INDEXING OF MEDICAL XML DOCUMENTS STORED IN WORM STORAGE*

Naim Aksu, Taflan İmre Gündem

Computer Engineering Dept., Boğaziçi University
34342 Bebek, İstanbul, Turkey
gundem@boun.edu.tr

Abstract. Write-Once-Read-Many (WORM) storage devices may be used to store critical medical data to prevent them from easy modification. In this paper, we propose a novel indexing structure for encrypted medical XML documents stored in WORM storage. The proposed indexing that uses Generalized Hash Tree (GHT) expedites projection and selection operations on encrypted medical XML records stored in WORM storage. We implemented and tested the proposed system.

Keywords: XML, indexing, WORM storage, medical documents.

1. Introduction

Some medical records containing sensitive information such as organ donor information must be kept unaltered. Write-Once-Read-Many (WORM) storage devices [1, 2] may be used to enable the effective preservation of sensitive medical records. The basic property of WORM storage is not to allow updates on the data inserted.

Due to the large volume of records and increasingly stringent query response time [3], some form of direct access mechanisms such as indexes must be available in order to access the records [4]. However, if an index is not properly designed, the medical records stored in WORM storage can in effect be hidden or altered easily [5].

Currently, XML (eXtensible Markup Language) [6] is rapidly becoming a standard for data representation and exchange over the Web. Another important issue for medical records is the privacy of critical medical data. Thus, some sensitive medical data may be encrypted to satisfy the security requirements.

In this paper, we present a novel index structure and a partial encryption schema for encrypted medical XML documents stored in WORM storage. The proposed index structure expedites the projection and selection operations for the medical XML documents. In our proposed system, the index structure is also stored in WORM structure in an unaltered form like the original medical XML data. The index structure we propose expedites not only simple insert and search operations on text data, but also projection and selection operations on the XML documents. Our proposed system makes use of the generalized hash tree index structure in [5] to query the medical XML documents stored in WORM structure.

To the best of our knowledge, there is no work about indexing encrypted medical XML documents stored in WORM storages, in the literature. However, there is some research on efficient indexing of records stored in WORM structures. In the most recent study on this type of indexing, *fossilized index* [5] that uses a generalized hash tree structure to handle insertion and search operations on text data is introduced. The index proposed in [5] performs only basic key-word insertion and look-up operations on text data. It does not support selection and projection operations in XML.

The rest of the paper is organized as follows. In Section 2, we give an overview of the preliminary concepts. In Section 3, we present our proposed index structure and the encryption schema for medical documents. In Section 4, we present the performance evaluation of the proposed system. In Section 5, we have the conclusion.

2. Preliminary Concepts and Related Work

2.1. Indexing in Rewritable Storage and WORM Storage

Indexes expedite retrieval of records [5]. However, if an index can be manipulated, then all of the records are vulnerable to logical modification, since indexes in rewritable storage allow altering or hiding the content.

* This research is funded by Boğaziçi University research fund under grant number 04A108.

The most important property that an index stored in WORM storage must have is that once a record is written in WORM storage, both the index entry for that record and the path to that entry must be immutable. Once the insertion of a medical record into the index has been committed in WORM storage, the record must be guaranteed to be accessible through that index unless the WORM storage is compromised [5]. The next important property that an index for WORM storage must have is that the index must support incremental growth [5]. Especially indexes on medical data must scale to large collections of medical records. Also, the space overhead of the index must be acceptable.

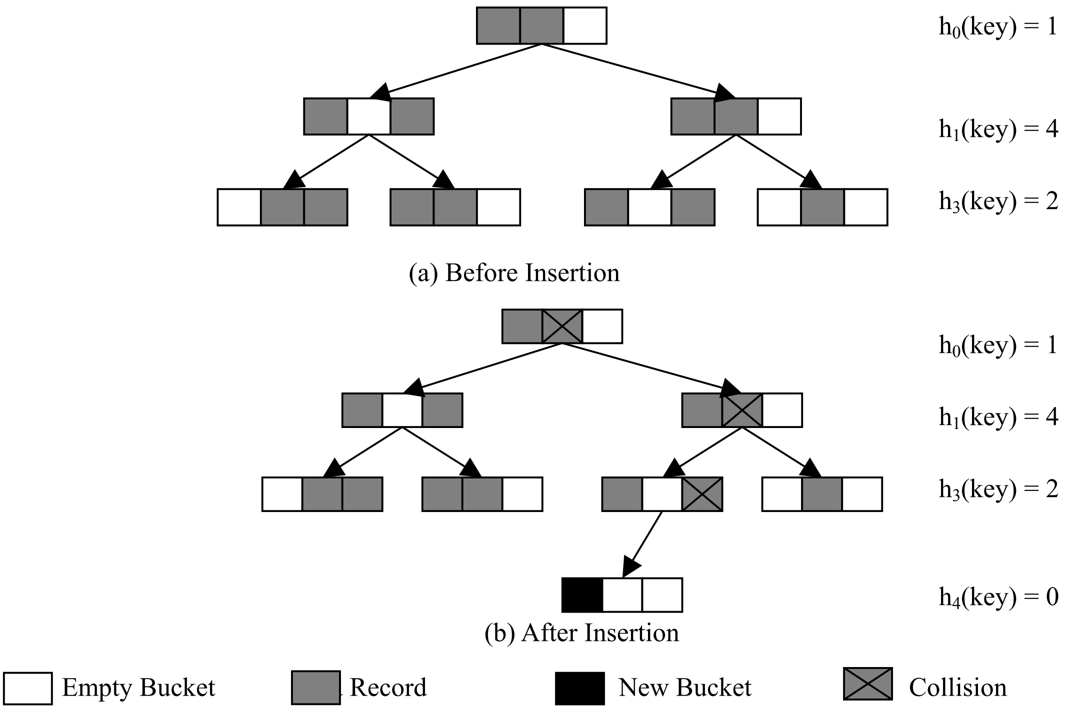
There has been much previous work on indexing structures for WORM storages. However, as it is explained in detail in [5], most of them do not meet the requirements of a *fossilized index*. Indexes stored in WORM storage are not suitable for indexing, if the index can be suitably manipulated [5]. Some of the approaches for indexes stored in WORM storages are write-once Btree [7], multi-version B-tree [8] and the append-only tree [9]. All of these approaches are vulnerable to tampering or are infeasible since they require storing each version of the tree.

2.2. Generalized Hash Tree

Generalized Hash Tree (GHT) [5] is the most recent index structure proposed for WORM storage and has all the properties (mentioned in the previous subsection) that an index for WORM storage must have. GHT data structure is a tree that grows from the root down to the leaves without relocating committed entries. Also it is balanced without requiring dynamic adjustments to its structure. GHT has an efficient dynamic hashing schema that does not require rehashing [5].

The basic properties of insertion and search algorithms for GHT are summarized in the following. Inserting or retrieving a record starts at the root node of the tree. If it is unsuccessful, the process is repeated at one or more of its children nodes. When a record cannot be inserted into any of the existing nodes, a new node is created and added to the tree as a leaf. At each level, the possible locations for inserting the record are determined by hashing the record's key field. Consequently, the possible locations for a record in the tree are fixed and determined solely by that record. Moreover, inserted records are never rehashed or relocated. The data structure is called *generalized* because it represents a family of hash trees. By using different parameters and hash functions, hash trees can have different characteristics. We use *Thin Tree* in our indexing system for encrypted medical XML databases.

In basic GHT, a *record* is represented by a key and a pointer to the actual data. A *bucket* is an entry in a tree node to store a record. A *tree node* consists of buckets and is the basic allocation unit of a tree. The size of a tree node may vary with its level in the tree. Let M = $\{m_0, m_1 \ldots m_i \ldots\}$ where m_i is the size of a tree node at level *i*. A *growth factor* k_i denotes that the tree may have k_i times as many buckets at level (i+1) as at level *i*. The growth factor may vary for each level. Let K = $\{k_0, k_1, \ldots, k_i, \ldots\}$ where k_i is the growth factor for level *i*. Let H = $\{h_0, h_1, \ldots, h_i, \ldots\}$ denote a set of hash functions, where h_i is the hash function for level *i*.



(a) Before Insertion

(b) After Insertion

Figure 1. Insertion of a record into a GHT where m =3 and k=2

Figure 1 illustrates the insertion of a record (whose key is designated as 'key') into a thin GHT tree where m = 3, and k = 2 for all levels. Here, we assume that (h_0, h_1, h_2, h_3) (key) = (1, 4, 2, 0) and use (level, node, index) to denote a bucket in a tree node. We first try to insert the record into the root node with first level hash function value h_0(key) = 1. However, the target bucket (0, 0, 1) is not empty. Thus, we try again at the next level, which is formed by the two children nodes of the root. h_1(key) = 4 indicates that the target bucket is (1, 1, 1) which is not empty either. Since the collision happens in node 1, its two children nodes form the next level hash table. But, the next attempt collides in the bucket (2, 0, 2). The fourth attempt succeeds since the tree node containing the target bucket does not exist. We allocate a new tree node and insert the record into the bucket (3, 0, 0). Intuitively, if the hash functions are uniform, the tree grows from the root down in a balanced fashion.

The search operation in GHT contains the steps to retrieve a record given its key. The process returns the record if it exists in the tree and NULL otherwise. When a target bucket is full, we test if its key matches the search key. If they match, then the record is found. Otherwise we follow the same process as in insertion to probe the next level of the tree. When a target bucket is empty or the target tree node has not been allocated, the search fails. Please refer to [5] for detailed information on GHT.

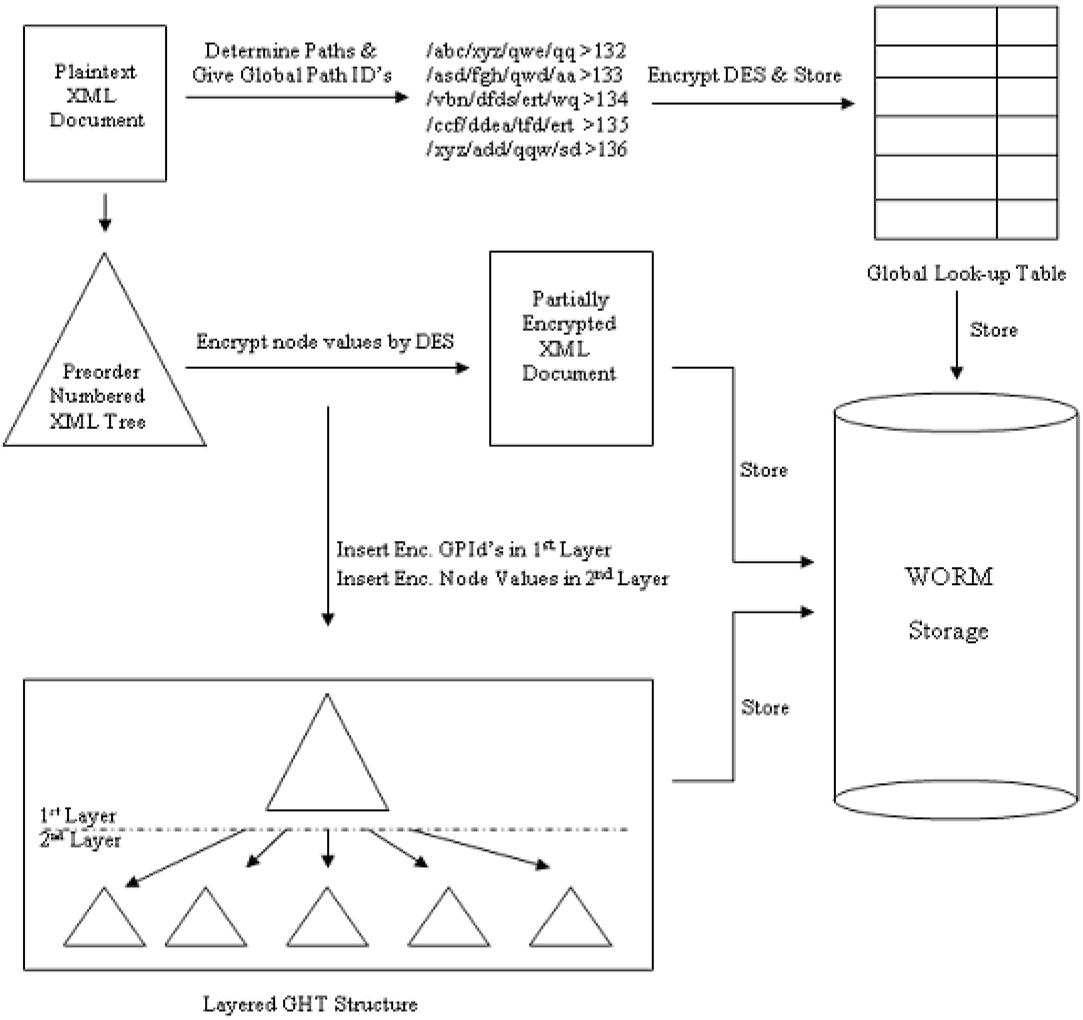


Figure 2. General layout of the complete system

3. Proposed System

The various stages in inserting a document into WORM storage and in processing a query using the proposed system are explained in this section. Figure 2 gives a general layout of the proposed system.

3.1. Inserting a New XML Document into WORM Storage

When a medical XML document (to be stored in WORM storage) is given as input, first the preorder tree representation of the XML document is constructed in the internal memory. Then the following sub operations are applied. 1. Global Path Indexing (GPI) of XML Document. 2. Preorder Traversing of XML

Document (Local Id Numbering). 3. Partial or Complete Encryption of XML Document by DES.

```
<medical-treatments>
        <medical-treatment>
            <patient-info>
                <patient-name>Pelin Korkmaz</patient-name>
                <patient-age>45</patient-age>
            </patient-info>
            <diagnosis-info>
                <disease-name>breast cancer</disease-name >
                <diagnosis-date>12.10.2003</diagnosis-date>
            </diagnosis-info>
            <medicine-info>
                <medicine-name>salsalate</medicine-name>
                <medicine-name>palifermin</medicine-name>
                <medicine-name>busulfan</medicine-name>
            </medicine-info>
        </medical-treatment>
        <medical-treatment>
            <patient-info>
                <patient-name>Ayhan Ersoy</patient-name>
                <patient-age>54</patient-age>
            </patient-info>
            <diagnosis-info>
                <disease-name>tuberculosis</disease-name >
                <diagnosis-date>03.01.2004</diagnosis-date>
            </diagnosis-info>
            <medicine-info>
                <medicine-name>amoxicillin</medicine-name>
                <medicine-name>bisacodil</medicine-name>
            </medicine-info>
        </medical-treatment>
    </medical-treatments>
```

Figure 3. A sample medical XML document

Global Path Indexing of an XML Document

After constructing the preorder tree representation of the given XML document, all possible paths which end at a leaf node in the XML tree are given unique identifiers (ID s). Two paths with the same label have the same path ID. The path IDs are globally unique among all the documents in the WORM storage. For each of the XML documents stored in the WORM structure, all possible paths and their corresponding global path ID s are stored in the *look-up table* residing in WORM storage. Only insertions (and no modifications) are performed on the look-up table. A record in the global look-up table is in the form [Path Name, Global Path Id]. Due to the security of sensitive medical data, the *Path Name* column of the look-up table is encrypted using DES encryption algorithm. The *global path ID* values of the document are inserted into the 1st layer GHT structure. According to the insertion algorithm of GHT storage structure, the inserted key value is hashed by a hash function $h_i(x)$ for level *i*.

Preorder Traversal of XML Document

The next operation is to give local ID numbers to each element, attribute and leaf node in the constructed XML tree. This numbering operation is done by using the preorder traversal algorithm. Unlike the global path IDs, the scope of these IDs is the XML document not the whole WORM storage. The reason for numbering all the nodes is that we index both structure and content. After giving the *local ID* numbers for all the nodes in the XML tree, an additional number is computed for each node. As a result, each node has a number pair (n_1, n_2). The second number computed, n_2, is the total number of descendant nodes of node n_1 (where n_1 is the local id number of the node). The second number, n_2, in the number pair of an XML tree node is used in the encryption phase. Our encryption schema supports partial encryption. For example, if a node with value pair <3, 4> has an encryption flag set to *TRUE*, then the 4 nodes starting from the node with n_1 value equal to 3 (i.e. nodes with n_1 values 3, 4, 5, 6, 7) will be encrypted.

Figure 4 illustrates the XML tree of the document in Figure 3 after the local parsing sub operation. In Figure 4, some abbreviations are used due to space constraint. MTs stands for Medical Treatments, MT stands for Medical Treatment, PI stands for Patient Info, DI stands for Diagnosis Info, MI stands for Medicine Info, PN stands for Patient Name, PA stands for Patient Age, DN stands for Diagnosis Name, DD stands for Diagnosis Date, and MN stands for Medicine Name. The leaf node values stand for the values in our example XML document. The second number, n_2, in the number pair of a leaf node is always zero. Thus it is not shown in the figure.

Partial Encryption Phase

Our encryption algorithm uses DES with key length 112 bits. The proposed encryption schema permits partial encryption. Also, it enables us to process queries on encrypted XML data. An extra attribute called "encryption flag" is added to the tag of the element to be encrypted and is set to "TRUE" such as <diagnosis-info **encryptionFLAG = 'TRUE'**>. Let us assume that the number pair of *diagnosis-info* element is <8, 4>. This means that the start and end points of the encrypted part are 8 and 12 (8+4=12). In other words, nodes with local ID 8, 9, 10, 11, and 12 will be encrypted.

Encrypted data are stored as CDATA in XML. After encrypting the marked nodes of the XML tree, an "*encryption-data*" element is inserted into the encrypted part in the XML document. Also, start and end point values are written between the *encryption-data* tags. Then, the XML document mentioned in the preceding paragraph becomes as follows. <encrypted-data start = "8" end = "12"> *encrypted data (DES algorithm used)* </ encrypted-data >. After encrypting the medical XML document using the proposed encryption, the encrypted document is stored in the WORM storage. The next operation is to insert the *global path ID* values and the DES encrypted leaf node values into our proposed index structure (as explained in the following).

3.2. Proposed Indexing System

The proposed indexing system contains a two layered GHT data structure. The first layer of GHT contains the global path ids of the inserted documents and is accessed using the global path ids as the key values. The global path ID of each numbered path is inserted into the 1st layer GHT. The 2nd layer GHT contains several small-depth Generalized Hash Trees which are pointed to by the nodes of the 1st level GHT. This means that each 2nd layer GHT contains the encrypted leaf node values of a specific path in the 1st level GHT. Each global path ID in the 1st level GHT has one small-depth GHT among the 2nd layer General Hash Trees for storing its contents.

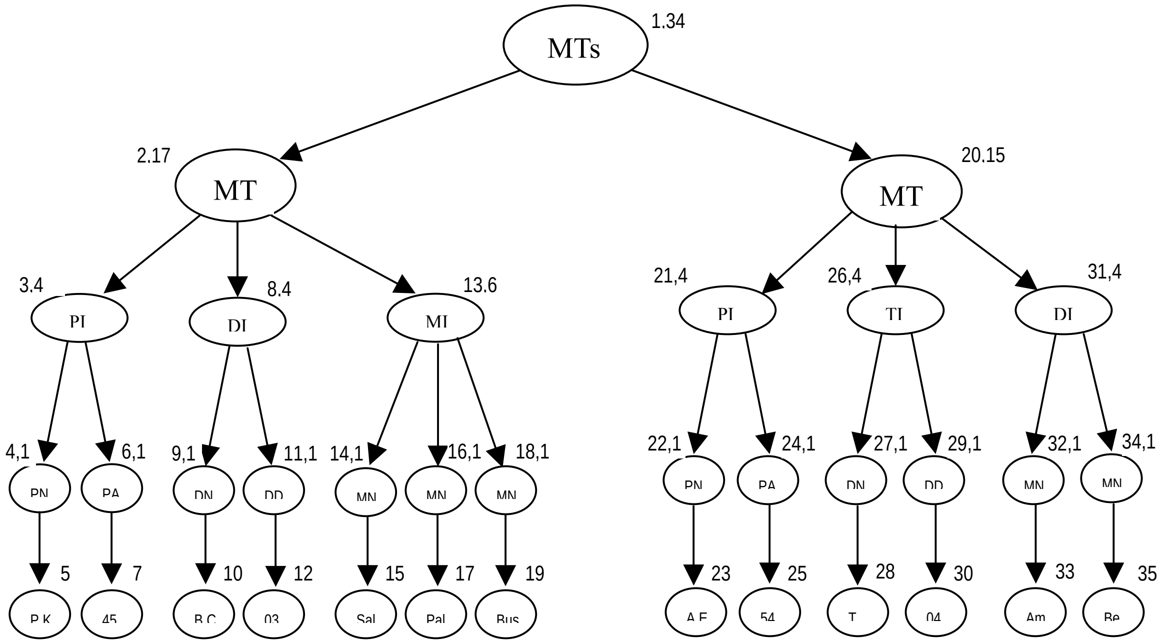
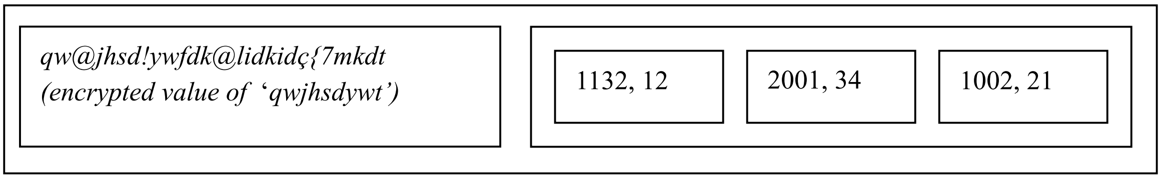


Figure 4. Local ID numbering of the XML tree of the document in Figure 3


qw@jhsd!ywfdk@lidkidç{7mkdt
(encrypted value of 'qwjhsdywt')

1132, 12

2001, 34

1002, 21

Figure 5. Structure of an inserted record in 2nd layer GHT

Each GHT when considered in isolation in our layered structure has the same properties (i.e. buckets, nodes, hash function and extension factor properties) as those of the basic GHT described in [5]. Our layered structure is designed considering the structure of XML documents. XML documents contain both the structure and the data (content) together in the same document. In our layered index, the first layer manages the structural part (i.e. paths) while the second layer manages the data (contents) associated with each path. Let us assume we have the path department/ patient/name in the first layer. In the 2nd layer corresponding to this we have all the instances of this path. Since there may be a lot of patient names, we have the 2nd level GHT to expedite finding a specific patient name. In the 1st level GHT we also store all the document ids and offsets for all the instances associated with a path. For example, for the path department/patient/name the document id and offsets for all the patient names are stored in the first level (i.e. where they are stored in the document are present but not the actual names). This expedites the projection operation.

Insertions in the Indexing System

First, the global path IDs of paths in the given medical XML document are inserted into the 1st layer GHT data structure according to the basic insertion algorithm of GHT. Then the encrypted leaf node values are inserted as explained in the following. To insert an encrypted leaf node value into the 2nd layer GHT, the global path ID of the owner path is searched in the 1st layer GHT. If the search fails, then the global path ID is inserted into 1st layer GHT. If the search is successful, then the pointer value of this bucket is checked. If the pointer of the bucket points to a 2nd layer GHT, it means that an encrypted leaf node value has been already inserted for this path. Then the DES encrypted value of the leaf node of this path is inserted into the pointed 2nd layer GHT data structure. On the other hand, if the pointer to the 2nd layer GHT is NULL, it means that no leaf node value for this path has been inserted into the 2nd layer GHT yet. In this case, a new 2nd layer GHT root is created in the WORM structure, and the encrypted leaf node value is inserted into the 2nd layer GHT. Also the document ID and a local ID (offset) values are inserted into the 1st layer in order to expedite projection operation. All these steps are repeated for every leaf node value associated with this path, and also for every path in the XML document.

The insertion algorithm, Algorithm 1, uses the following functions. The *EXTRACT* function separates the path value from the leaf value. For example, given the query *(/medical-treatments/medical-treatment/medicine-info[medicine-name= 'palifermin'] /medicine-name)*, *EXTRACT* separates the path value *"/medical-treatments/medical-treatment/medicine-info /medicine-name"* from the leaf value *'palifermin'* . The *ENCRYPT* function encrypts a string with DES according to the encryption flag value, and returns the encrypted value. The *RETRIEVE* function returns the global path ID value of a path, by accessing the look-up table. The *INSERT* and *SEARCH* functions are the basic insertion and search functions of the basic GHT. The search function of our proposed system is used for projection and selection operations as explained in the next section. The *CHECK* function controls if the pointer value of a bucket in the 1st layer GHT is NULL. The *CREATE* function allocates new space in the WORM structure for the root of the new 2nd layer GHT. Finally, the *POINT* function points to the root of the newly created 2nd layer GHT.

Algorithm 1 - LAYERED-GHT-INSERT

```
1. EXTRACT the path value of the leaf node
2. ENCRYPT the path value with DES
3. RETRIEVE the global path ID of the encrypted
     path from look-up table
4. SEARCH the global path ID in the 1st layer GHT
5. IF the global path ID is NOT found THEN
     {SEARCH returns FAILURE}
     INSERT the global path ID into the 1st layer GHT;
     also insert the array of
    document ID and a local ID value pairs (to expedite
    projection operation)
ELSE IF the global path ID is found THEN
     {SEARCH returns the BUCKET}
     CHECK the 2nd layer GHT pointer value of the
         bucket
     IF the pointer is NULL THEN
         CREATE the root for a new 2nd layer GHT
         POINT to the root of  the 2nd layer GHT from
         the 1st layer GHT
     END IF
     ENCRYPT the leaf node value with DES
     INSERT the encrypted value into the 2nd layer
     GHT
     insert the array of document ID and a local ID
     value pairs into 1st layer
     (to expedite projection operation)
     Return SUCCESS
END IF
```

The inserted record of the encrypted leaf node value in the 2nd layer GHT contains the encrypted value of the leaf node and an array. The array consists of records where each record contains a document ID and a local ID. The local ID is an offset which is used in expediting the search with a document ID. A sample record is given in Figure 5. The record in Figure 5 states that the encryption of *'qwjhsdywt' (i.e. qw@jhsd!ywfdk@lidkidç{7mkdt)* is present in documents with document IDs 1132, 2001, and 1002. The offset values of these documents are 12, 34 and 21, respectively.

Searching in the Indexing System

Search operation finds the address of the document for a specific value associated with a path. First, the value searched and its path are encrypted using DES. Then, the global path ID of the path is retrieved from the global look-up table. The global path ID is searched in the 1st layer GHT (using the search algorithm of basic GHT). If the ID is found, then the 2nd layer GHT pointed to by the global path ID is searched. The search in the 2nd layer GHT is materialized using the basic GHT search algorithm.

Algorithm 2 - LAYERED-GHT-SEARCH

```
Steps 1 to 4 in Algorithm 1 are also used here.
IF the global path ID is NOT found THEN {SEARCH
returns FAILURE}
    Return FAILURE
ELSE IF the global path ID is found THEN
    {SEARCH returns the BUCKET}
    CHECK the 2nd level GHT pointer value of the
    bucket
    IF the pointer is NULL THEN
        Return FAILURE
    ELSE IF the pointer is NOT NULL THEN
        ENCRYPT the leaf node value with DES
        SEARCH the encrypted leaf node value in the
        2nd level GHT
        IF it is NOT found THEN {SEARCH returns
        FAILURE}
            Return FAILURE
        ELSE IF encrypted value is found THEN
            Return the BUCKET
END IF
        END IF
    END IF
```

If the record is found in the 2nd level GHT, then the function returns the bucket. The *document ID (let us call x)* and the *local ID (let us call y)* in the bucket may be used to locate the record in document with *document ID x*. The *local ID,* y, of the record is used to decrypt the appropriate part of the encrypted medical XML document. The comparison operations for finding the appropriate part of the encrypted XML document are done without decrypting the document.

When the location of the searched data in the medical XML document is found with the comparison operations on the encrypted data, then only this part of the whole medical XML document may be decrypted. Next the essential data can be retrieved from the partially decrypted part of the XML document specified with *document ID x*.

Projections and Selection operations

Processing a projection operation is equivalent to retrieving all the data for a specified path. The following is an example of projection: *List the disease names diagnosed so far in the hospital* (*/medical-treatments/ medical-treatment/diagnosis-info/disease-name).* In processing a projection query, first the path value in the query is encrypted with DES algorithm and then the corresponding global path ID information is retrieved for that path from the look-up table. Next, the global path ID is searched for in the 1st layer GHT with the basic search operation of the GHT (using the global path ID as a key). If the key is found in the 1st layer GHT, the *document ID* and *local ID* value pairs of all elements that have the path that we searched for are found in the 1st layer GHT bucket. Let us assume we obtain the *document ID* and *local ID* pairs [5, 10] and [5, 28]. Number 5 is the document ID value of the document in the WORM structure. Then, these values are used to access the partially encrypted document with Document ID 5. Local ID offset values are used for fast localization of data on the encrypted data (without decryption) in the found document.

Performing a selection operation is explained in the following. Let us consider the query: List the medicine names used in treatments together with medicine 'palifermin' (/medical-treatments/medical-treatment/medicine-info[medicine-name='palifermin'] /medicine-name).

The global path ID of this path, say 197, is retrieved from the global look-up table. The path 197 is searched for in the 1st layer GHT with the basic search operation of the GHT data structure. If the key is found in the 1st GHT, the pointer information of this bucket is retrieved, and the 2nd layer GHT that this bucket points to is accessed in the WORM storage. Let us assume that key 197 is found in the 1st layer GHT, and the 2nd layer GHT that the bucket with global path ID 197 points to is accessed. Then, another search operation is performed in the 2nd layer GHT. For that purpose, the 'palifermin' value is encrypted using the DES algorithm, and the encrypted value is searched for according to the basic GHT search algorithm. When it is found in the 2nd layer GHT, the document ID and local ID value pair of the found bucket is retrieved. Let us assume the value pair found is [5, 17]. Then, the corresponding part of the document, with *document ID* **5**, is decrypted. Let us assume the corresponding part is the encrypted part between the starting local ID = 13 and ending local ID = 19. This interval is decrypted and the decrypted medicine names are obtained after the decryption phase in the actual XML document.

A way to process a join operation involves performing two projection operations (one for the left and the other for the right hand side of the = sign). Since the proposed system expedites the projection operation, it also indirectly helps expedite the join operation. An example to a join operation would be "*List the diagnosis dates of diseases which had a surgical operation" (/medical-treatments/medical-treatment/ diagnosis-info [disease-name = /surgery-operations/ surgery-operation/disease-info/disease-name]/diagnosis-date).* Processing this query involves processing the projections associated with paths */medical-treatments/medical-treatment/diagnosis-info/disease-name* and */surgery-operations/surgery-operation/disease-info/disease-name.*

4. Performance Analysis

We have implemented the proposed indexing system and its associated algorithms given in Section III to measure their performance. The programs for the implementation are coded in Java 1.5, and tested on a pc that has an Intel(R) Pentium(R) 4 CPU with 3.00 GHz, and 2 GB internal memory.

We created a synthetic dataset that contains medical XML documents for various departments in a hospital. The dataset contains 60,000 medical documents from 15 departments. Each department has 4,000 documents. Each document contains 50 records, and each record contains approximately 12 leaf node values and 12 path values. This means that, inserting a document corresponds to inserting 600 leaf node and 600 path values, in other words, 1200 items. Thus, the number of all items in the dataset which contains 60,000 documents is 72,000,000. The total size of the complete dataset is approximately 1.22 Gigabytes.

4.1. Time and Space Performance Results

The space to store the layered GHT for the amount of data (1.22 GB) in WORM storage is approximately 0.6 GB. Figure 6 compares the time performance of search, projection, and selection operations on the inserted 72,000,000 items. The results in Figure 6 are the average results of many different search, projection, and selection query processing times.

The search operation is the fastest one since the search operation only tries to find the searched value in the layered GHT. The other operations take more time because of the time losses resulting from retrieving the query results from the specified XML documents in WORM storage. Also, the projection operation is faster than the selection operation. This is mainly due to saving all the document ids and offsets for a specified path in the 1st layer GHT buckets. By the help of this property, there is no need to parse all the nodes in the 2nd layer GHT which is pointed to by the path in the 1st layer GHT.

4.2. Sensitivity to Encryption

We tested the effect of encryption on insertion and selection. In the tests, we considered the case where we have only the content information encrypted and also the case where we have both the path and content information of the documents encrypted. The look-up table, which stores the path information, is also encrypted in the later case. Approximately half of the dataset is encrypted for the test runs.

The insertion of 72 million items with only content encryption and with path and content encryption takes up to 35% and 47% more time, respectively, than that without encryption. The insertion of 72 million items into the layered GHT and the WORM storage with content encryption takes 8% more space than that without encryption. If both path and content are encrypted, then up to 21% more space is used than that for the unencrypted version. In search operation, if encryption is applied to both path and content data, the time loss is around 5% while the time loss in content encryption alone is around 3%. In selection, if encryption is applied to both path and content data, the time loss may go up to 24% while the time loss in content encryption alone is 17%.

4.3. Comparisons with B-Tree

In this section, we compare the indexing system that we propose (write-once layered GHT) with the rewritable B-tree. The layered GHT structure has an advantage in space efficiency over the B-tree that exceeds 7%. The time performance comparisons of the search operation for both indexing structures are summarized in Figure 7.

B-tree performs well but is vulnerable to logical tampering of records [5]. Layered GHT satisfies the requirements of an index for WORM storage and still its time performance is equal to or up to 9% better than that of B-tree. We can not compare projection and selection performance of our layered GHT with that of a B-tree since B-Tree does not support these operations.

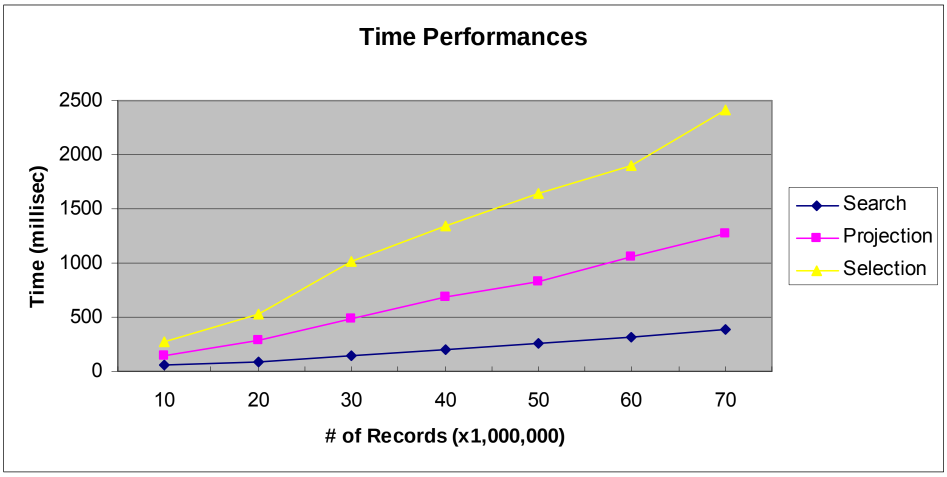


Figure 6. Average processing times of search, projection and selection operations

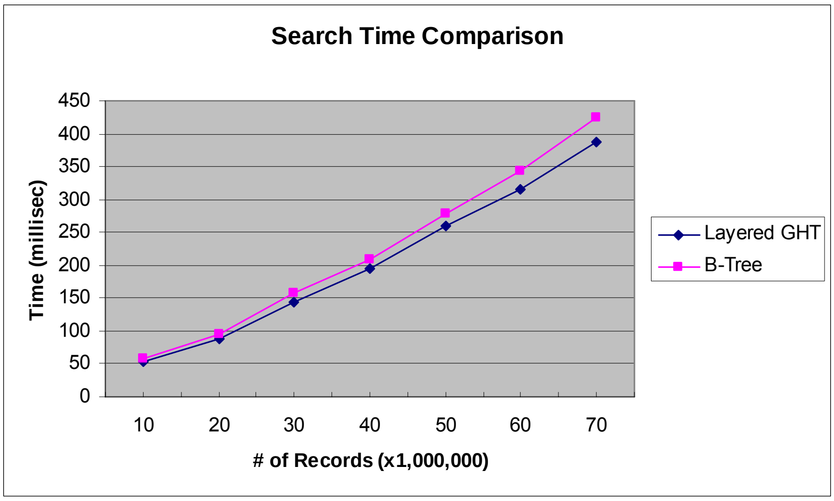


Figure7. Time performance comparison with B-tree for search operation

5. Conclusion

Some medical records have to be kept untampered due to regulations. In this paper, we presented a novel indexing structure for encrypted medical XML documents stored on WORM structures. We showed how our indexing is used to process projection and selection operations on encrypted medical XML documents kept in WORM storage. We have tested the indexing system and its associated algorithms. The experiments demonstrated that the proposed system performs slightly better than a B-tree which is not suitable for WORM indexing.

To the best of our knowledge, we are not aware of any other indexing method for expediting projection and selection operations on encrypted or plaintext medical XML documents in WORM storage, in the literature.

REFERENCES

[1] EMC Corp. EMC Centera Content Addressed Storage System. 2003, *http://www.emc.com/products/systems/centera_ce.jsp*.

[2] BM Corp. IBM TotalStorage DR550. 2004, *http://www-1.ibm.com/servers/storage/disk/dr*.

[3] Cohasset Associates, Inc.. The Role of Optical Storage Technology. White Paper, 2003, *http://www.nexstor.co.uk/Resources/cohasset.pdf.*

[4] **H. Wang, S. Park, W. Fan, P.S. Yu.** ViST: a dynamic index method for querying XML data by tree structures. *Proceedings of the* 2003 *ACM SIGMOD International Conference on Management of Data*, 2003, 110-121.

[5] **Q. Zhu, W.W. Hsu.** Fossilized index: the linchpin of trustworthy non-alterable electronic records. *Proceedings of the* 2005 *ACM SIGMOD International Conference on Management of Data*, 2005, 395-406.

[6] World Wide Web Consortium. Extensible Markup Language (XML) 1.1. *April* 2004, *http://www.w3.org/TR/2004/REC-xml11-20040204/.*

[7] **M. C. Easton.** Key-Sequence Data Sets on Indelible Storage. *IBM Journal of Research and Development, Vol.*30, *Issue* 3, 1986, 230-241.

[8] **D.R. Stinson.** Cryptography: Theory and Practice. *CRC Press,* 2*nd edition*, 2002.

[9] **P. Rathmann.** Dynamic Data Structures on Optical Disks. *Proceedings of the* 1*st International Conference on Data Engineering*, 1984, 175-180.

Received July 2008.