Set-at-a-time Access to XML through DOM

Hai Chen Frank Wm. Tompa
School of Computer Science
University of Waterloo
Waterloo,ON,Canada
+1-519-888-4567
{h24chen,fwtompa}@db.uwaterloo.ca

ABSTRACT

To support the rapid growth of the web and e-commerce, W3C developed DOM as an application programming interface that provides the abstract, logical tree structure of an XML document. In this paper, we propose ordered-set-at-a-time extensions for DOM while maintaining its tightly managed navigational nature. In particular, we define the NodeSequence interface with functions that filter, navigate, and transform sequences of nodes simultaneously. The extended DOM greatly simplifies writing some application code, and it can reduce the communications overhead and response time between a client application and the DOM server to provide applications with more efficient processing. As validation of our proposals, we present application examples that compare the convenience and efficiency of DOM with and without extensions.

Categories and Subject Descriptors

I.7.1 [**Document and Text Processing**]: Document and text editing– *document management, languages, DOM*; H.2.4 [**Database Management**]: Systems– *textual databases, DOM.*

General Terms

Design, Standardization, Languages.

Keywords

DOM, set-at-a-time, navigation, XML, application program interface.

1. INTRODUCTION

W3C, the World Wide Web Consortium, proposed XML to facilitate information interchange and integration from heterogeneous systems [14]. XML is a semantics-independent markup language useful for data that may or may not conform to a rigid and predefined schema. It is being developed by industry as a universal data representation format. Two APIs (application programming interfaces) are commonly used to process XML documents: SAX (Simple API for XML) and DOM (Document Object Model). A SAX parser provides a simple interface that does not pass information to applications about the nesting in a document [5]. On the other hand, the DOM interface represents a parsed XML document as an abstract tree structure that consists of objects [11]. As a result DOM trees can be directly stored in an object database, and an application can access the XML data in any order, update the content as needed, and restructure the document via this interface.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DocEng'03, November 20–22, 2003, Grenoble, France.
Copyright 2003 ACM 1-58113-724-9/03/0011…$5.00.

DOM provides access to XML data through a set of interfaces that allows an application to traverse the DOM tree beginning from the document root. For example, by calling various methods an application can move from any node to its children, parent, or immediate siblings, one step at a time. However, although DOM can identify one node's children as a NodeList through a single method call, it cannot get all those children's children in one additional operation. Therefore the application programmer must write a loop to process each node in turn. For example, assume an application is using a catalog illustrated as a simplified DOM tree in Figure 1. To obtain a list of all book authors, it could invoke C++ code as follows (using the Apache Recommended C++ Language bindings [1]):

```
DOM_Document doc = parser.getDocument();
DOM_NodeList books = doc.
                     getElementsByTagName("book");
unsigned int numBooks = books.getLength();
DOM_Node bookAuthors [MAX_NUM];
int k=0;
for (int i=0; i<numBooks; i++)
{ DOM_Node authors = books.item(i).
                    getElementsByTagName("author");
  unsigned int numAuthors = authors.getLength();
  for (int j=0; j<numAuthors; j++)
    bookAuthors[k++] = authors.item(j);
}
```

In this example, a list of all books can be retrieved in one operation, but each book's author list and subsequently each individual author must be retrieved by looping. (A nested loop is also required for the complementary approach: retrieve all author elements directly and then check the tag name of each parent for a match to "book.") Furthermore, for an application to produce robust code, checks must also be included to ensure that the result array size is sufficiently large. Such a situation often arises, where

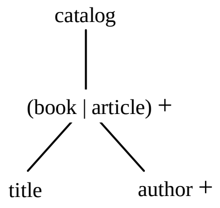


Figure 1. Element relationship in simple structure

we need to manipulate all elements of a node collection in a similar loop, and it would be useful to express this as a single operation. In this paper we provide some extensions to DOM for operating on collections of nodes.

The idea to support operations for sequences of nodes comes from other environments in which set-at-a-time data access has been shown to be worthwhile. The best-known examples include relational and object-relational database systems (as opposed to earlier, navigational hierarchic and network systems) and array-processing languages such as APL (as opposed to earlier languages such as Fortran and Cobol). These systems support operations for collections of data that do not require a user to iterate through the collection element by element.

For XML, W3C produced the XQuery language [16], which is based on processing collections of nodes and includes subtree matching and extractions through XPath [15] and operations to construct and combine node sequences [17]. XQuery is designed for applications that need complex extractions from large XML collections, and it requires that the data be fully validated using XML Schema. DOM, on the other hand, is tailored for applications that maintain (query, manipulate, and update) individual documents, whether or not they have an associated schema. Whereas W3C is considering a candidate recommendation to support XPath from within the DOM framework [13], operations such as concatenate, distinct, insert, sublist, intersect, except, filter, and sort are not provided in the DOM interface.

For applications that use DOM, we propose to add new light-weight functions that operate on simple sequences. Obvious advantages of such a solution are flexible application, ease of use, efficiency, and tight integration with the application environment.

After defining the extensions in Sections 2 and 3, we present the results of experiments and analysis in Section 4 to demonstrate the efficiency gains from this approach.

2. DESIGNING EXTENSIONS TO DOM

2.1 Set-at-a-time Processing

The Core DOM interface provides an application with a sequence of child nodes for a given parent through the NodeList interface, which provides only two limited operations: the attribute *length* holds the number of nodes in a list, and the method *item(i)* returns the *i*th item in a list [11]. This idea is extended in DOM Level 3 to a sequence of names and a sequence of strings through the NameList and DOMStringList interfaces, respectively [12]. The DOM Level 3 XPath specification includes the XPathResult interface, which provides a sequence of matched nodes through an iterator [13]. However, there is no interface in DOM that provides a rich set of operators on an application-defined collection of nodes.

Our proposal defines the *NodeSequence interface* in order to provide two types of methods based on the Core's NodeList interface:

- mapping Node operations to every node in a NodeSequence: These methods operate on each node in the NodeSequence <n_1, n_2, … ,n_k> and produce a sequence of nodes <f(n_1), f(n_2), … ,f(n_k)>. For example, *getParents()* extends the attribute *parentNode* for one node to achieve that effect for each node in a given collection.
- manipulating the NodeSequence itself: Such methods include concatenating two sequences of nodes, sorting a sequence of nodes, extracting a subsequence of nodes, transforming a sequence of nodes, eliminating duplicates, and so on.

The NodeSequence interface is described more fully in Section 3.

2.2 Detailed Design Considerations

In designing extensions to a language or API, it is important to respect its modality. For DOM, this means close coupling of an application with document access, so that "programmers can build documents, navigate their structure, and add, modify, or delete elements and content" [11]. In fact, a major reason for the paucity of operations on a NodeList is so that it can be "*live*, that is, changes to the underlying document structure are reflected in all relevant NodeList … objects."

It is intended that the NodeSequence interface provide applications with more convenience at the cost of sacrificing liveness. However we need to strike a balance between the convenience provided to an application when operators are high level and potentially highly optimizable (typical of XQuery, for example) and the close integration provided by the DOM Core.

2.2.1 Include Copies or References

In the DOM Core, navigation is restricted to one node at a time. It is left to the application to determine whether to copy this node (e.g., via *cloneNode*) or to reference the original (via a reference or pointer). In the NodeSequence interface, a collection of nodes is returned at one time. When an operation concatenates two NodeSequences, for example, should the new sequence contain copies or references to the nodes in the arguments? As explained below, it is usually preferable to return references.

- Access to context

Duplicate nodes as defined by *cloneNode* have no parents. Thus, access to their ancestors or siblings is not preserved.

- Updatability

A collection of references to nodes can be used as a handle to manipulate those nodes. For example, an application that wishes to delete the referenced nodes from the DOM tree can remove them directly. If, however, the application is given a sequence of copies, it must traverse the DOM tree to locate those nodes that have equal values to the nodes in the sequence. Even so, in the presence of duplicate structures such an application cannot identify which structure is intended to be deleted, even if it uses a deep equality test.

- Storage consumption

When the node size is large, storing references rather than copies will save significant storage.

- Speed

Creating a sequence of node references is efficient and easy to implement. If we instead make a copy of each node, and especially if we choose to make a deep copy, more execution time is required.

- Well-formedness

Of course, if an application inserts a single node or a sequence of nodes into a DOM tree, it must ensure that the nodes are not already present elsewhere in the tree. The simplest approach is to use a copy of these nodes so that the result is guaranteed to be a tree.

2.2.2 Comparison Among Nodes

In the NodeSequence interface, many operations compare collections of nodes. For text and attributes nodes, comparisons based on values are often required by applications. For other types of nodes, such as element nodes, applications typically require object identity. Thus, we provide a parameter to allow applications to choose whether comparisons are to be done by object or by value.

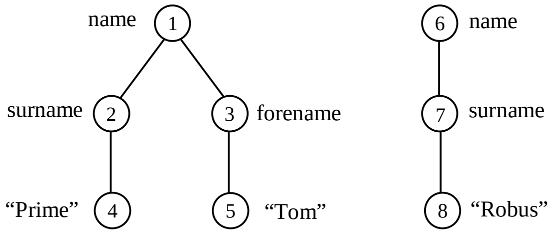
2.2.3 White Space Consideration

White space is often used in well-formed and valid XML documents to set apart markup for better readability and clarity. Although such white space is neither data nor markup, in the absence of validation, XML parsers must process all white space as data. This can occasionally cause surprising results.

For example, a section of an XML document and a corresponding NodeSequence <1,6> may be represented by a simplified DOM tree structure as shown as Figure 2. A naïve use of the method *mapChildNodes* would return a NodeSequence whose length is 8, if it were to mimic the *childNodes* attribute defined for a single node in DOM. Most users, however, would expect the result to have 3 children only, corresponding to nodes 2, 3, and 7. The reason for the unexpected result is that five text nodes containing white space only are (correctly) included as children. To avoid such surprising results and to simplify application programming without resorting to element-at-a-time navigation, a parameter for node type is provided in our proposed method *mapChildNodes*.

```
<name>
    <surname> Prime </surname>
    <forename>Tom </forename>
</name>
<name>
     <surname> Robus </surname>
</name>
```

(a) Document fragment



(b) Structure corresponding to NodeSequence <1,4>

Figure 2. Simplified DOM representation of XML

Thus, a call of *mapChildNodes(ELEMENT_NODE)* will return only nodes of type "element" and therefore the expected result.

3. The NodeSequence Interface

3.1.1 Basic Operations

The NodeSequence interface starts with the same definition as NodeList interface, except that the requirement for liveness is removed. In particular, the attribute *length* holds the number of nodes in a sequence, and the method *item(i)* returns the *i*th item.

3.1.2 NodeSequence constructors

Two constructors are defined: the first converts an individual node into a NodeSequence, and the second converts a NodeList into a NodeSequence. In particular,

```
NodeSequence createNodeSequence (in unsigned short
                                 num);
```

is added to the Node interface. A NodeSequence is constructed with *num* (repeated) references to the object Node.

```
NodeSequence createNodeSequence();
```

is added to the NodeList interface. A NodeSequence is constructed containing the sequence of items included in the object NodeList.

3.1.3 Mapping Node Operations to Sequences

Table 1 summarizes operations of the NodeSequence interface that simply extend attributes and methods of the Node interface.

For Node operations of the form f_i: Node $\rightarrow$ Node (that is, *parentNode*, *firstChild*, *lastChild*, *previousSibling*, *nextSibling* and *cloneNode*), the corresponding newly defined operations for sequences F_i: NodeSequence $\rightarrow$ NodeSequence will include *null* in the resulting sequence wherever the object node is *null* or the returned value corresponding to that object node is *null*. Thus, for example, given the node sequence *s* = <2, null, 3, 6> corresponding to Figure 2(b), *s.mapParentNode*() will return the sequence <1, null, 1, null>. This ensures that the object and resulting NodeSequences are aligned.

On the other hand, for operations that return either a NodeList or NamedNodeMap (that is, *childNodes* and *attributes*) where no alignment is possible, the corresponding resulting NodeSequence will contain the concatenation of the underlying result collections with no *null* values included. Looking again at Figure 2(b) and assuming s contains the sequence <1, null, 3, 6, 8>, *s.mapChildNodes*(*ELEMENT_NODE*) will return the sequence <2, 3, 5, 7>.

For those operations that match nodes in the argument with nodes in the object NodeSequence, namely *mapAppendChild* and *mapRemoveChild*, if the lengths of the two lists are not equal, an exception is thrown.

In consideration of increasing expressitivity for this type of mapped operations, a discussion about including a generalized map operator is included in Section 5 below.

3.1.4 Collection Operations on Sequences

Table 2 summarizes operations that manipulate NodeSequences independently of the values of the nodes in the collection. The first two (*concatenate* and *reshape*) build new sequences from

Table 1. Map Node operations to NodeSequence

`mapParentNode()` ≡ return a NodeSequence referencing each node’s parent

`mapFirstChild (in unsigned short type)` ≡ return a NodeSequence referencing each node’s first child of the specified node type.

`mapLastChild (in unsigned short type)` ≡ return a NodeSequence referencing each node’s last child of the specified node type.

`mapPreviousSibling (in unsigned short type)` ≡ return a NodeSequence referencing the closest preceding node of the specified node type.

`mapNextSibling (in unsigned short type)` ≡ return a NodeSequence referencing the closest following node of the specified node type.

`mapChildNodes (in unsigned short type)` ≡ return all of this NodeSequence’s children having the specified node type.

`mapAttributes()`≡ return all of the attributes of a NodeSequence (as a sequence of nodes).

`mapAppendChild (in NodeSequence children)` ≡ append a copy of the subtree rooted by each node of *children* to the end of the list of children for the corresponding node in the object NodeSequence.

`mapRemoveChild (in NodeSequence children)` ≡ for each node of *children*, remove it from the children of the corresponding node in the object NodeSequence.

`appendChildren(in NodeSequence children)` ≡ append copies of the complete NodeSequence *children* to the list of children of each node in the object NodeSequence.

`removeChildren(in NodeSequence children)` ≡ remove all occurring nodes of *children* from the list of children of each node in the object NodeSequence.

`mapCloneNode (in boolean deep)` ≡ return a NodeSequence in which each node is a new copy by value of the object NodeSequence.

existing ones. Given the node sequence *s* = <2, null, 3, 6>, *s.concatenate(s)* produces the sequence <2, null, 3, 6, 2, null, 3, 6> and *s.reshape(6)* produces the sequence <2, null, 3, 6, 2, null>. The next five functions (*subList*, *filterNodeType*, *filterTagName*, *filterValue*, and *filterNonNull*) extract subsequences based on position, type, tag, value, and whether or not the value is null, respectively. *Sort* and *distinct* are complementary operations commonly provided in database languages. The final two operations (*subtract* and *intersect*) are adapted from set operators to work on sequences. (Note that the effect of *union* can be achieved by applying *concatenate* followed by *distinct*.)

3.1.5 Example

Consider again the example given in Section 1. Obviously, much of the code when using the DOM Core is for looping over a collection of nodes, which is simplified with the NodeSequence interface, as follows:

Table 2. Manipulate NodeSequences

`concatenate (in NodeSequence nodes)` ≡ return the concatenation of all members of the object NodeSequence with all members of *nodes* appended.

`reshape (in unsigned long num)` ≡ generate a NodeSequence equal to the first *num* members of the repeated concatenation of the object NodeSequence with itself. Adopted from APL.

`subList (in unsigned long startindex, in unsigned long length)` ≡ return a section of this NodeSequence beginning at position *startindex* and ending at *startindex*+*length*-1.

`filterNodeType (in unsigned short type)` ≡ return the subsequence of nodes matching *type*.

`filterTagName (in string tag)` ≡ return the subsequence of element nodes having tag name *tag*.

`filterValue (in string value)` ≡ return the subsequence of text nodes matching the string *value*.

`filterNonNull ()` ≡ return the subsequence of non-null nodes.

`distinct (in boolean byValue)` ≡ return a NodeSequence that retains the first copy of each node in the object NodeSequence in order. If *byValue* is true, eliminate duplicate text nodes by value, otherwise eliminate duplicates based on reference (i.e., object identity).

`sort (in boolean order)` ≡ return the sequence of nodes in document order if *order* is true or reverse document order if false, preserving duplicates. Nodes not in the document (for example, cloned nodes not added to the DOM tree), are placed at the end in their original order in the object NodeSequence.

`subtract (in NodeSequence nodes, in boolean byValue)` ≡ return a NodeSequence containing the nodes in the object NodeSequence, but not in the argument *nodes*. If *byValue* is true, text nodes are compared by value.

`intersect (in NodeSequence nodes, in boolean byValue)` ≡ return a NodeSequence containing the nodes in the object NodeSequence if they also appear in *nodes*. If *byValue* is true, text nodes are compared by value.

```
DOM_Document doc = parser.getDocument();
DOM_NodeList books = doc.
                     getElementsByTagName("book");
DOM_NodeSequence names = books.
                  createNodeSequence().
                  mapChildNodes(ELEMENT_NODE).
                  filterTagName("author");
```

Consider next the communication messages exchanged between the client and the DOM server. Let *m* be the number of books and *n* be the number of authors in the catalog. Using the DOM Core, the number of method invocations, each of which might require a message sent to the server and a reply back to the client, is *2m+n+3*. Using NodeSequence, the number of messages is always constant, *5*. Therefore, the extensions also reduce the communications overhead and response time between a client and a DOM server.

4. EXPERIMENTS AND ANALYSIS

4.1 Prototype System Environment

The prototype system supports a server-client architecture (Figure 3) in which the executable program comprising the extended DOM and all XML documents are installed on a server, the applications are executed in the client, and each time the client calls the extended DOM interface, communication messages are exchanged between the client and server. We have implemented the NodeSequence interface on the server by extending the Xerces C++ parser [1]. We expect that the performance gains we observe will also be obtainable by similarly extending other DOM implementations.

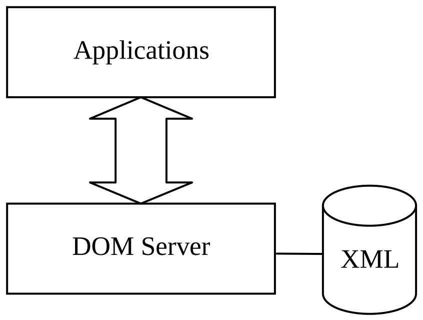


Figure 3. Target architecture

4.2 XML Sample Data

With the extensive usage of XML in various application domains, several XML benchmarks have been created to measure, evaluate and optimize the performance of proposed approaches to deal with XML documents. Since the broad scope of XML makes it difficult to cover all varieties and characteristics of XML data, each family of XML benchmarks can be used to assess specific classes of applications.

Among the principal benchmarks, the Michigan benchmark is a "micro" benchmark in which the data is designed to test basic query operations [9]. Thus the structure of XML data generated is relatively simple, and each item is only suitable for a few operations. The data generated by XOO7 has few hierarchical element structures [4], also making it unsuitable for testing our extensions. The XMach-1 multi-user benchmark is based on a web application and includes a variety of XML data forms: text documents, schema-less data, and structured data [3]. It provides hierarchical element structures in each kind of XML document but has few cross-references. XBench data is categorized as data-centric and text-centric [18], but again each kind of data has too simple a structure to meet our needs. The XMark benchmark represents an auction application combining text and non-text data for items, persons, open auctions, closed auctions, categories, bidders, sellers, and buyers [10]. The relationships between them are expressed through cross-references.

The XMark benchmark satisfies four characteristics proposed by Gray [7]: application benchmarks should be scalable (applicable to different size of computer systems), portable (available to implement on different systems), simple (credible), and relevant (performing typical operations for the respective domain).

- It provides a unified XML document named "auction.xml" that covers our requirements. The structure of the sub-tree rooted at description elements is similar to that of the XML data in XOO7, and its hierarchical structure is similar to that of the XML data in XMach-1. Its definition in terms of one unified document make it simple to understand, yet varied enough to capture a wide variety of applications.
- It includes a random data generator to create a single document with a size range from several kilobytes to 10Gb. The XML generator for XMark considers the tree fanout, tree depth, and the relationships among different elements through references, which can be found at a variety of locations throughout an XML tree. These make generated documents rich in structure so it can test our extended DOM's ability to query efficiently.

4.3 Application Examples

We use some practical application examples, listed in Table 3, to evaluate our system. For example, application Q9 needs to extract two sublists from the "closed auctions" in order to find their difference. The DOM solution requires either sorting or nested loop comparisons to be coded (Figure 4a), but the application is encoded directly using the proposed extensions (Figure 4b). Note in particular, that the use of the *subtract* method eliminates both the nested loop for comparing sellers to buyers and the nested loop for checking for duplicate sellers.The implementations of all 11 examples using DOM with or without the NodeSequence interface are given in detail elsewhere [6].

4.4 Experiments and Analysis

In addition to making the API simpler, storage space and running time are important measures of performance. The NodeSequence interface does not affect the amount of space needed for the DOM structures themselves. However, it does affect the running time of the entire system, including the processing time on the server and

Table 3. Application examples

Update a collection	Q1: Delete all descriptions of items Q2: Remove open auctions whose initial price is 0
Select a collection based on structure	Q3: Get all intervals in all open auctions Q4: Get names of items
Select a collection based on value	Q5: Count the number of bidders whose id are "person0" Q6: List open auctions whose bidder's id is "person0"
Selection via a join	Q7: Get bidders' name nodes in the private open auctions Q8: Get the number of persons who bid on items from Africa
Difference	Q9: List all ids of persons who are sellers and do not buy other items in closed auctions
Sort	Q10: List in document order open auctions in which all buyers watch
Construct collections	Q11: List the first five and last five closed auctions in document order

```
DOM_Document doc = parser.getDocument();
DOM_Element site = doc.getDocumentElement();
DOM_NodeList closed_auctions = site.getElementsByTagName("closed_auctions").item(0).getChildNodes();
unsigned int closedCount = closed_auctions.getLength();
DOM_Node selected_ids[MAXLENGTH];
unsigned int selectCount=0,i,j;
DOM_Node sellerAttr;
for( i=0; i<closedCount; i++)
 { DOM_Node closedElem = closed_auctions.item(i);
   if (closedElem.getNodeType() == ELEMENT_NODE)   /*  1 represents  */
    { DOM_Node seller=((DOM_Element &)closedElem).getElementsByTagName("seller").item(0);
      sellerAttr = seller.getAttributes().item(0);
      DOMString seller_id = sellerAttr.getNodeValue();
      for (j=0; j<closedCount; j++)
       { DOM_Node closedNode = closed_auctions.item(j);
         if (closedNode.getNodeType() == ELEMENT_NODE)
          { DOM_Node buyer = ((DOM_Element &)(closedNode)).getElementsByTagName("buyer").item(0);
            DOMString buyer_id = buyer.getAttributes().item(0).getNodeValue();
            if (seller_id.equals(buyer_id)) break;
          }
       }
      if (j == closedCount)
       { for (j=0; j<selectCount; j++)
            if (seller_id.equals(selected_ids[j].getNodeValue())) break;
          if (j == selectCount) selected_ids[selectCount++] = sellerAttr;
       }
    }
 }
```

(a) Using DOM without the NodeSequence interface

```
DOM_Document doc = parser.getDocument();
DOM_Element site = doc.getDocumentElement();
DOM_NodeSequence closed_auction = site.getElementbyTagName("closed_auctions").createNodeSequence.
                                 mapChildNodes(ELEMENT_NODE).mapChildNodes(ELEMENT_NODE);
DOM_NodeSequence sellers = closed_auction.filterTagName("seller").mapAttributes().mapChildNodes(0);
DOM_NodeSequence buyers = closed_auction.filterTagName("buyer").mapAttributes().mapChildNodes(0);
DOM_NodeSequence selected_ids = sellers.subtract(buyers,true);
```

(b) Using the NodeSequence interface

Figure 4. Coded solutions for application Q9

client and communication overhead between client and server.

We implemented the extended DOM for a single machine environment (Pentium IV CPU clocked at 2.4GHz with 512Mb of main memory, a 10Gb hard disk, and running Windows2000) and conducted experiments to test the processing time based on the applications in Table 3. Using XMark we generated various sizes of documents from 100Kb to 50Mb to evaluate performance (see Table 4).

Table 4. Generated data sizes and numbers of nodes

Data size	*Open auction nodes*	*Closed auction nodes*	*Item nodes*	*Person nodes*	*Bidder nodes*
116 Kb	12	10	22	25	60
211 Kb	24	19	43	51	81
460 Kb	48	38	86	102	196
906 Kb	96	77	173	204	439
2.45 Mb	240	195	435	510	1140
5.7 Mb	600	489	1087	1275	2989
11.6 Mb	1200	975	2175	2550	6182
23.5 Mb	2400	1950	4350	5100	12097
46.5 Mb	4800	3900	8700	10200	23521

Before running the applications, a generated document is parsed and converted to a tree structure in main memory. The time for this preprocessing is linear in the size of the data (from 78ms for the smallest sample to 22.2s for the largest), and it is unaffected by the introduction of the NodeSequence interface (Figure 5).

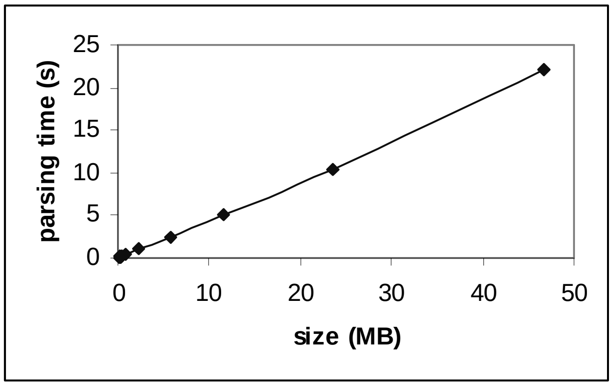


Figure 5. Parsing times

When the size of data is less than 900Kb, the processing time for each example application is less than 1ms whether the implementation is based on the Core or on NodeSequences. Table 5 shows the processing times with the remainder of the data sizes, and, as an example, the times for the 11.6 Mb document are graphed in Figure 6. (For Q10, we do not give the processing time for the Core implementation when the size of data is larger than 2.45Mb because it is unacceptably large as a result of implementation by nested loop joins.) All times include the processing in the server and in the client as well as procedure calls between them, but not any communication delays.

As DOM features main-memory processing and both the server and client are on the same processor, the operations are quite fast when the size of data is under 1Mb. For data sizes over 1Mb, the processing times of the two systems for the first four examples are comparable, with the NodeSequence extension outperforming the Core for Q1 and Q3 for the largest data size only. However, the processing times for the last seven examples are improved with the NodeSequence interface, and the differences increase with the size of XML data and complexity of query. This reflects the earlier observation that implementations based on the Core take time processing the nested loop structures.

In a distributed environment, communication overhead impacts the application performance because of latency inherent in the network bandwidth. Fewer message exchanges between client and server reduce total remote access delays and thus improve system throughput. We compare the number of message pairs required by an application of the DOM to the number required for our extension, assuming that each method call requires a message to be sent and returned.

Table 5. Processing times with various sizes of data

(times in ms except where explicitly marked as seconds)

Size (Mb)		*0.9*	*2.3*	*5.7*	*11.6*	*23.5*	*46.7*
Q1	*Core*	0	31	62	125	282	735
	Seq.	0	31	76	149	309	657
Q2	*Core*	0	15	31	62	141	297
	Seq.	0	24	57	109	225	469
Q3	*Core*	0	16	31	94	171	375
	Seq.	0	16	39	83	167	330
Q4	*Core*	0	8	15	32	78	203
	Seq.	0	16	57	109	219	469
Q5	*Core*	16	31	71	172	375	1000
	Seq.	0	28	68	141	281	578
Q6	*Core*	15	16	63	172	375	968
	Seq.	0	24	73	141	281	580
Q7	*Core*	32	266	2672	10.8s	39.6s	164s
	Seq.	0	31	203	1s	6.1s	28.3s
Q8	*Core*	16	47	203	657	2735	10.4s
	Seq.	0	63	157	367	930	2.8s
Q9	*Core*	32	250	1672	7984	38.7s	225s
	Seq.	0	31	78	281	1.4s	6.2s
Q10	*Core*	2440	150s	—	—	—	—
	Seq.	31	0.1s	0.5s	1.8s	8.3s	39.7s
Q11	*Core*	31	250	1750	7875	38.5s	205s
	Seq.	16	46	218	781	3.3s	14s

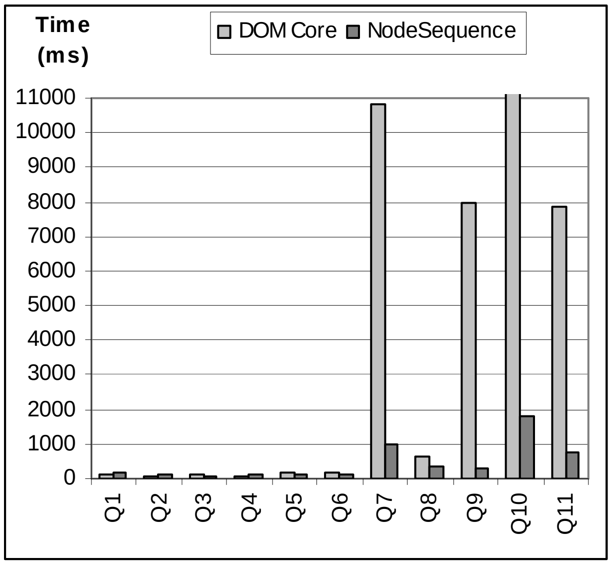


Figure 6. Comparative times for 11.6 Mb document

When using DOM, the communication messages between client and server typically vary with the amount of XML data and the complexity of queries. However, since our extension provides operations on node sequences, applications can manipulate collections of nodes in one operation directly. Thus, communication messages will be reduced sharply. Interestingly, for all of our sample queries, the number of communication messages is independent of the document size.

Table 6 lists a comparative analysis, where let n_1 is the number of *items*, n_2 is the number of *open_auctions*, n_3 is the number of *persons*, n_4 is the number of *closed auctions*, n_5 is the number of *bidders*, and the notation a ··· b denotes lower and upper bounds on the number of messages (depending on the outcomes of various conditional expressions).

Consider the communication messages based on the DOM Core. Since the original DOM provides operations based on one node, when applications need to get node sets with repeated features,

Table 6. Message pairs between client and server

	DOM Core	NodeSequence
Q1	$7n_1+80$	9
Q2	$13n_2+8 \cdots 14n_2+8$	11
Q3	$6n_2+8$	5
Q4	$6n_1+80$	7
Q5	$7n_5 +6n_2+8$	11
Q6	$10+6n_2 \cdots 7n_5 +3n_2+6$	13
Q7	$12+6n_2 \cdots 14n_5n_3+22n_2+14n_5+12$	28
Q8	$O(n_2\ n_1\ n_3)$	34
Q9	$17\ n_4+8 \cdots 10n_4^2+9\ n_4+8$	12
Q10	$O(n_2\ n_4\ n_3)$	28
Q11	$17n_4+8 \cdots 10n_4^2+12n_4+8$	25 ··· 28

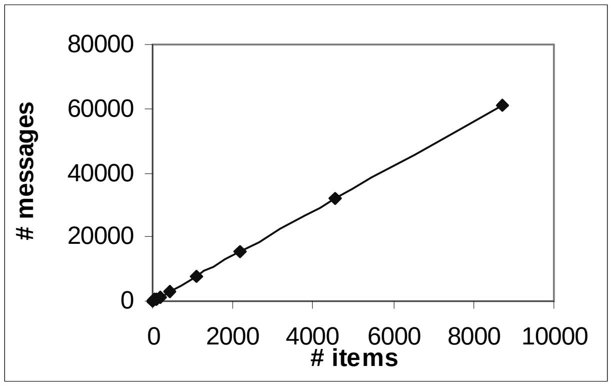


Figure 7. DOM communications messages for Example 1

they have to use loops to process each of these nodes. The first four examples are relatively simple to analyze. Examples Q1 and Q4 contain nested loops, where the outer loop iterates on six *regions*, but seven extra iterations are also needed to process white space. They also scan each *item* node in the inner for loop. Therefore, the number of communication messages is directly proportional to the number of items. For Q1, for example, when updating a 11.4Mb document having 2175 items, 15,305 communication messages are required (Figure 7). Similarly, in Q2 and Q3, the number of communication messages is directly proportional to the number of open auctions.

For examples Q5 and Q6 the number of messages varies with the size of *open_auction* and *item*. For example, for a 460Kb document, 1668 communication messages are required for Q5, which is greater than what is needed for any of the first four examples for a document twice as large. The last five examples require merges or joins on various sub-trees, and thus these applications need to use more loops and handle more nodes. Consequently, many messages have to be conveyed back and forth between the client and the server processes.

Thus we conclude that when the amount of XML data is large, the running time of most applications using the NodeSequence interface will be reduced greatly if *any* DOM implementation is extended with our proposed operators. This benefit is obtained at the same time that our extended DOM greatly simplifies coding and thus is more convenient and less error-prone for application programmers.

5. CONCLUSIONS AND FUTURE WORK

DOM models an XML document as a virtual tree structure. Applications access and update the content and structure of documents dynamically through this platform-neutral and language-neutral interface. Such applications visit objects of the trees through the Node interface and use loops to visit all children of a node. In this paper, we propose the NodeSequence interface to manipulate collections of nodes without requiring node-at-a-time navigation.

The extended DOM provides significant benefits to applications. Application code can be simplified greatly, processing time is reduced, and communication messages between client and server are significantly decreased. Through an implementation and experimentation, we have validated our ideas and demonstrated the resulting gains for a variety of simple applications.

Since our extension only adds a new interface, it does not change any properties of the DOM Core, and thus all of the methods in the original DOM interfaces can still be used. In addition, the extensions fit the same navigational philosophy of DOM, thus making it natural to use them with DOM. This is in marked difference to abandoning DOM in favour of a high-level database language or implementing high-level query facilities on top of (but independently) of DOM [2].

We expect our research work to continue along these lines. In the short term, we wish to define additional functions for the NodeSequence interface to provide more capabilities, as follows.

- Originally, we intended to adopt the *mapcar* operator from LISP and other functional programming languages, where *mapcar*$(f,v_1,...,v_n)$ returns the list obtained by applying the n-ary function f to successive elements taken from vectors v_1 through v_n. In order to avoid requiring explicit support for second-order functions, however, we anticipated passing the name of the function as a string. Thus, for example, using the definition

  ```
      apply (in string functionName)
  ```

  if NodeSequence s = <n_1,...,n_k>, then *s.apply*("getParent") returns the sequence <*getParent*(n_1),...,*getParent*(n_k)>. With such a method, not only would applications be able to map any user-defined functions available for nodes, but the specification of NodeSequence could be simplified by eliminating individual definitions for each built-in function as in Table 1.

  We considered several potential design solutions. Initially, we expected a C++ mapping such as:

  ```
     NodeSequence apply (char *funcname,…);
  ```

  having a variable argument list as the second parameter. However, such a variable list is not a fundamental parameter type in IDL [8], the specification language for DOM.

  An alternative is to define the IDL interface to include:

  ```
  apply (in string *funcname, in string *para)
  ```

  which requires an application to marshal the variable number of arguments into a string before calling *apply* and the server to recover the arguments in their original form. Obviously, this solution incurs some inconvenience and processing cost.

  Another alternative is to define a *family* of functions differing from each other by the number and types of arguments. Three such *apply* functions appear as follows:

  ```
  apply(in string fname)
  apply(in string fname, in unsigned short type)
  apply(in string fname, in NodeList nodes)
  ```

  where the second form would pass the second parameter with each function call and the third form could be defined either to pass the whole NodeList as a parameter to each application of the function or, alternatively like *mapcar*, to pass only the ith node from the NodeList to the corresponding application of the function. This solution is cumbersome to define and will limit the flexibility and scalability of the *apply* function.

Consequently, although the *apply* function could well be useful for the extended DOM interface, more design work is required first.

- Three methods are defined with the capability to compare text nodes based on value. These should be extended to provide value-based comparisons for other types of nodes as well.

```
distinct (bool byValue, unsigned short
         nodetype)

subtract (NodeSequence list, bool byValue,
         unsigned short nodetype);

intersect (NodeSequence list, bool byValue,
          unsigned short nodetype);
```

  In all three cases, the subsequence of nodes of the given type is first selected, and then the operation is applied using equality (perhaps as defined for XQuery) for comparing nodes if *byValue* is true, and object identity otherwise.

- The *sort* function allows applications to reorder node sets by document order only. Many applications may need to order elements by value, especially when text nodes in a node set contain numeric or date values. In addition, an application may again want to operate a specific type of nodes other than text. Consequently, we should modify the method *sort* (*bool order*) to give more functionality:

```
sort (bool order, unsigned short nodetype,
      bool byValue)
```

  This operation will select the subsequence of nodes having type *nodetype*, and sort them into increasing or decreasing order, according to *order*. If the nodetype is "text node" (or some other type with comparators defined) and *byValue* is true, then the nodes are ordered by node value rather than by position.

More importantly, however, communication and processing costs based on a practical system in a distributed environment remains to be tested, and a quantitative analysis of the relationship between communication messages and throughput needs to be evaluated. It is hoped that the resulting system will find applications in areas that require high performance processing of XML documents.

Finally, a change proposal should be submitted to W3C's DOM Working Group to incorporate these extensions into a future level of DOM.

6. ACKNOWLEDGMENTS

Financial support from the Natural Sciences and Engineering Research Council of Canada and the University of Waterloo is gratefully acknowledged.

7. REFERENCES

[1] Apache XML Project. Xerces C++ Parser. http://xml.apache.org/xerces-c/.

[2] Böhm, K. On extending the XML engine with query-processing capabilities. in Proceedings of ADL 2000 (Washington DC, May 2000), IEEE, 127-140.

[3] Böhme, T. and Rahm, E. Multi-user evaluation of XML data management systems with XMach-1. in Proceedings of EEXTT and DIWeb 2002 (Hong Kong, Aug. 2002), LNCS 2590, Springer, 2003, 148-159. http://dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html

[4] Bressan, S., Lee, M.L., Li, Y.G., Lacroix, Z., and Nambiar, U. The XOO7 Benchmark. in Proceedings of EEXTT and DIWeb 2002 (Hong Kong, Aug. 2002), LNCS 2590, Springer, 2003, 146-147. http://www.comp.nus.edu.sg/~ebh/XOO7.html.

[5] Brownell, D. (ed.). SAX. http://www.saxproject.org/.

[6] Chen, H. Supporting Set-at-a-time extensions for XML through DOM, Technical Report CS-2003-27, School of Computer Science, University of Waterloo, 2003, 152 pp. http://www.cs.uwaterloo.ca/cs-archive/CS-2003/CS-2003.shtml

[7] Gray, J. The Benchmark Handbook for Database and Transaction Systems (2nd Edition). Morgan Kaufmann, 1993. http://www.benchmarkresources.com/handbook/.

[8] Object Management Group. Common Object Request Broker Architecture:Core Specification. Version 3.0. Chapter 3: IDL Syntax and Semantics. December 2002. http://www.omg.org/cgi-bin/apps/doc?formal/02-06-39.pdf.

[9] Runapongsa, K., Patel, J.M., Jagadish, H.V., Chen, Y., and Al-Khalifa, S. The Michigan Benchmark: towards XML query performance diagnostics. in Procedings of EEXTT and DIWeb 2002 (Hong Kong, Aug. 2002), LNCS 2590, Springer, 2003, 160-161. http://www.eecs.umich.edu/db/mbench/.

[10] Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., and Busse, R. Assessing XML data management with XMark. in Procedings of EEXTT and DIWeb 2002 (Hong Kong, Aug. 2002), LNCS 2590, Springer, 2003, 144-145. http://monetdb.cwi.nl/xml/.

[11] World Wide Web Consortium. Document Object Model (DOM), Level 2 Core Specification. W3C Recommendation. http://www.w3.org/TR/DOM-Level-2-Core/, 2000.

[12] World Wide Web Consortium. Document Object Model (DOM), Level 3 Core Specification. W3C Working Draft. http://www.w3.org/TR/DOM-Level-3-Core/, 2003.

[13] World Wide Web Consortium. Document Object Model (DOM) Level 3 XPath Specification. W3C Candidate Recommendation. http://www.w3.org/TR/DOM-Level-3-XPath , 2003.

[14] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation. http://www.w3.org/TR/REC-xml, 2000.

[15] World Wide Web Consortium. XML Path Language (XPath) 2.0.W3C Working Draft. http://www.w3.org/TR/xpath20/, 2002.

[16] World Wide Web Consortium, XQuery 1.0: An XML Query Language. W3C Working Draft. http://www.w3.org/TR/xquery, 2003.

[17] World Wide Web Consortium, XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft. http://www.w3.org/TR/xquery-operators, 2003.

[18] Yao, B.B., Özsu, M.T., and Keenleyside, J. XBench - A Family of Benchmarks for XML DBMSs. in Proceedings of EEXTT and DIWeb 2002 (Hong Kong, Aug. 2002), LNCS 2590, Springer, 2003, 162-164. http://db.uwaterloo.ca/~ddbms/projects/xbench/.