Version Management of XML Documents

Shu-Yao Chien
Computer Science Department
University of California, Los Angeles
CA 90095, USA
csy@cs.ucla.edu

Vassilis J. Tsotras
Department of Computer Science and Engineering
University of California, Riverside
CA 92521, USA
tsotras@cs.ucr.edu

Carlo Zaniolo
Computer Science Department
University of California, Los Angeles,
CA 90095, USA
zaniolo@cs.ucla.edu

ABSTRACT
The problem of ensuring efficient storage and fast retrieval for multi-version structured documents is important because of the recent popularity of XML documents and semistructured information on the web. Traditional document version control systems, e.g. RCS, which model documents as a sequence of lines of text and use the shortest edit script to represent version differences are inadequate since they do not preserve the logical structure of the original document. Instead we propose a new approach where the structure of the document is preserved intact, and their sub-objects are timestamped hierarchically for efficient reconstruction of current and past versions. Our technique, called the Usefulness Based Copy Control (UBCC), is geared towards efficient version reconstruction while using small space overhead. Our analysis and experiments illustrate the effectiveness of the overall approach to version control for structured documents.

1. INTRODUCTION
The problem of managing multiple versions for XML and semistructured documents is of significant interest for content providers and cooperative work. The XML standard is considering this problem at the transport level. The WEBDAV working group [9] is developing a standard extension to HTTP to support version control, meta data and name space management, and overwrite protection. In this paper, we take a database-oriented view to the problem of storing, processing and querying efficiently multiple versions of documents by drawing on related work done in the realm of databases and persistent object managers. In fact, the flat structure of traditional file systems, currently used for the storage of these documents, is not best suited to support the raising demand for more powerful, complex computations on XML-based information, such as querying on content and document restructuring. Our interest focuses on managing structured documents where tagged objects are time-stamped with either their version number or their life-span.

2. PREVIOUS WORK
Traditional line-based methods for version management, such as RCS [2] and SCCS [1], are not efficient for object-oriented structured documents. In fact, since these methods store versions using editing differences, information from many such differences are often required to reconstruct a single version. This results in more I/O cost and computation complexity in reconstructing the document's objects.

In particular, RCS [2] stores the most current version intact while all other revisions are stored as reverse editing scripts. These scripts describe how to go backward in the document's development history. For any version except the current one, extra processing is needed to apply the reverse editing script to generate the old version. Another popular text version management tool is SCCS [1]. Instead of appending version differences at the end like RCS, SCCS interleaves editing operations among original document/source code and associates a pair of timestamps with each text segment specifying the lifespan of that segment. Versions are retrieved from an SCCS file via scanning through the file and retrieving valid segments based on their timestamps. Both RCS and SCCS may read extra text segments which are invalid for the retrieved (target) version, resulting in additional processing cost. For RCS, the total I/O cost is proportional to the size of the current version plus the size of changes from the retrieved version to the current version. For SCCS, the situation is even worse: the whole version file needs to be read for any version retrieval. RCS furtherly reduces version retrieval cost by maintaining an index for valid segments of each version; but it still has the problem that valid document segments may be stored sparsely among pages generated in different versions thus creating many page I/Os. Another missing feature of RCS and SCCS for XML documents is that they do not preserve the logical structure of the original document; this makes structure-related computation on the documents difficult and expensive to support.

3. *U*SEFULNESS *B*ASED *C*OPY *C*ONTROL (UBCC)
To control version retrieval efficiency, we propose a new technique for storing versioned document objects. This strategy guarantees that the total I/O cost is linear in the size of the target version, instead of the size of the total number of changes between the most current (latest) and the target versions.

3.1 Page Usefulness
For simplicity we view a document as a collection of objects. As the document evolves over time, objects are added, modified, and deleted. thus creating a sequence of document versions, namely, V_1, V_2, ... where V_i is earlier than version V_{i+1}. Initially we may assume that objects in the document's very first version are physically stored in few, nearby

disk pages. However, after a number of updates, objects of a specific version may be physically scattered around different disk pages. Moreover, since a page may store objects from different versions, when retrieving a specific version, a page access (read) may not be completely "useful". That is, some objects in an accessed page may be invalid for the target version. For example, assume that five document objects O_1, O_2, O_3, O_4 and O_5 are created and stored in data page P at version V_1. The size of these objects are 30%, 10%, 20% 25% and 15% of the page size, respectively. Let the evolving history of this data page P be the following:

1. At version V_2, object O_2 is deleted,
2. At version V_3, object O_3 is updated,
3. At version V_4, object O_5 is deleted, and
4. At version V_5, object O_1 is deleted.

Accordingly, the *usefulness* of page P during version V_1 is 100% because all of its contents are alive for this version. For the next version V_2, only 90% content of P is useful, since object O_2 is deleted at V_2. Similarly, P is 70% useful during version V_3, 55% useful during V_4 and only 25% useful after V_5.

Clearly, a page maybe more useful for some versions and less for others. Obviously, usefulness influences how well objects of a given version are clustered into pages: higher usefulness means that the objects for a version are stored in fewer pages. As a result, reconstructing the version requires accessing fewer pages. On the other hand, low usefulness implies higher I/O for version retrieval. We would like to maintain a minimum page usefulness over all versions. To achieve this, we apply a new technique based on copying objects from a page when this page's usefulness falls below the minimum usefulness allowed. However, this object copying procedure is controlled in a way that keeps the overall space linear in the number of changes in the version history. Setting the minimum page usefulness is a performance parameter of our approach.

3.2 Copying Procedure

Objects of the original document are stored by their logical order in the document. Figure 1 shows the storage of a sample document. In part (a) the first version of the document is shown, while pages P1, P2 and P3 of part (b) are the data pages storing the first version of this document. We have assumed that the sizes of document objects, Root, CH A, SEC D, SEC E, CH B, SEC F, SEC G, SEC H, CH C, SEC I, SECJ are 50%, 25%, 10%, 15%, 5%, 30%, 35%, 30%, 5%, 10%, and 5% of a data page size, respectively. Assume that we want to maintain a minimum page usefulness of 70%. Then pages P1 and P2 are well above the threshold for version V_1. Note that we allow the last page of a version to have usefulness less than the threshold (page P3 is only 20% useful). This is simply because there may not be enough objects in a given version to fill up the last page. However, as there is always only one last page the above exception does not affect the performance of our scheme.

Version V_2 is created by the following changes: *((delete SEC D) , (update SEC G with SEC G') , (delete SEC H) , (insert CH K after CH B) , (insert SEC L after CH K) , (insert SEC M after SEC J))* . The sizes of SEC G', CH K, SEC L and SEC M are 20%, 20%, 25% and 50% of a data page size, respectively. Hence, the logical order of objects in version 2 are: Root, CH A, SEC E, CH B, SEC F, SEC G', CH K, SEC L, CH C, SEC I, SEC J, SEC M. After applying these changes, Page P1 becomes 90% useful (SEC D is not part of version V_2), page P2 becomes 35% useful (since the original SEC G and SEC H are not part of V_2) and page P3 remains 20% useful (no change affected it). Then, pages P2 and P3 are *useless* for the second version and, thus, alive objects in P2 and P3 are copied into a new data page. Copied objects include CH B, SEC F , CH C , SEC I, and SEC J. Notice that even though page P3 is not full, its usefulness is only 20% and its objects are also copied. This is for preventing pages like P3 from accumulating in the document evolving process and increasing the I/O cost. Such copy happens only once for each version insertion.

After determining which objects need copying, the copied objects are inserted into new pages in their logical order as shown in Figure 1 (b). Both new objects and copied objects are stored in page P4 and P5 based on the linear order they appear in Version 2.

In addition, we also need to record the edit script for later version retrieval. The stored edit script (*UBCC script*), is : *(del(SEC D , 2), ins(CH B , 4 , P4), del(CH B , 4), ins(SEC F , 5 , P4) , ins(SEC G' , 6 , P4) , del(SEC G , 6) , del(SEC H , 6) , ins(CH K , 7 , P4), ins(SEC L , 8 , P4) , ins(CH C , 9 , P5) , del(CH C , 9) , ins(SEC I , 10 , P5) , del(SEC I , 10) , ins(SEC J , 11 , P5) , del(SEC J , 11) , ins(SEC M , 12 , P5)).* An UBCC script is derived by from the original edit script as follows:

- Each copied object is treated as a delete followed by an insert. For example, a delete operation is added for CH B and followed by an insert operation.
- Attach to each operation the position of its target object in the new version. For example, the position of SEC F in the new version is 5.

Notice that, the position for a deleted object is its position in the new version as if it was not deleted. For example, the position of SEC D is 2 in the new version if it was not deleted, so, the delete operation *del(SEC D)* has a position value of 2. We will explain how to use these position values to recover the total order of these objects in the next section.

Version V_3, is generated by the following changes: *((update SEC E with SEC E'),(delete SEC F),(insert SEC N after SEC G'), (insert SEC P after SEC L),(delete SEC I),(update SEC M with SEC M'), (insert SEC Q after SEC M')).* Here the sizes of SEC E', SEC N, SEC P, SEC M', and SEC Q are 35%,15%,45%, 30% and 29% of the page size, respectively. As a result, pages P1, P4, and P5 become 75%, 70%, and 10% useful, respectively. Thus, valid objects in P5, *CH C, and SEC J*, must be copied. New objects and copied objects are stored into new data pages P6 and P7 in their logical

```
          Version 1
------------------------------

  <Root>
    <CH A>
      <SEC D> ... </SEC>
      <SEC E> ... </SEC>
    </CH>
    <CH B>
      <SEC F> ... </SEC>
      <SEC G> ... </SEC>
      <SEC H> ... </SEC>
    </CH>
    <CH C>
      <SEC I> ... </SEC>
      <SEC J> ... </SEC>
    </CH>
  </Root>

          Version 3
------------------------------

  <Root>
    <CH A>
      <SEC E'> ... </SEC>
    </CH>
    <CH B>
      <SEC G'> ... </SEC>
      <SEC N> ... </SEC>
    </CH>
    <CH K>
      <SEC L> ...</SEC>
      <SEC P> ...</SEC>
    </CH>
    <CH C>
      <SEC J> ... </SEC>
      <SEC M'> ...</SEC>
      <SEC Q> ...</SEC>
    </CH>
  </Root>
```

(a)

```
                Data Pages
------------------------------------

New pages generated for Version 1

    +----------------------------+
 P1 | Root, CH A, SEC D, SEC E   |
    +----------------------------+
 P2 | CH B, SEC F, SEC G, SEC H  |
    +----------------------------+
 P3 | CH C, SEC I, SEC J         |
    +----------------------------+


New pages generated for Version 2

    +----------------------------+
 P4 | CH B, SEC F, SEC G', CH K  |
    | SEC L                      |
    +----------------------------+
 P5 | CH C, SEC I, SEC J, SEC M  |
    +----------------------------+


New pages generated for Version 3

    +----------------------------+
 P6 | SEC E', SEC N, SEC P       |
    +----------------------------+
 P7 | CH C, SEC J, SEC M',       |
    | SEC Q                      |
    +----------------------------+
```

(b)

```
                UBCC Script
--------------------------------------

E1 :

  ins(Root,1,P1), ins(CH A,2,P1),
  ins(SEC D,3,P1), ins(SEC E,4,P1),
  ins(CH B,5,P2), ins(SEC F,6,P2),
  ins(SEC G,7,P2), ins(SEC H,8,P2),
  ins(CH C,9,P3), ins(SEC I,10,P3),
  ins(SEC J,11,P3)

E2 :

  del(SEC D,3),
  del(CH B,4), ins(CH B,4,P4),
  del(SEC F,5), ins(SEC F,5,P4),
  del(SEC G,6), ins(SEC G',6,P4),
  del(SEC H,6), ins(CH K,7,P4),
  ins(SEC L,8,P4),
  del(CH C,9), ins(CH C,9,P5),
  del(SEC I,10), ins(SEC I,10,P5),
  del(SEC J,11), ins(SEC J,11,P5),
  ins(SEC M,12,P3)

E3 :

  del(SEC E,3), ins(SEC E',3,P6),
  del(SEC F,4), ins(SEC N,6,P6),
  ins(SEC P,9,P7), del(SEC I,10),
  del(CH C,10), ins(CH C,10,P7),
  del(SEC J,11), ins(SEC J,11,P7),
  ins(SEC M',12,P7), ins(SEC Q,13,P7)
```

(c)

Figure 1: A sample UBCC version file.

order in Version 3. The *UBCC script* for this version is: *(ins(SEC E' , 3 , P6), del(SEC E , 3), del(SEC F , 3), ins(SEC N , 6 , P6), ins(SEC P , 9 , P7), del(SEC I , 10), ins(CH C , 10 , P7), del(CH C , 10), ins(SEC J , 11 , P7) , del(SEC J , 11) , ins(SEC M' , 12 , P7), ins(SEC Q , 13 , P7))*

The outline of the insertion algorithm is shown in Figure 2. Two auxiliary hash tables are used to improve the version insertion process. The first hash table stores for each valid object of the current version the data page which contains the object. The second hash table keeps the usefulness of each page in the current version.

3.3 Version Retrieval

The total order of objects belonging to a given version are reconstructed in a *gap-filling* fashion based on the UBCC script. Let's take the sample version in Figure 1 as an example. We will explain the algorithm by describing how to reconstruct Version 3.

The reconstruction starts with retrieving the first object of Versions 3 from its UBCC script, E3. We try to find the first object in the first edit operation. However, we get a *gap* from the operation. The position value of the first operation, *del(SEC E,3)*, is 3. That means, we miss the first two objects and need to *fill the gap* from the previous version, Version 2. Recursively, we start to retrieve the first two objects of Version 2. This retrieval starts from the first operation, *del(SEC D,3)*, of E2. We get a gap again and need to retrieve two objects from the previous version, Version 1. ¿From E1, we find the first two objects of Version 1 and return them to Version 2. Recursively, these two objects are sent back to Version 3. When Version 3 receives these two records, it reads the data page P1 which contains these two objects, Root and CH A, and output them. Page P1 is not thrown away from main memory because it still contains one valid object, SEC E, for Version 3. The reconstruction of Version 3 continues from the previous stop point, *del(SEC D,3)* and the next object is the third object. Since the current operation is a delete, that means its target object is deleted from the previous version. Therefore, Version 3 requests the next object of Version 2 and it is ex-

```
INSERT() {
  Compute page usefulness based on the
    version editing script, Diff, and
    find pages need copying;

  Insert new objects and copied objects
    into new data pages based on their
    appearing order in the new version;

  Generate the UBCC script :
    For each (update Obj with New-Value)
      in DIFF replace the update
      operation, to two consecutive
      operations - (delete Obj),
      (insert New-Value);

  For each copied object insert an insert
    operation - ins(new_copy) - and a
    delete operation - del(old_copy) -
    into Diff by their logical position;

  Associate a position value for each
    operation in Diff. If the object is
    new or copied, its position value is
    its position in the new version.
    If the object is deleted, its position
    value is the same as that of the object
    right in front of it in the new version;

  Associate each insert operation with the
    page number of the page which contains
    its target object;

  Store UBCC script.
  Update hash tables.
}
```

Figure 2: Insert version difference

```
RECOVER(UBCC_SCRIPT S, INT request) {
    count = 0;
    while (count < request) {
        if (s is out of edit operation)
        {
          gap = request - counter;
          RECOVER(UBCC script of previous
                  version of S, gap);
          Append returned object list.
          count = count + gap;
        }
        else if (target position < the position
                 of current operation)
        {
          gap = position of current operation
                - target position;
          RECOVER(UBCC script of previous
                  version of S, gap);
          Append returned object list.
          count = count + gap;
        }
        else if (current operation is delete)
        {
          RECOVER(UBCC script of previous
                  version of S, 1);
          Nullify the retrieved object.
        }
        else if (target position = the position
                 of current operation) {
          Append the object of current operation
            to return_object_list.
          count++;
        }
    }
    return return_object_list;
}
```

Figure 3: Version retrieval algorithm

pected to be SEC E. To answer the request for next object, Version 2 needs to retrieve its third object, because its first two objects have been retrieved in the previous run. In a similar manner, Version 2 needs to request one next object from Version 1 because of its *del(SEC D,3)* operation. As expected, the returned record is *ins(SEC D,3,P1)* and it is nullified by the delete. However, at this point, the third object of Version 2 has not been retrieved yet. So another next-object request is issued from Version 2 to Version 1 and Version 2 gets back SEC E which is the third object of Version 2. Version 2 sends the *ins(SEC E,4,P1)* record back to Version 3 to reply its next-object request as expected. And this insert record is nullified with the *del(SEC E,3)* record of E3. The search for the third objet of Version 3 continues with checking the next edit operation in E3 which is *ins(SEC E',3,P6)*. And we find the third object because its position value is 3. This gap-filling procedure continues through the script E3 until all objects of Version 3 are retrieved.

This recursive gap-filling algorithm is used to reconstruct any version in the UBCC version file. Reconstructing Version N may need to involve UBCC script E_1, E_2, ..., E_N, but only useful data pages for each version are read. As a result, the requested version is reconstructed with few page I/Os. The algorithm is shown in Figure 3.

3.4 Complexity Analysis and Performance Evaluation

To reconstruct a version, only data pages which are useful for that version need to be read. Take Version 3 as an example, the total order of Version 3 objects are first recovered by applying the recover algorithm in Figure 3 on edit scripts E3, E2 and E1. The resultant edit operation list is : *ins(Root , 1 , P1), ins(CH A , 2 , P1), ins(SEC E' , 3 , P6), ins(CH B , 4 , P4), ins(SEC G' , 6 , P4), ins(SEC N , 6 , P6), ins(CH K , 7 , P4), ins(SEC L , 8 , P4), ins(SEC P , 9 , P7), ins(CH C , 10 , P7), ins(SEC J , 11 , P7), ins(SEC M' , 12 , P7), ins(SEC Q , 13 , P7)*. Pages pointed by the pointers in these records are read sequentially and objects are retrieved from those pages. Useless pages, such as pages P2, P3, and P5, need not be read because valid objects in those pages have been copied into useful pages. However, consider the way in which the objects of Version 3 are stored. These objects are stored in useful pages generated for Version 3 (page P6 and page P7), Version 2 (page P4) and Version 1 (page P1). Objects generated in the same version are stored in the order in which they appear in the version. Therefore, the retrieving process is actually a merge of these 3 ordered object lists. Merging 3 object lists will involve at most 3 pages at any instance during the process. That means that, with enough memory to hold 3 pages, each useful page of Version 3 needs to be read only once through the whole retrieval process.

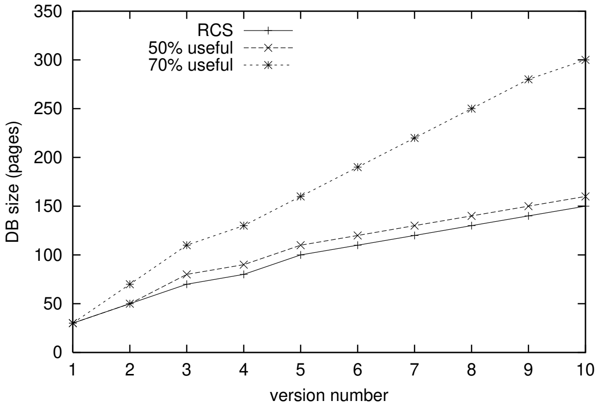

Figure 4: Total version file size

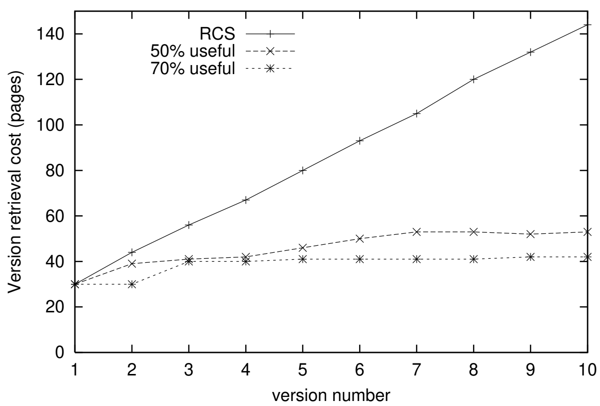

Figure 5: Version retrieval I/O cost

The above discussion is applicable to retrieving any version. That is, the I/O cost of reconstructing Version N , with N pages in memory, is that of

1. reading edit scripts $E_N \cdots E_1$, plus
2. reading useful pages of Version N whose total size is bound by S_i/U where S_i is the size of of Version N and U is required usefulness.

The size of edit scripts is significantly smaller than the total size of useful pages and can be ignored.

The size of the database is determined by two parts: new objects and copied objects. New objects include first-time inserted new objects and updated objects. Since deleted objects are not removed from storage, they do not affect the size of database. The new object part is bound by the *total number of changes* which include insert, update, and delete, that is, $O(S_{chg})$, where S_{chg} is the size of total changes of all time. For the copied object part, the number of objects that got copied once is bound by $U * S_{chg}$ where U is the required usefulness. Objects which got copied twice must be copied from those objects copied once already. Therefore, the total number of objects copied twice is bound by $U^2 * S_{chg}$. Similarly, the number of objects that got copied i times is bound by $U^i * S_{chg}$. Collectively, the total number of copied objects is bound by :

$$\sum_{i=1}^{\infty} U^i * S_{chg} \;=\; S_{chg}/(1-U)$$

Therefore, the total number of copied objects is bound by $O(S_{chg})$.

Combining these two parts, the size of database is bound by $O(S_{chg})$, that is, linear in the size of total object changes.

In Figures 4 and 5, version retrieval cost and total database size are compared for different usefulness requirement - 70% useful, 50% useful and 0% useful. The 0% case is the original RCS strategy. In the presented experiments:

- the size of each version is about 30 pages,
- in each version about 40% of the total document size is changed,
- changes are uniformly and randomly distributed among data pages,
- there are a total of 10 versions.

The experimental result in Figure 4 shows the size of the version file for these three usefulness requirements. The X-coordinate is the total number of versions stored in database and the Y-coordinate is the size of database (pages). The size of RCS version file is always the same as the evolving object base since it is never copied. The document evolving object base includes the first version plus all following updates. In the case of 50% usefulness, the version size grows linearly and about 30% faster than the growth rate of the evolving process. The case of 70% usefulness shows a similar trend, but the size here increases at a higher linear rate—i.e., about 80% faster than the evolving process.

The second graph shows the version retrieval cost measured as the number of pages read when retrieving a version. The X-Coordinate is the version number of the retrieved version and the Y-Coordinate is the total number of page read. The graph shows that the RCS strategy needs to read the whole evolving process prior to the target version; therefore, retrieving later versions is much more expensive than earlier versions. For example, retrieving Version 9 needs to read 132 pages which is 440% of its size. For the case of 50% useful, the I/O cost is just about 30% more than version size for earlier versions and the extra cost saturates at 50% for later versions which is significantly less than the RCS situation. The 70% case is similar in trend, but its I/O cost is even lower than the 50% case.

¿From these two graphs, we can clearly see that, with the controlled copying technique, the I/O cost is significantly reduced to become linear in the size of target version and the extra storage cost is only a constant percentage of the size of the document evolving object base.

4. CONCLUSION

In this paper we propose a new strategy—usefulness-based control $UBCC$—to manage evolving structured documents. $UBCC$ is I/O optimal and requires only minimal extra space. Compared with traditional strategies like RCS, the new strategy is much more efficient in version retrieving at the cost of some controlled copying. Furthermore our evolving document object base is a much better platform for supporting various kinds of document queries and computations.

We are currently investigating related issues, including querying and restructuring of versioned documents, and efficient generation for browser output and/or transport over the internet of versioned documents. These include the automatic generation of DTDs for versioned documents from the DTD of the unversioned document.

In addition we are investigating alternative storage techniques for the page list methods. These include extensions of the page-methods and of the versioned B+ trees from the temporal database indexing [3], [4], and [5]. Performance comparisons between various techniques will be discussed at the workshop.

5. REFERENCES

[1] Rochkind, Marc J., *The Source Code Control System*, IEEE Transactions on Software Engineering, SE-1, 4, Dec. 1975, pp. 364-370.

[2] Walter F. Tichy, *RCS–A System for Version Control*, Software–Practice & Experience 15, 7, July 1985, pp. 637-654.

[3] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer, *"On Optimal Multiversion Access Structures"*, Proceedings of Symposium on Large Spatial Databases, Vol 692, 1993, pp. 123-141.

[4] A. Kumar, V. J. Tsotras, C. Faloutsos, *Access Methods for Bi-Temporal Databases*, IEEE Transactions on Knowledge and Data Engineering, Vol. 10, No. 1, 1998, pp 1-20.

[5] V.J. Tsotras, A. Kangelaris, *"The Snapshot Index, an I/O-Optimal Access Method for Timeslice Queries"*, Information Systems, An International Journal, Vol. 20, No. 3, 1995.

[6] Robert A. Wagner and Michael J. Fischer, *The String-to-String Correction Problem* Journal of ACM, Vol. 21, No. 1, January 1974, pp. 168-173.

[7] D. Shasha, K. Zhang, *Fast algorithms for the unit cost editing distance between trees* Journal of Algorithms, 1990, 11:581-621.

[8] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, *Change Detection in Hierarchically Structured Information*, SIGMOD, 1996.

[9] WWW Distributed Authoring and Versioning (webdav). *http://www.ietf.org/html.charters/webdav-charter.html*