Clock: Synchronizing Internal Relational Storage with External XML Documents *

Xin Zhang[‡], Gail Mitchell[†], Wang-Chien Lee[†], and Elke A. Rundensteiner[‡]

(‡)Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609-2280
(xinz | rundenst)@cs.wpi.edu

(†) Verizon Communications
40 Sylvan Rd.
Waltham, MA 02451
(gmitchell | wang-chien.lee)@verizon.com

Abstract

In many business settings, a relational database system (RDBMS) will serve as the storage manager for data from XML documents. In such a system, once the XML data is dissembled and loaded into the storage system, XML queries posed against the (virtual) XML documents are processed by translating them into SQL queries against the relational storage. However, for applications which frequently update their XML documents, we cannot afford to reload a complete, possibly large, document for each update, instead we must be able to incrementally propagate document updates to the stored XML data. In this paper, we address the issue of correctly reflecting updates of external XML documents into the loaded XML data in a relational database system. We describe Clock, a framework for synchronizing the relational storage with updated XML documents by exploiting a metadata-driven technology. First, we propose a set of (DTD preserving) update primitives for XML documents. Second, based on the mapping between XML and the relational model, we describe the propagation of those update primitives. Validation of the updates ensures they will not violate the constraints specified by the DTD. We have implemented a working prototype of the Clock system using the IBM's XML4J parser, JDBC 2 and Oracle 8I. We report on preliminary experiments conducted using this prototype to analyze our algorithms in a document update setting.

*This work was supported in part by several grants from NSF, namely, the NSF NYI grant #IRI 97-96264, the NSF CISE Instrumentation grant #IRIS 97-29878, and the NSF grant #IIS 97-32897. Dr. Rundensteiner would like to also thank IBM for the IBM partnership award and Verizon Communications for partial support of Xin Zhang.

1 Introduction

XML is an emerging technology for information representation in web applications. By enabling automatic data flow between businesses, XML is pushing the world into the electronic commerce era. We envision that management of XML data will become an increasingly important task. In many business settings, a relational database system (RDBMS) will very likely serve as the storage manager for data from XML documents[1]. In such settings, the XML data needs to be transformed into the relational format and imported into the storage before it can be manipulated or queried. Moreover, a query result, once computed, must be converted back to the XML format for front-end applications. We call such a system that supports queries over XML data an *XML information system.* As shown in Figure 1, an XML information system acts as a middle layer between users' applications and the underlying XML data source(s). A RDBMS serves as the information system's storage for XML data and as the query processor. An XML query engine translates a user's XML query into SQL statements and exports the query result back to the XML format.

Recent studies investigate different approaches for storing XML data in RDBMS [7, 10, 5]. Work on translating XML queries into SQL statements and reconstructing the XML query results has also appeared in the literature [6, 2]. [7] and [10] both show that importing XML data into relational storage is an expensive yet essential step for query processing. Thus, for applications which frequently update their XML documents, e.g., online billing systems or trouble reporting systems, we cannot afford to reload a complete, possibly large, document for each update. Instead, we must be able to incrementally propagate document up-

[1] A number of recent studies are actively pursuing various issues related to storage of XML data in databases, such as relational, object-oriented, and semi-structured data storages. In our study, we choose relational database systems for their maturity, efficiency, and popularity.

0-7695-0957-6/01 $10.00 © 2001 IEEE

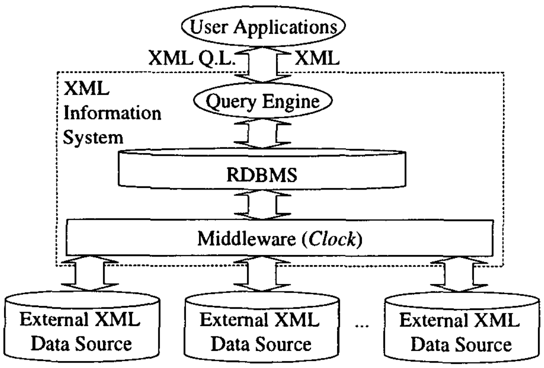

Figure 1. Architecture of XML Information System.

dates to the stored XML data.

In this paper, we describe our approach at supporting the propagation of updates to external XML data source(s) into updates of internal relational representation of those sources. We propose a middleware, called *Clock*, that can keep an internal relational data storage up-to-date with external XML documents. *Clock* uses an incremental approach to updating the internal data storage, hence keeping it up-to-date and query-ready with very little expense. The *Clock* system is based on a metadata-driven approach to XML management proposed earlier [9]. Metadata collected from a DTD document is used across all functionalities of the XML information system, including loading, importing, synchronization and querying. In this paper, we show the application of this metadata-driven approach to XML update propagation and validation. Specifically, we:

- Propose *Clock*, a data warehouse-like system that enables synchronization of updates to an external (possibly virtual) XML document with internal relational data storage.
- Argue the need for metadata in all XML management processes, including the update propagation process.
- Define a set of XML data update primitives and a translation from the large set of more general DOM data updates to these primitives.
- Develop algorithms for validating the updates against a document's DTD and for converting the XML update primitives into operations on the relational database.
- Implement a working *Clock* prototype using Oracle 8i, IBM's XML4J parser, and JDBC 2.
- Run some preliminary performance studies to assess the behavior of update propagation versus data loading.

The rest of the paper is organized as follows. In the next section, we describe the *Clock* architecture. Section 3 reviews our metadata-driven mapping solution. In Section 4 we detail the synchronizer of the *Clock* architecture. We present the implementation and initial experimental evaluation of the *Clock* system in Section 5. A summary and discussion of future work in Section 6 concludes the paper.

2 The Clock System Architecture

In the XML Information System of Figure 1, the external XML data sources can be viewed as (possibly virtual) collections of XML documents which can be updated by some text or document processing application. The XML Information System maintains the documents in relational data storage for query processing purposes, and thus must keep the internal storage synchronized with updates to the external XML data sources to ensure the correctness of the query results. The middleware shown in Figure 1 serves this purpose.

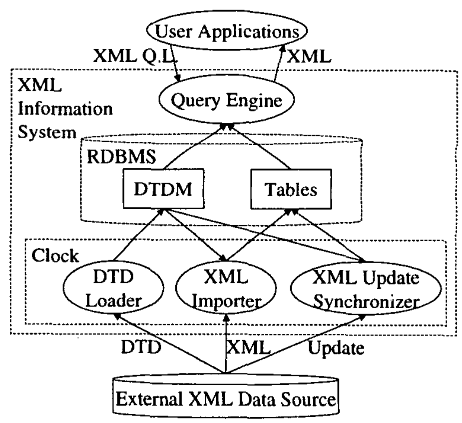

Figure 2. Architecture of Clock System.

In this section, we propose the *Clock* system as the middleware between external updates to XML documents and the information system's internal relational storage. While Figure 2 depicts the architecture of *Clock* composed of a *DTD loader*, *XML importer*, and *XML update synchronizer*, a detailed example of the processes are given in Sections 3 and 4.

DTD Loader loads a DTD, provided by an external source, into the RDBMS, and stores it as metadata. We called this the DTD Metadata (DTDM) [9]. The DTDM will be used by the other components of the system to identify the schema of XML data to be imported and synchronized.

XML Importer brings external XML data losslessly into the relational database. It uses a fixed mapping approach, although other approaches are also feasible [9].

XML Update Synchronizer synchronizes external XML data sources with their internal relational data. It is based on the same mapping approach used by the *XML importer*. We provide four XML update primitives that can be used to notify our system of updates to the external XML data sources. Those primitive changes are then propagated into updates to the relational data storage.

We assume that updates on external XML date sources by third-party applications are submitted to the *Clock* system expressed as DOM updates with each element identified by an XPath reference [11]. Then, the *Clock* system will translate the DOM updates into update primitives using the XPath as translated internal ID of the loaded data. Thereafter, the update primitives will be propagated by the synchronizer to the internal relational data storage. Further details can be found in Section 4.2.

3 Metadata-driven Mapping of XML

To import XML documents into an RDBMS, our first step is to identify a relational schema for the XML documents. We assume each document conforms to a DTD, and use the DTD to determine the schema. Information extracted from the DTD is first stored in metadata tables (Section 3.1), and then a mapping protocol is applied for creating a relational schema from this metadata (Section 3.2). We use a simple telephone bill, as shown in Example 1.

3.1 DTD Loader: Extract Metadata From the DTD

Document Type Definitions (DTDs) describe the structure of XML documents. We have identified the different structural elements described in a DTD in DTD metadata tables [9], e.g., element type definitions, attribute definitions, nesting relationships, groups, etc. Within an element type definition, elements associated within parentheses participate in a *grouping* relationship. Note that these kinds of properties and relationships are, in effect, constraints on the data.

In order to be able to make use of such metadata and constraints expressed in DTDs, we store the DTD information into relational tables as well. That will provide a uniform interface for tools such as the *importer* and *synchronizer* access to both data and metadata.

We classify the different types of DTD objects as *items*, *attributes*, and *relationships*. An *item* represents an object in a DTD that contains, or is contained by, other objects. Element types, groups, and PCDATA are all items. An *attribute* is a property of an item. An item can have multiple unique attributes. A *relationship* models the hierarchical connection between two items. We use three relational tables, called the DTD Metadata (DTDM) tables, to store the DTD objects and the properties of each. Each item, attribute or nesting relationship in the DTD is represented by a tuple in its respective table; with each tuple having an unique internal ID.

Example 1 DTD and XML of a telephone bill:

```
<?xml version="1.0" encoding="US-ASCII"?>
<!DOCTYPE invoice [
<!ELEMENT invoice (account_number,
                   bill_period,
                   carrier+,
                   itemized_call*,
                   total)>
<!ELEMENT account_number (#PCDATA)>
<!ELEMENT bill_period (#PCDATA)>
<!ELEMENT carrier (#PCDATA)>
<!ELEMENT itemized_call EMPTY>
<!ATTLIST itemized_call
          no ID #REQUIRED
          date CDATA #REQUIRED
          number_called CDATA #REQUIRED
          time CDATA #REQUIRED
          rate (NIGHT|DAY) #REQUIRED
          min CDATA #REQUIRED
          amount CDATA #REQUIRED>
<!ELEMENT total (#PCDATA)>
]>

<invoice>
  <account_number>555 777-3158 573 234 3</account_number>
  <bill_period>Jun 9 - Jul 8, 2000</bill_period>
  <carrier>Sprint</carrier>
  <itemized_call no="1" date="JUN 10"
    number_called="973 555-8888" time="10:17pm"
    rate="NIGHT" min="1" amount="0.05"/>
  <itemized_call no="2" date="JUN 13"
    number_called="973 650-2222" time="10:19pm"
    rate="NIGHT" min="1" amount="0.05"/>
  <itemized_call no="3" date="JUN 15"
    number_called="206 365-9999" time="10:25pm"
    rate="NIGHT" min="3" amount="0.15"/>
<total>$0.25</total>
</invoice>
```

As described in Figure 3 The **DTDM-Item** table captures the *type* of an item; i.e., whether it is a DTD element, group, etc. The **DTDM-Attribute** table stores attributes of items, thus referring via *pid* to the item to which the attribute belongs. The **DTDM-Nesting** table captures the relationships between different items. Figure 4 shows the DTDM tables for the DTD described in Example 1.

3.2 XML Importer: Mapping XML Data to Relational Model

We adopt a simple mapping approach for this paper to convert XML data into its relational form. However, alternate mapping strategies are possible [7, 10, 9] and could be

DTDM_Item

Fields	Meaning
ID	Internal ID for Items.
Name	Element Type or Group Name.
Type	Defines the type of this item within this domain, e.g. PCDATA, ELEMENT, GROUP, etc.

DTDM_Attribute

Fields	Meaning
ID	Internal ID of this attribute.
PID	ID of parent item of this attribute.
Name	Name of this attribute, e.g., AuthorIDs, id.
Type	Type of the attribute, e.g., ID, IDREFS.
Default	a keyword or a default literal value of this attribute, e.g., #IMPLIED.

DTDM_Nesting

Fields	Meaning
ID	Internal ID of this nesting relationship.
FromID	ID of parent item of this nesting relationship.
ToID	ID of child item of this nesting relationship.
Ratio	Cardinality between the parent item and child item.
Optional	Used to indicate whether a child item optional.
Index	The schema order of the child item.

Figure 3. Table Schema of DTDM Tables

achieved within our system simply, for example, by restructuring the loaded data using views. The mapping strategy we assume for the purpose of this paper is described next.

Item Mapping: *For each ELEMENT and PCDATA typed item defined in the DTDM-Item table, create an application table named item.Name. The table has three default columns:* **iid**, **pid**, **order**. **iid** *represents an internal unique ID that will be generated when the XML data is loaded.* **pid** *represents the* iid *of a parent item and* **order** *represents the local order among siblings.*

For example, Item Mapping creates an empty *itemized_call* table from the tuple with $id = 6$ in the $DTDM\_item$ table of Figure 4.

Attribute Mapping: *For each tuple* t *in the DTDM-Attribute table, create a column named* t.Name *of type string in the relational table identified by* t.pid[2].

For example, the columns of table *itemized_call* in Figure 5 are deduced from the records defined by $PID = 6$ in the $DTDM\_Attribute$ table (Figure 4).

Once the application tables for this particular DTD have been created, XML documents can be loaded by the Importer into these tables. The XML importer traverses a DOM tree uses the metadata mapping described above to move the XML data into relational tables. The left part of Figure 5 shows the DOM tree for the XML data of Example 1. The right part of the figure shows the data loaded into the corresponding relational tables. The dashed line shows the hierarchical relationships between nodes (or tuples). We hide the attributes in the DOM tree for clearness purpose.

[2]Further parsing on the value to decide its data type is the future work.

DTDM-Item

ID	Name	Type
1	PCDATA	PCDATA
2	invoice	ELEMENT.ELEMENT
3	account_number	ELEMENT.PCDATA
4	bill_period	ELEMENT.PCDATA
5	carrier	ELEMENT.PCDATA
6	itemized_calls	ELEMENT.EMPTY
7	total	ELEMENT.PCDATA

DTDM-Attribute

ID	PID	Name	Type	Default
1	6	no	ID	#REQUIRED
2	6	date	CDATA	#REQUIRED
3	6	number_called	CDATA	#REQUIRED
4	6	time	CDATA	#REQUIRED
5	6	rate	(NIGHT|DAY)	#REQUIRED
6	6	min	CDATA	#REQUIRED
7	6	amount	CDATA	#REQUIRED
8	1	value	CDATA	#IMPLIED

DTDM-Nesting

ID	FromID	ToID	Ratio	Optional	Index
1	2	3	1:1	false	1
2	2	4	1:1	false	2
3	2	5	1:1	false	3
4	2	6	1:n	true	4
5	2	7	1:1	false	5
6	3	1	1:1	false	1
7	4	1	1:1	false	1
8	5	1	1:1	false	1
9	7	1	1:1	false	0

Figure 4. DTDM Tables for DTD in Example 1.

4 XML Update Synchronizer

The synchronizer in the *Clock* system (Figure 2) is responsible for incrementally updating the internal relational data storage to reflect updates to external XML documents.

4.1 XML Update Primitives

The JAVA binding of the DOM APIs typically defines a set of DOM data updates (Figure 6). These updates are redundant and not primitive. For example, the DOM functions $Node.setNodeValue()$, $Attr.setValue()$, and $Element.setAttribute()$ are basically the same, all three setting the value of an attribute of an element. Also, the functionality of the $Text.splitText()$ function overlaps with that of $Document.createTextNode()$ and $Node.setNodeValue()$. Hence, instead of implementing this set of DOM updates, we instead determine a minimal set of simple primitive — for update — which then could be used to compose any of the DOM update functions.

We designed data update primitives with the following goals: 1) *Data oriented:* These updates are DTD preserving, i.e., they update XML data but not their DTDs. 2) *Complete:* The set completely covers all the data updates a user could specify at the DOM level. 3) *Consistent:* The consistency of internal data storage with external XML data is guaranteed. 4) *Valid:* Any update will be validated against the DTD, trapping any incorrect update. 5) *Atomic:* Updates are minimal, changing only

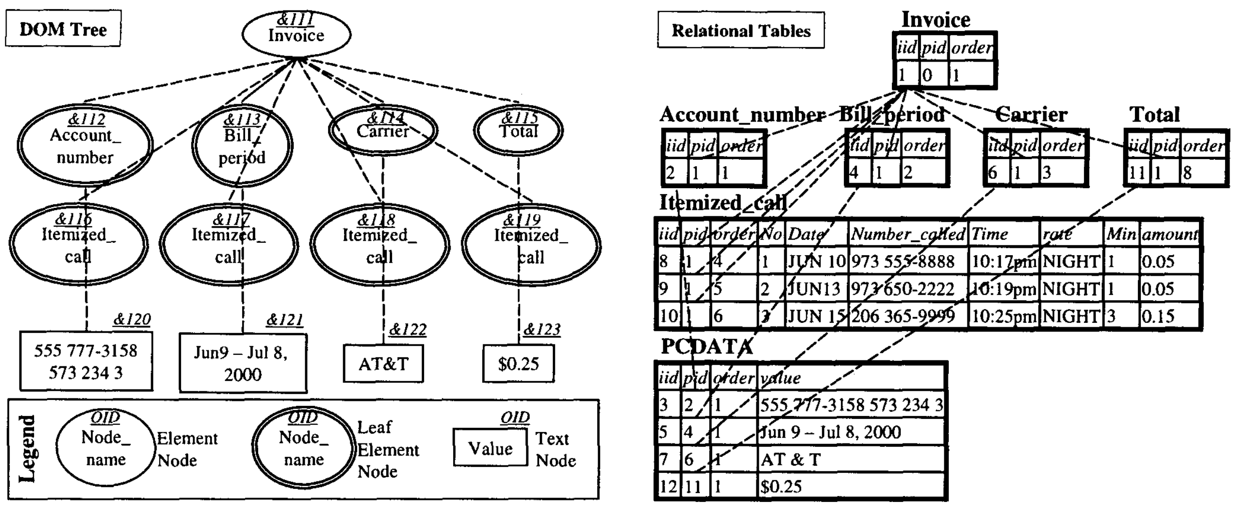

Figure 5. DOM Tree and Relational Tables Imported from XML in Example 1.

DOM Data Updates	Meaning
Node.setNodeValue()	set the value property of a specific node.
Node.insertBefore()	insert a new child node to this node before a specific child.
Node.removeChild()	remove a specific child node.
Document.createElement()	create a new element in a document.
Document.createTextNode()	create a new text node in a document.
Attr.setValue()	Set the value of the attribute node.
Element.setAttributeNode()	set an attribute node of an element node.
Text.splitText()	split a text node into two text nodes.

Figure 6. DOM Data Updates.

DOM Data Updates	Update Description
Node.setNodeValue()	ModifyElement
Node.insertBefore()	MoveElement
Node.removeChild()	DeleteLeafElement (maybe a set of)
Document.createElement()	CreateLeafElement
Document.createTextNode()	CreateLeafElement
Attr.setValue()	ModifyElement
Element.setAttributeNode()	ModifyElement
Text.splitText()	ModifyElement, CreateLeafElement, MoveElement

Figure 7. Mapping from DOM Data Updates to Clock Update Primitives.

one element at a time. In the future, sets of objects can be updated by iterating over our tuples using the same primitives. The *Clock* system provides four update primitives in terms of XML documents: $CreateLeafElement$, $DeleteElement$, $ModifyElement$, and $MoveElement$. The mapping from DOM data updates to *Clock* update primitives is listed in Figure 7. Due to space limitations, we only explain one representative DOM update: $Text.splitText()$. $Text.splitText()$ takes two parameters: the OID of the parent node and the offset of the split. Our system will translate that to three primitives: First $CreateLeafElement$ will create one text element with the right part of the split text, then $ModifyElement$ will update the current text element to only contain the left part of the split text, and finally, $MoveElement$ will move the newly created text element to be placed after the original text element.

4.2 General Update Propagation Process

We can see that the DOM operations are object-oriented operations, in the sense that every node is identified by its OID. We cannot guarantee that the OID used by the external XML data source will be the same as the internal IID used in the relational storage. To assure identification of the same elements, such identification of items in DOM could be achieved by assuming that the XML data source uses the XPath to uniquely identify the to be modified node. For example, the XPath of the node with OID &115 in Figure 5 is "/invoice[1]/itemized_call[1]".

However, in *Clock*, elements are identified by the pair *node_name* and *iid*, where *node_name* leads to the relational table in which data is to be found with the *iid*. The *iid* can be easily computed from XPath information by an XPath-to-iid index. The *node_name* can be gotten by parsing the XPath. For example, "/invoice[1]/itemized_call[1]" will be translated into "iid=8" and "node_name=itemized_call" (Figure 5).

Hence, the DOM update *Node.removeChild()* for node "/invoice[1]" deleting the first child node "/invoice[1]/itemized_call[1]" will be be translated into the *Clock* primitives: *DeleteLeafElement(8, "itemized_call")*. In general *Clock* first finds the correct table name and update statement, then the synchronizer will propagate those updates into the relational storage. Concepts needed for the more detailed explanation of the update propagation are given in Definition 1.

Definition 1 Sibling Tuples of tuple `t`**:** *The tuples corresponding to sibling nodes of the node corresponding to the tuple* `t` *are called* **sibling tuples of tuple** `t`.

Sibling Tables of tuple `t`**:** *The tables containing the sibling tuples of tuple* `t` *are called the* **sibling tables of tuple** `t`.

4.3 Propagation of XML Updates

This section now shows how to map these four primitive updates into operations on their relational counterpart by explicitly exploiting knowledge in the DTDM tables.

Create Leaf Element Operation. *CreateLeafElement(node_name, list_of_attributes)*: This operation creates a new leaf element and returns its new *iid*. This element is not connected to any existing element yet. The connection could be created later by the *MoveElement* primitive [3].

The *CreateLeafElement* operation will add one tuple to the table identified by *node_name* with the applicable attributes that is known from DTDM tables, and empty *pid* and *order*. A unique *new_iid* is generated by the system for the new tuple. The SQL template generated for the update is:

```
INSERT INTO <node_type> (iid, pid, position,
                         <list_of_attributes>)
VALUES (<new_iid>, null, null,
        <values_of_list_of_attributes>)
```

Modify Element Operation. *ModifyElement(iid, node_name, attribute_name, new_value)*: This operation updates the attribute specified by *attribute_name* of an element identified by *iid* of type *node_name* with the new value *new_value*. The SQL template is:

```
UPDATE <node_type> SET <attribute_name>=<new_value>
WHERE iid = <iid>
```

Delete Leaf Element Operation. *DeleteElement(iid, node_name)*: This operation deletes the leaf element identified by the *iid* of element type *node_name*. The *DeleteLeafElement* first gets a list of the sibling tables of the current element by querying the DTDM tables. Then, it goes through the sibling tables to decrease the positions of the sibling tuples of the to be deleted tuple. Third, we delete the to be deleted tuple identified by the *iid* from the table *node_name*. At last, if the to be deleted tuple is of type ELEMENT.PCDATA as noted in the *DTDM_item* table, we also delete the tuple of the to be deleted tuple from the PCDATA table.

Move Element Operation. *MoveElement(iid, iid_node_name, pid, pid_node_name, new_position)*: This operation moves the element identified by the *iid* of element type *iid_node_name* as the child of the element identified by the *pid* of element type *pid_node_name* into the position *new_position*.

The complexity of the relationship between old and new positions of an element complicates this operation. There are three kinds of relationships between the two positions. If the element is moved between different parent elements (either located in one table or two different tables), then increase the position of the tuples in the sibling tables that are larger than the *to* position, and decrease the position of the tuples in the sibling tables that are larger than the *from* position.

If the element is moved within the same parent, there are two cases. First, if the *from* position is less than the *to* position, then we decrease positions of the sibling tuples with the position greater than the *from* position but less than the *to* position. Second, if the *from* position is larger than the *to* position, we increase the positions of sibling tuples with the position greater than the *to* position but less than the *from* position.

Finally, we update the *pid* and position of the moved tuples.

4.4 Validation of XML Updates

An XML document is said to be valid if it is compliant to a specific DTD. Though most of current available XML parsers can validate the whole XML document, they do not validate an XML update. If updates are specified without any kind of validation checking, The data would then be in a non-valid state according to its DTD. Hence, *Clock* will incrementally validate the update based on the DTD (i.e., the metadata captured in the DTDM tables) before executing the update operation. We support three kinds of update validations:

- *AttributeCheck(node_name, attribute_name, new_value)* will check whether the new value of the attribute satisfies the specification of that attribute. It will also check the uniqueness of the ID/IDREF typed attribute using the DTDM_attribute table. This check will be used for the validation of *CreateLeafElement* and *ModifyElement*.

[3] An insertion of a leaf element can be represented by the combination of *CreateLeafElement* and *MoveElement*.

- *NestingCheck(node_name, iid)* will check the quantifier of the nesting relationship between the element identified by the *node_name* and *iid* with its parent element, by using the DTDM_nesting table. This will be used for the validation of *DeleteElement*.

- *NestingCheck(from_node_name, from_iid, to_node_name, to_pid)* will check the quantifier of the nesting relationship between the element identified by *from_node_name* and *from_iid* with its parent and also the nesting relationship with the element identified by *to_node_name* and *to_pid*, by using the DTDM_nesting table. This will be used for the validation of *MoveElement*.

5 Preliminary Evaluation

In order to verify the feasibility of our *Clock* system and to study the characteristics of the performance of the update propagation, we implemented and then evaluated the system.

We have verified the correctness of our system by loading the XML data in, updating the XML data through the XML update primitives, and then confirming the correctness of the exported XML data.

First, we use a fixed XML data set and vary the number of data updates to get a sense of the point where the update propagation is worse than the reloading. Second, we use a fixed XML data set and test the overheads of different types of update primitives.

We have implemented the *Clock* system, including the update propagation system, in Java. The database back-end is Oracle 8i running on a PII400 with 256MB. Update validation is implemented in Java using JDBC calls. We perform our experiments on a Windows NT 4.0 workstation with a 128 MB PII400. We use Shakespeare's play XML data test set [1], which contains 7MB of the XML documents of the plays as test data set for our experiments.

5.1 Experiment on Incremental Update vs. Reloading

Clearly, if the database has to be updated in real time we want to propagate updates. However, if the database is updated periodically, as in a data warehouse, we have the option of choosing when to update. This experiment gives information on how to make that choice.

We study a set of mixed *CreateLeafElement*, *DeleteLeafElement*, *MoveElement*, and *ModifyElement* update primitives. The four types of updates have equal probabilities of occurrence and are evenly distributed over the documents. Our results are shown in Figure 8.

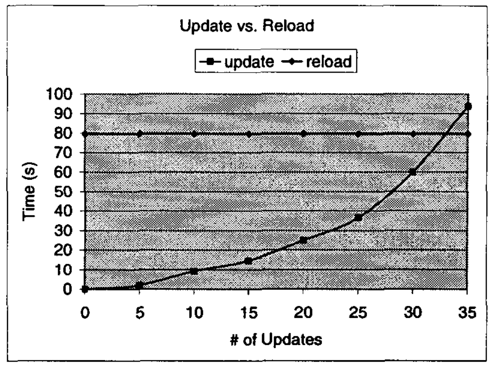

Figure 8. Time of Reload vs. Update

As we can see from Figure 8, for the above data set, the performance of the update propagation is slower than reloading the whole document up to roughly 30 to 40 updates. We have found similar results for different test data sets. This experiment shows that an update is definitely much faster than a complete reloading, but due to less optimization of our system. Given an scenario with low refresh requirement of the loaded XML data and very busy environment with many updates, reloading is still a considerable choice.

5.2 Experiment of Different Types of Updates

Different update primitives have different performance. In this example we want to determine the relative cost of the four different types of update primitives. For this, we use the full data size of 7MB of XML data. We have done experiment for 20 updates of each type of update primitives separately.

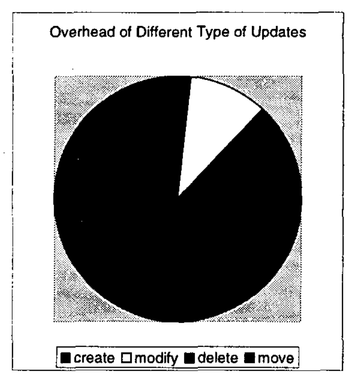

Figure 9. Overhead of Different Types of Updates

As we can see in Figure 9, the *MoveElement* and *DeleteLeafElement* are more expensive than the *CreateLeafElement* and *ModifyElement*. The reason for that is that the *CreateLeafElement* and *ModifyElement* only affect a single element, while the *MoveElement* and *DeleteElement* will affect all their siblings. Contrasting the later two more closely, we can easily see that the *MoveElement* is more expensive than the *DeleteElement*, because the *MoveElement* must update both the sibling elements of *from* and *to* positions, while *DeleteElement* only updates the sibling elements of the *delete* position. This gives us a better insight into how the update propagation behaves. A further performance optimization could be done based on this experiment with the goal to support ordered data types like lists in the DBMS.

6 Conclusions

In this paper, we have proposed a *Clock* system to keep internal relational data synchronized with external XML data. We have identified four type of primitives, i.e., create leaf element, delete leaf element, move element, and modify element. We also designed an update propagation and validation algorithms. Our experiments confirm that the create element is the cheapest operation among those four primitives. Also, we can see that the update synchronization is typically faster than performing a complete reload of modified XML documents.

Open issues to be addressed next include: design a general language for mapping from XML into relational model [12], and propose an algorithm of transformation a XML Query Language XML-QL [4], or Quilt [3] into SQL based on the given mapping description [12].

References

[1] J. Bosak. Shakespeare 2.00. http://metalab.unc.edu/bosak/xml/eg/shaks200.zip.

[2] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Publishing Object-Relational Data as XML. In *VLDB*, pages 646–648, 2000.

[3] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *WebDB*, pages 53–62, 2000.

[4] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *Proceedings of the Eighth International World Wide Web Conference (WWW-8)*, pages 1155–1169, 1999.

[5] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 431–442, Philadephia, USA, June 1999.

[6] M. Fernandez, W. Tan, and D. Suciu. SilkRoute: Trading between Relations and XML. http://www.www9.org/w9cdrom/202/202.html, May 2000.

[7] D. Florescu and D. Kossmann. Storing and querying xml data using an rdbms. In *Bulletin of the Technical Committee on Data Engineering*, pages 27–34, September 1999.

[8] H. A. Kuno and E. A. Rundensteiner. Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation. *IEEE Transaction on Data and Knowledge Engineering*, 10(5):768–792, Sep./Oct. 1998.

[9] W. Lee, G. Mitchell, and X. Zhang. Integrating xml data with relational database. In *Int. Conference Distributed Computing Systems*, 2000.

[10] J. Shanmugasundaram, G. He, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99)*, pages 302–314, Edinburgh, Scotland, UK, September 1999.

[11] W3C. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath, 1999.

[12] X. Zhang and E. A. Rundensteiner. Rainbow: Bridge over the Gap between XML and Relational Databases. Technical report, Worcester Polytechnic Institute, 2000. in progress.