3C Working Draft, March 2002. http://www.w3.org/TR/DOM-Level- 3-XPath.

[202] R. Whitmer. Document Object Model (DOM) level 3 XPath specification version 1.0. Technical report, W3C Working Draft, March 2002. http://www.w3.org/TR/DOM-Level-3-XPath.

[203] Stephen D. Williams. The esXML data format. 2006. http://esxml.org/esxml-specification.html.

[204] Peter T. Wood. Containment for XPath fragments under dtd constraints. In *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, pages 300–314, London, UK, 2002. Springer-Verlag.

[205] François Yergeau, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. XML 1.0: Extensible markup language (XML) 1.0 (third edition), w3c recommendation, October 2000. http://www.w3.org/TR/REC-xml.

[206] Jimmy Zhang. Non-extractive parsing for xml, May 2004. http://www.xml.com/pub/a/2004/05/19/parsing.html.

[207] Jimmy Zhang. Virtual Token Descriptor (VTD) XML Parser, February 2011. http://vtd-xml.sourceforge.net/.

[208] Wei Zhang and Robert van Engelen. A table-driven streaming XML parsing methodology for high-performance Web services. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, pages 197–204, Washington, DC, USA, 2006. IEEE Computer Society.

[209] Wei Zhang and Robert A. van Engelen. TDX: a high-performance table-driven XML parser. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 726–731, New York, NY, USA, 2006. ACM.

[210] Wei Zhang and Robert A. van Engelen. An adaptive XML parser for developing high-performance Web services. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 672–679, Washington, DC, USA, 2008. IEEE Computer Society.

[211] Wei Zhang and Robert A. van Engelen. High-performance XML parsing and validation with permutation phrase grammar parsers. In *ICWS '08: Proceedings of the 2008 IEEE International Conference on Web Services*, pages 286–294, Washington, DC, USA, 2008. IEEE Computer Society.

[212] Yue Zhuge and Hector Garcia-Molina. Graph structured views and their incremental maintenance. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pages 116–125, Washington, DC, USA, 1998. IEEE Computer Society.

BIOGRAPHICAL SKETCH

Wei Zhang graduated with a B.E. in Electrical Engineering from Chengdu University of Information Technologies in 1991. After graduation, he worked for Tianjin Meterology Bureau and Chinese Sciences and Technologies Network as a networks and systems administrator and software engineer before returning to school in the fall of 2002 to pursue an advanced degree in Computer Science. After graduated with an M.S. in Computer Science in the Summer of 2004, he joined in Florida State University to pursue a Ph.D, in computer Science at Florida State University. Wei is an seasoned software engineer and an expert in XML processing stacks and Web services research. In addition to his work as a student, he also has significant experience in industries.