Reform Based Version Management System for XML Data

Sayed MD. Fahim Fahad
Department of Computer Science and Engineering
Jahangirnagar University
Dhaka, Bangladesh

Md. Abdur Rafi Ibne Mahmood
Department of Computer Science and Engineering
Jahangirnagar University
Dhaka, Bangladesh
Email : rafisameen2 {at} gmail.com

Mohammad Zahidur Rahman
Department of Computer Science and Engineering
Jahangirnagar University
Dhaka, Bangladesh

Abstract—**XML has become a popular medium to store data. As there could be multiple versions of an XML document, users may want to search previous versions, detect changes in documents and retrieve a particular document version efficiently and quickly. This paper proposed an efficient way to manage versions of XML data that will save both memory space and process time.**

Keywords- XML; Version control; Forward delta

I. INTRODUCTION

Version Control is a very necessary and efficient way to manage any project or system. A version control system (or revision control system) is a system that tracks incremental versions (or revisions) of files and in some cases, directories over time. Collaboration, change management and ownership tracking are the main reasons for using version control system. As the use of XML data is increasing because of its' usefulness and efficiency, many users use XML to store, update and search data. [6][7][8][9]Many versions of same XML file exist and the management of each version is a difficult task. Managing both time and space complexity for version management is difficult. Many version control system use more space to reduce time complexity. Here we tried to handle both space and time complexity. We use "Reform" process and forward deltas to keep track of the changes. All the technical terms will be described in the later part of the paper.

II. RELATED WORKS

In [2], the forward and backward deltas are used to manage versions. But we used forward deltas to manage versions that will save memory space and calculation time to retrieve a version. In [5], the changed objects are added to the new version and unchanged objects are referenced. This may become time consuming for a growing system to retrieve a version from huge file. Time is not constant or predictable. [1],[3],[4] also discussed about XML version management. Based on these papers we tried to improve the time and space complexity for XML version management process. Here we use "Reform" technique that will regenerate the complete version after a specific number of versions and store it to new XML file. This will allow us to retrieve any version of XML data in a very short time. Forward deltas will save much memory space. We don't have to store complete version every time or backward edit script to regenerate any version.

III. LOGICAL MODEL

In the paper, we will use few technical terms. Here we first elaborate what these terms actually mean.

A. *Forward deltas:*

We use forward deltas to keep track of the changes. Here delta means new child node that contains the updated values of the original child node of the previous version. A forward delta is a child node that can be applied to a complete version of a document to obtain the next complete version of a document. The structure of the forward delta will be same as the structure of the child node of XML document for which we are maintaining our versions. The deltas will be stored at the end of the latest XML file.

B. *Reform and Specific version number:*

Our system is a "Reform" based system. The system will regenerate the complete version of an XML file after a specific number of versions and store it to a new XML file. We call this process "Reform". So at the beginning of the version management, this specific version number must be fixed. This will depend on the kind of system that uses the model. This "specific version number" will give us some mathematical advantages to perform any operation on the file. e.g. if the specific version number is 4 then the XML file will be reformed after every 4^{th} version. This means 5^{th}, 9^{th} and so

on. But no sub-version like 2.1 or 3.5 will trigger the "Reform" operation. Users can create any number of subversions. This will save much memory space because the system does not have to store a new XML file for every new version.

Now we will use the model to manage the versions of an XML file. Initially an XML file will be created to store data. Let us consider that the user named the file as "abc.xml" and stores a complete version of XML data. This is the first version of the file. The file structure is shown below:

```
<Root>
    <Child>
            <Status></Status>
            <Version></Version>
            <Sub-child > </ Sub-child >
                    …
    </Child>
    ...
</Root>
```

The structure of deltas will be like the child node. The "Status" and "Version" sub child node will be used for version control. They will indicate the status and version of the child node. Now we will use our naming process to make the file more manageable. But the user of that file may not be needed to know about that. The new name of the XML file will be "FileName + (Version/n + 1)" (n = specific version number). Version indicates the current version.

It may comes to mind that why the naming of the file is important for version control. As we said earlier the specific version number plays a crucial role in our model. For any xml file "abc.xml" the first "n" versions of the file will be stored in "abc1.xml" (considering specific version number is n). Then the $(n+1)^{th}$ to $2n^{th}$ version will be stored in "abc2.xml" file and so on. Now if we want to search any version of that xml file, we can use our mathematical equation to find which file contains that version. We don't have to check any other file or database to track that version. E.g. if we search p^{th} version of "abc.xml" where specific version number is n then it will be in the "FileName+(((p)/n)+1)" file. This will save much time to retrieve a version of an XML file.

We will need to use an index structure to maintain the versions of XML data. The index structure will contain information like this:

File Name	Child node identifier	Version	Initial version	Status

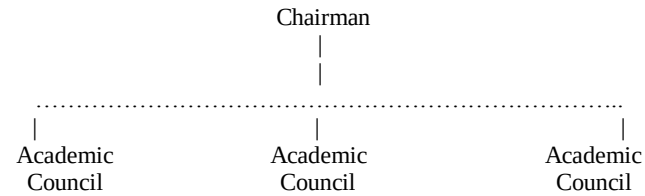
The "File Name" will contain the name of the XML file. Every node in the file must be identified uniquely. The node that uniquely identifies the child node should be used as an identifier. Version shows when the node was last updated. Initial version indicates the version when the node was first introduced. Status shows whether the node is still available or deleted.

Another file index is needed to that will help us to track us the latest version and handle reform operation.

File Name	Latest Version

Latest version will allow us to know what the latest version of that file is. This information will help us to decide whether the file is already reformed or not. E.g if we made a change to an XML file and that triggers the reform operation then we have to avoid the reform operation for next changes of the same version. Latest version will help us to avoid unnecessary reform operation.

Now we will use "Course syllabus system" to explain our model. The basic structure for a department is shown below:



For CSE department Honors degree, The Chairman will create an XML file to store the syllabus and set the specific version number for the system. Let's consider the file name is "HonorsCSE.xml" and specific version number is 4. Then academic councilors will create course syllabuses. The chairman approves them and adds them to the main syllabus. This will be considered as 1st version of the syllabus. So the first version of syllabus will be stored in "HonorsCSE+((1/4)+1)" or "HonorsCSE1.xml" file. The example structure of the syllabus is shown below:

```
<Syllabus>
    <Course>
            <Status></Status>
            <version></ version>
            <Name> </Name>
            <CourseID></CourseID>
            <Credit></Credit>
            <CourseSyllabus></ CourseSyllabus>
            <Reference></ Reference >
    </Course>
    ...
</Syllabus>
```

So the file index will be:

File Name	Latest Version
HonorsCSE	1

Now if academic council makes any changes in any of the courses of the syllabus, the changes will be stored in the XML as deltas and a new version number will be given to those delta parts. For example if the academic council changes the

"Reference" of a course then the new child node will be created for that course and version number and "Reference" will be changed but all the other sub child nodes will be same. This delta part will be tracked against the course syllabus that is changed in the new version. The deltas will be stored at the end of the XML file. After every fourth version, the syllabus will be reformed and will generate the latest complete version. This latest complete version will be stored in a new XML file. The name of the new XML file will be given by using the proposed naming technique. For example, if the latest version is 5 then the model will generate the complete 5^{th} version of the syllabus and write the syllabus to file "HonorsCSE2.xml" file where as the 2 comes form ((version=5)/4+1) and set latest version to 5. The whole process can be expressed as below:

Version 01 = Complete version 01
Version 02 = Version 01 + changes in the Version 01
Version 03 = Version 02 + changes in the Version 02
Version 04 = Version 03 + changes in the Version 03
Version 05 = Reform and generate complete 5^{th} version and store it to new XML file

Thus the process will continue. We used a course index to maintain the versions of the XML data. The course index is shown below:

File Name	Subject Name	Version	Initial version	Status

"Subject name" is the unique identifier for this file. All these information will be used for the operations like insert, update and delete. The benefit of storing deltas is we don't have to store a complete version of a syllabus for each time we make changes to it. This will allow us to retrieve any version of the syllabus in quick time. When the user search a version, we need to find the closest complete version of the syllabus and then we will be able to rebuild the requested version form the complete version and the deltas.

IV. OPERATIONS

A. Insert operation:

For insert operation, we will insert the new child nodes (deltas) at the end of the current XML file that contains the latest version of the syllabus. The insert operation will force some changes in the course index. We will set the initial version and version to the current version and set status to "current. But if current version!=1 and (current version%n=1 and Latest version%n!=1) then we have to generate the complete latest version and insert the deltas at the end of the new XML file. Here n is specific version number. If the (Latest version < current version) then set "Latest version" to current version.

B. Update operation:

For update operation, we will search the child node using child node identifier. Then the new child node will be added to the end of the XML file and changed the sub child node values that are updated. Other values will remain unchanged. We will set course index version value to current version. But if current version!=1 and (current version % n =1 and Latest version % n !=1) then we have to generate the complete latest version and insert the updated deltas at the end of the new XML file. Here n is specific version number. If the (Latest version < current version) then set "Latest version" to current version.

C. Delete operation:

Delete operation is the simplest amongst the operations. This will only set the status value to "Deleted" and update the version to (current version-1). This will indicate that the subject was active till that version.

D. Version retrieval:

To retrieve any version of XML data we need to find which file contains the version. If we want to retrieve p^{th} version of the file then by using our equation we can say that it is in "FileName+((p/n)+1).xml" file. n is specific version number. This is where our naming process becomes very useful. We can easily find the required file without even looking in the database. Now we need to determine the closest complete version of the XML data. The equation for finding closest complete version is (((p/n)*n) + 1). So we will use complete version and the deltas of the complete version to retrieve the searched version of XML data.

E. Query:

We can use two criteria to query the XML file. We can use child node identifier and a target version or a target version number to retrieve a complete version.

Identifier node and targeted version number will return matched node(s) of that particular version. For this we will use the information of index structure. First the specific version will be checked against the initial version of that node. If initial version <= targeted version then we will check it against the version. If version<=targeted version and (version%n!=0) then the node will be found in the "FileName+((targeted version/n)+1)" XML file. Otherwise if ((targeted version %n)==0) "FileName + (((targeted version-1)/n)+1)" contains the node. But if (initial version<targeted version < version) then we have to find the nearest complete version of the file and related deltas to find the node.

For a specific version of an XML file, we use the version retrieval algorithm and show user the targeted version of that file.

F. Change detection:

The users may want to detect the changes of a particular version. User will provide the version number and the model will calculate to find the file name that contains that version.

Then the version number will be checked against the <version> sub-child node. When model finds a match, it will return the child node. The whole file will be searched to detect all the changes.

V. ALGORITHMS

Insert (node, ver, LatestVersion)

1. //ver indicates the current version
2. //n indicates specific version number
3. if(ver!=1 && ver%n==1 && LatestVersion%n!=1)
4. Reform(ver)
5. insert node at the end of the file
6. else
7. insert node at the end of the file
8. initial version = ver; version = ver;
9. if(LatestVersion < ver)
10. LatestVersion = ver

Update (node, ver, LatestVersion)

1. if(ver!=1 && ver%n==1 && LatestVersion%n!=1)
2. Reform(ver)
3. insert the updated node at the end
4. else
5. insert the updated node at the end
6. version = ver;
7. if(LatestVersion < ver)
8. LatestVersion = ver

Delete (node)

1. Find the node from the latest version file
2. set status to "Deleted"

Reform (ver)

1. Use (ver-n) to (ver-1) versions to generate the complete version ver
2. Store new complete version ver to new XML file

VersionRetieval (ver)

1. if(ver % n == 1)
2. ver is complete version
3. return complete version 'ver'
4. else
5. closest complete version number =((ver/n)*n)+1
6. V = store closest complete version
7. find deltas of (v+1) to (ver) versions
8. use v and deltas to generate the complete version ver
9. cver = store the complete version
10. return cver

VI. PERFORMANCE

At the beginning we said that our model will handle both space and time complexity. The model use deltas to store the changes and an index structure to track it. We don't need to store the complete version or forward and backward deltas to manage versions of the file. As deltas will be stored at the end of the current XML file, we don't have to create a new XML file every time to store new version of XML data. This will save lots of space. Now if we store all the deltas in the same file it may perform better for smaller versions but it is not efficient for long term management. When versions increase with time it will become more difficult to track and retrieve a version form a single file. To overcome that problem we use "Reform" operation after a specific number of versions. This will regenerate the complete version and store it to a new XML file. Every XML file contains the same number of versions of the XML data. So, searching version 2 or version 20 will take almost same amount of time. On the other hand, searching a specific node of a specific version, we don't have to generate the whole version of that file. Index structure will help us to find that node easily. This will save much process time and make the system more predictable.

VII. EXAMPLES

Example 01(insert operation):
If we insert 4 subjects in four consecutive versions then the state of the course index will be:

Degree	Department	Subject Name	Version	Initial version	Status
Honors	CSE	DLD	1	1	Current
Honors	CSE	Database	2	2	Current
Honors	CSE	OOAD	3	3	Current
Honors	CSE	Algorithm	4	4	Current

So the first 4 versions will be stored in "HonorsCSE1.xml" file for the syllabus will look like:

```
<Syllabus>
    <Course>
            <Status>Current</Status>
            <version>1</ version>
            <Name> DLD</Name>
            <CourseID>101</CourseID>
            <Credit>2</Credit>
            <CourseSyllabus>Syllabusnew</
    CourseSyllabus>
            <Reference>Book01</ Reference >
    </Course>
    …
</Syllabus>
```

Example 02(update operation):

If we update DLD and change the Credit from 2 to 3 and set the new version number to 5 and (5%4==1). So the "Reform" operation will be performed and a complete 5th version will be stored in the "HonorsCSE2.xml" file. So the new course index will look like:

Degree	Depart ment	Subject Name	Version	Initial versio n	Status
Honors	CSE	DLD	5	1	Current
Honors	CSE	Database	2	2	Current
Honors	CSE	OOAD	3	3	Current
Honors	CSE	Algorithm	4	4	Current

So "HonorsCSE2.xml" file for the syllabus will look like:

```
<Syllabus>
   <Course>
        <Status> Current </Status>
        <version>2</ version>
        <Name>Database</Name>
        <CourseID>102</CourseID>
        <Credit>3</Credit>
        <CourseSyllabus>Syllabus DB</ CourseSyllabus>
        <Reference>Book02</ Reference >
   </Course>
   <Course>
        <Status> Current </Status>
        <version>3</ version>
        <Name> OOAD</Name>
        <CourseID>103</CourseID>
        <Credit>3</Credit>
        <CourseSyllabus>
                SyllabusOOAD
        </ CourseSyllabus>
        <Reference>Book 03</ Reference >
   </Course>
   <Course>
        <Status> Current </Status>
        <version>4</ version>
        <Name> Algorithm</Name>
        <CourseID>104</CourseID>
        <Credit>3</Credit>
        <CourseSyllabus>
                Syllabus Algorithm
        </ CourseSyllabus>
        <Reference>Book 04</ Reference >
        </Course>
        <Course>
                <Status>Current</Status>
                <version>5</ version>
                <Name> DLD</Name>
                <CourseID>101</CourseID>
                <Credit>3</Credit>
                <CourseSyllabus>Syllabusnew</
        CourseSyllabus>
                <Reference>Book01</ Reference >
        </Course>
   </Syllabus>
```

Example 03(delete operation):
If we delete Database from the syllabus then the status of the Database will be set to "Deleted" and version to 5 as the current version is 6. Others will remain unchanged. So the new course index will look like:

Degree	Depart ment	Subject Name	Versi on	Initial version	Status
Honors	CSE	DLD	5	1	Current
Honors	CSE	Database	5	2	Deleted
Honors	CSE	OOAD	3	3	Current
Honors	CSE	Algorith m	4	4	Current

So "HonorsCSE2.xml" file for the syllabus will look like:

```
<Syllabus>
    <Course>
        <Status> Deleted </Status>
        <version>5</ version>
        <Name>Database</Name>
        <CourseID>102</CourseID>
        <Credit>3</Credit>
        <CourseSyllabus>
                Syllabus database
        </ CourseSyllabus>
        <Reference>Book02</ Reference >
    </Course>
    …
</Syllabus>
```

Example 04(Version Retrieval):
If we want to retrieve the version 6 for the CSE Honors then we will find it in "HonorsCSE2.xml" file (using (ver/n) + 1 where ver=6 and n=4).
Here closest complete version = ((ver/4) * 4) + 1 = ((6/4)*4) +1 = 5
So we will use complete version 5 and deltas of version 6 to generate complete searched version 6. Now complete version 6 will look like:

```
<Syllabus>
    <Course>
        <Name> OOAD</Name>
        <CourseID>103</CourseID>
        <Credit>3</Credit>
        <CourseSyllabus>
                Syllabus OOAD
        </ CourseSyllabus>
        <Reference>Book 03</ Reference >
```

```
    </Course>
    <Course>
            <Name> Algorithm</Name>
            <CourseID>104</CourseID>
            <Credit>3</Credit>
            <CourseSyllabus>
                    Syllabus Algorithm
            </ CourseSyllabus>
            <Reference>Book 04</ Reference >
    </Course>
    <Course>
            <Name> DLD</Name>
            <CourseID>101</CourseID>
            <Credit>3</Credit>
            <CourseSyllabus>Syllabusnew</
    CourseSyllabus>
            <Reference>Book01</ Reference >
    </Course>
</Syllabus>
```

VIII. CONCLUSION

Handling both space and time complexity is a big performance issue for any model. In our model we show an efficient way to manage multiversion XML data that can handle both space and time complexity by using forward deltas and performing "Reform" operation. Query multi version XML data becomes efficient as we don't have to regenerate every version.

REFERENCES

[1] Amélie Marian, Serge Abiteboul, Laurent Mignet, "Change-centric Management of versions in an XML warehouse", 2001

[2] Raymond K. Wong and Nicole Lam, "Managing and Querying Multi-Version XML Data with Update Logging", 2002.

[3] Shu-Yao Chien, Vassilis J. Tsotras, Carlo Zaniolo, "Copy-Based versus Edit-Based version Management Schemes for Structured Documents"

[4] S-Y. Chien, V.J. Tsotras, and C. Zaniolo, "Version Management of XML Documents", WebDB 2000 Workshop, Dallas, TX, 2000.

[5] Shu-Yao Chien, Vassilis J. Tsotras, Carlo Zaniolo, "Efficient Management of Multiversion Documents by Object Referencing ", 2001.

[6] XML:DB Initiative for XML Database. (http://xmldb-org.sourceforge.net/)

[7] XML Databases - The Business Case. (http://www.cfoster.net/articles/xmldb-business-case/)

[8]Simon st.Laurent, "Why XML?", (http://www.simonstl.com/articles/whyxml.htm)

[9] Bill Trippe and Dale Waldt, "Using XML and Databases", 2008.