ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2010, Vol. 39, No. 4

PARALLEL PROCESSING OF ENCRYPTED XML DOCUMENTS IN DATABASE AS A SERVICE CONCEPT

Ozan Ünay, Taflan İmre Gündem

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

Abstract. Data-owners who possess large confidential data are looking for methods to securely store and efficiently query their data when using database services. In this paper we focus on "database as a service" concept and propose a methodology that securely and efficiently queries encrypted XML documents using parallel processing. The content of the database is not revealed to the service provider and immunity against attacks is also provided. The query execution is performed at the service provider side that uses several computation nodes. Proposed methodology is tested and the performance results are presented.

Keywords: XML, parallel processing, database as a service.

1. Introduction

In business, companies usually prefer to concentrate on their own proficiencies and outsource the rest of the work. This situation has brought the service concept to information technology. In this paper, we focus on "database as a service" concept which makes handling the database easier for the client but brings some privacy issues into consideration [1, 2, 3, 4]. Regardless of the database type used, the data kept in untrusted third parties have to be secured. To overcome this security issue, the databases are encrypted and the keys are not disclosed to the service provider. On the other hand, unrevealing the key to the service provider brings forth the problem of querying the encrypted database.

Extensible Markup Language (XML) is a widely used standard for creating documents. Numerous firms store their data in XML format. It is expected to find considerable amount of sensitive information in XML format [5]. The clients, who outsource their confidential data, need to be sure that their data are secure and visible neither to attackers nor to database service providers. One of the solutions suggested to resolve the data security problem in XML databases is using "access control mechanisms" [6, 7, 8, 9, 10]. But using access control mechanisms alone is usually not sufficient. The attackers, who once break into the system, may gain access to private information. The weak point of access control mechanism is that either the communication channel or the storage itself may be insecure, e.g. the hard disk may be stolen. Therefore, something more than an access control mechanism is required.

Encryption plays an important role in database security. In order to provide reliable encryption, encryption key should only be known by the data owner. The whole database should be a black box for the service provider. The administrative issues such as database backups and space managements should not be affected by the fact that the database is encrypted or not. The critical question to be drawn now is how the service provider is going to answer user queries without decrypting the database and without knowing the content of the database. Some research has been done on this subject. Mainly the problem can be handled by maintaining indexes at server side and/or at client side. The techniques in the literature also try to minimize the effort for data owner and give most of the work to the service provider. However, this has to be accomplished in such a way that the overall system security is high. Therefore querying encrypted XML databases without giving the key to the service provider is a big challenge.

There are models in the literature which propose parallel processing of unencrypted XML documents. The main working principle of these systems is dividing the processing load at the service provider into subgroups. These subgroups process XML document in parallel and consequently query processing time is shortened. In our work, we propose to combine parallel processing techniques and querying encrypted XML document techniques as a solution for securely storing encrypted XML documents in a third party service provider and efficiently querying the data without revealing the content. To accomplish this, we modified existing indexing techniques in the literature

in a particular way in our proposed system. The indexing techniques are integrated with parallel processing so that the load on the service provider is divided into multiple computation nodes. The computation nodes can be either single processor-distributed servers or multiprocessor-single server.

In the next section, we first give the necessary preliminary information. In Section 3, we present our methodology and show that the data stored at the service provider in the proposed methodology are secure and defensive to attacker's statistical analysis. In Section 4, we give the performance results.

2. Preliminaries

Encryption is a common method for providing confidentiality of databases. Research on encryption started with key management [11] and continued with the development of the techniques which are used for efficiently searching keywords based on encrypted textual strings [12]. Independent of the database type, the naive way of encrypted query processing is sending the encrypted database totally to the data owner. However this approach is not appropriate for large databases because of the fact that decryption and query processing responsibility are at the data owner side which may have limited processing capabilities. Moreover, data transportation is costly. A novel technique using bucketization and partitioning is proposed in [13]. The main idea is to map the plaintext values to ciphertext values by splitting the plaintexts in the domain into some partitions and giving them bucket ids. The success of this technique is due to the mapping function of the bucket ids that uses order preserving encryption functions [14]. As a result, the range queries can successfully be supported. In [15], mathematically well defined order and distance preserving encryption functions are used rather than partitioning techniques to encrypt the database. The proposed computing architecture is efficient in the sense that for some query types, query processing can be completed at the server without having to decrypt the database. One future work proposed in [15] was to handle SQL queries with arithmetic expressions and aggregate functions as well as complex SQL queries with nested subqueries. This is accomplished in [16]. The authors of [16] present query execution strategies for the mentioned types of queries. They also quantify additional costs incurred in executing these queries. In [17], a hash-based method suitable for selection queries is given. The index is maintained at the server side. The algorithm given in [17] provides a balance between efficiency and security. In [18], an algorithm for determining optimal bucket size for encrypted query processing is proposed.

2.1. The general architecture of encrypted query processing of XML documents

To provide a worthwhile service, most of the work load should be at the service provider side when a query is evaluated. Since the service provider does not have the access to the decrypted database, the client is supposed to give sufficient amount of clues to the service provider in order for it to return the correct encrypted data. However these clues should not enable the service provider to guess the structure of the whole database. The clues are generally given by maintaining crypto-indexes either at the client or at the server side.

The general architecture of encrypted query processing is as follows. The user creates a query which is then translated into its encrypted form by the query translator at the client side. The rules of encryption are determined by the client and given to the query translator. After the query becomes secure enough not to show the structure of the XML database, the service provider answers the query by some predefined rules that are determined at the server side. The result returned by the service provider is not the exact result that the user wants. It is a superset of the actual result set. The client decrypts the results and post filters the results in order to get the actual result. It should be noted that the client should have some processing capability in order to post process the results.

Some papers in the literature mention architectures different from the one explained above. For example in SymCrypt project, a number of messages should be exchanged between the server and the client in order to get the results.

2.2. W3C encryption standard

W3C specifies standards for encrypting XML [19, 20]. According to the mentioned standards, the tags and the contents that are going to be encrypted are replaced with a string called the Encrypted Data element. There are four subelements of Encrypted Data. (a) Encryption method which indicates the encryption algorithm and the parameters of the specified algorithm. (b) Key Info which indicates the key name but not the value. (c) Cipher Data which contains cipher value as subelements that indicate the encrypted element together with their content. (d) Encryption properties which contain additional information related to decrypting of Encrypted Data.

2.3. Advanced encryption standard

Advanced Encryption Standard (AES), which is adopted as an encryption standard by US government, is widely used worldwide and took the place of its predecessor Data Encryption Standard (DES). AES is fast, easy to implement and requires little memory.

There are mainly four steps in this standard. 1) SubBytes step in which each byte in the array is updated using an 8-bit substitution box. 2) ShiftRows step in which the bytes in each row of the state is cyclically shifted by a certain offset. 3) MixColumns step in which the four bytes of each column are combined using an invertible linear transformation.

4) AddRoundKey step in which a subkey which is derived from the main key is combined with the state.

2.4. Attack types

There are two main types of attacks that a database can face [3]. The first one is called frequency based attack and is possible if the attacker can find some number of matches between the cipher text and plain text values.

The attacker must know the exact frequency of occurrence of the domain values to be able to perform this type of attack. To give an example, let us assume that we have a hospital database and Michael Mackson is one of the patients. Let us also assume that it is known that Michael Mackson is the only patient who had 5 plastic surgery operations. From this, the attacker can infer that Michael Mackson corresponds to the encrypted value that occurs 5 times. If the attacker can find several matches like this, then it is possible for him to guess the encryption key.

Another attack type is "size based attack". If the length of the plain text determines the length of the cipher text, then attacker can eliminate the candidate databases whose lengths do not match and find the original database which corresponds to the encrypted database.

2.5. Index types

The structural index and the value index are the types of the index structures that are usually used in encrypted XML documents. Structural index is used for determining whether the path in the query matches any of the paths in the XML document, whereas the value index is used for checking the constraints in range queries. These indexes can be maintained either at the server side or at the client side.

2.5.1. Maintaining indexes at the server

There is a well-known indexing structure which is used in indexing XML documents. In this structure, each node is given a sequence number. The sequence numbers start from 1 and incremented by 1 for each node. The sequence number of the opening tag of a node represents the left bound of a node and the sequence number of the closing tag of a node represents the right bound of a node. The general rule for this structure is that “for a parent node p and child node c, p.leftbound<c.leftbound and p.rightbound>c.rightbound. The drawback of this structure is that whole tree has to be renumbered in case of insertions. This problem can be solved by leaving empty spaces when numbering the nodes.

In order to encompass the hierarchical structure of the XML documents, the structural index just explained is modified in [3]. Discontinuous structural index (DSI) is the name of the index introduced in [3]. In DSI, the interval (0, 1) is assigned to the root. The children are assigned sub intervals of their parents’ interval. The intervals of the children are determined by an algorithm at run time. Thus the structure of XML is hidden from the server.

The value index in [3] has an order preserving encryption with splitting and scaling (OPES). Splitting and scaling are used to prevent frequency based attacks. The main purpose of splitting and scaling is to change the frequency distribution of the encrypted data values in the value index so that they are different from the frequencies of the original values.

The main contribution of the approach in [3] is allowing the execution of range queries at the server side by employing order preserving encryption with splitting and scaling. One of the limitations of OPES is that security achieved by scaling encypted data causes an increase in data size. Increase in data size implies extra time in query processing. Another limitation is that it cannot provide security against prior knowledge of tag distribution, query workload distribution and correlation among data values. This approach is not very efficient in insertions and updates.

In the methodology proposed in [21], there are three phases for query processing. The first one is the query preparation phase which is offline. This phase contains encoding the structure and the instance of the XML document. The second phase is the actual query processing phase. This is the first online phase. Inappropriate XML document candidates are filtered out by examining query conditions in this phase. The selected candidate databases are returned to the client for further decrypting in the third phase.

2.5.2. Maintaining indexes at the client

In [5] an algorithm called XQEnc is used for encrypted XML query processing. This algorithm uses vectorization and skeleton compression together [22, 23]. Vectorization partitions an XML document into path vectors which are composed of nonempty leaf nodes. Skeleton compression removes the redundancy of XML documents by using common sub branch sharing. This approach shows that XML documents may become small enough to fit into the main memory.

This algorithm runs at the client side and it generates a selection query for the cryptoindexes and then sends it to the server. The server is treated only as an external storage. The server starts its job after the client sends the query. The task of the server is retrieving the encrypted results and sending them back to the client for further decrypting. The structural information always remains hidden from the server because the schema of the XML document is stored as a compressed skeleton at the client. One drawback of this approach is that the burden of the query processing is at the client side which decreases the performance. Every insertion into the XML database triggers the client side for an index update. Another drawback is associated with space management problems. Although skeleton compression makes the document

smaller, the client with its limited memory may still face problems.

2.6. HL7 clinical document architecture standard

We used clinical document architectures (CDA) in testing our system. CDAs are XML documents representing XML patient records. These documents contain detailed information about patients. One clinical document node contains information about one patient. Health Level Seven (HL7) Structured Documents Technical Committee has specified a CDA standard. A sample CDA schema and sample CDA instance can be examined in [24] and [25, 26], respectively.

3. Overall system architecture of the proposed system

The proposed model consists of two main phases: 1) offline phase in which the client splits, indexes and encrypts the database and 2) online phase in which query processing occurs. In this section the main phases and subphases of the system will be explained through examples from clinical documents.

3.1. First offline phase: splitting

The very first step of the proposed model is splitting the client's XML database without disrupting the hierarchical structure of the document. Due to the concerns of data confidentiality, this step takes place at the client side. The client makes use of an algorithm proposed in [27] to split an XML document into n partitions. According to this algorithm, client determines the number of computation nodes and the range factor and indicates the size of the original document. Next the resulting document sizes are computed. To give an example, if the original document size is 200 MB and the client aims 8 nodes to process concurrently with a range factor of 0.1, then the resultant documents have sizes between 22.5 and 27.5. (This is calculated by dividing the original document size, 200 MB, by the number of computation nodes, 8. The result of this division is 25 MB. When the range factor of 0.1 is multiplied by 25, we obtain 2.5. Thus we have the resulting document sizes varying in the interval 25 ± 2.5)

An important point to point out is that in our splitting algorithm, we split XML documents to subdocuments which contain different number of clinical document nodes. However, since it is not appropriate to disjoin one patient's information into several parts, we do not split clinical documents any further.

3.2. Second offline subphase: indexing

At the second step of our proposed model, split XML documents are indexed at the server side to expedite query processing. We make use of two different indexing techniques: 1) right and left bounds indexing [28] and 2) Dewey number indexing [29]. Tables 1 and 2 illustrate left bound and right bound indexing for a clinical XML document.

Table 1. Sample index table for nodes with Left and Right Bounds indexing

ID	Node Name	Left Bound	Right Bound	Value	Root to Path
1	Root	1	22	NULL	Root
2	CDA	2	11	NULL	Root/CDA
3	Patient	3	10	NULL	Root/CDA/Patient
4	Name	4	5	John	Root/CDA/Patient/Name
5	Surname	6	7	Parker	Root/CDA/Patient/Surname
6	Birthday	8	9	6/7/1963	Root/CDA/Patient/Birthday
7	CDA	12	21	NULL	Root/CDA
8	Patient	13	20	NULL	Root/CDA/Patient
9	Name	14	15	Caron	Root/CDA/Patient/Name
10	Surname	16	17	Brown	Root/CDA/Patient/Surname
11	Birthday	18	19	4/5/1980	Root/CDA/Patient/Birthday

Table 2. Sample index table for attributes with Left and Right Bounds indexing

ID	Attribute Name	Node ID	Value	Root to Path
1	ID	3	123	Root/CDA/PatientID
2	ID	13	124	Root/CDA/PatientID

In this indexing schema each node is given an incrementally increasing identification number. Then the following rule is applied: for a parent node p and a child node c, p.leftbound<c.leftbound and p.rightbound>c.rightbound. By looking at the left and right bound properties of the nodes, one can determine the ancestor – descendant relation between the nodes.

The absolute root to path index in Table 2 could be found by tracing the node ids but since it requires an extra join operation, we find it appropriate to explicitly store it in a table so that the queries can be executed efficiently.

Left and right bounds indexing expedite selection, deletion and update queries. The order of nodes is not affected when the target entry is deleted or updated. The important point in this schema is that, since left and right bound properties determine the ancestor-descendant relationship between the nodes, they have to be strictly preserved. In selection, deletion and update queries, left bound and right bound properties of the remaining nodes are preserved. In deletion, the left and right bounds of the deleted entry can later be used in an insertion.

However, this schema requires extra work in insertions because when an insertion occurs, left and right bounds of the other nodes become corrupted. One possible improvement to prevent this situation is creating empty space in the index boundaries. For instance, after indexing the XML document once, we can multiply every left and right bound property by a factor of 10. This will enable us to make a number of insertions without affecting the other nodes. Yet, it should be kept in mind that the XML document needs to be re-indexed after a while when we run out of empty space. There are two choices for re-indexing. In the first one, the server sends the sub document to the client and

gets it back from the client after re-indexing. This is not an efficient choice. The second choice is updating the index entries at the server side. In order to accomplish this, it is important to know where to insert the node. Hence the client first queries the server to find the related computation node. Then the client sends the insertion query together with an update operation in the index entries of the other fields. In this manner, the structure of the XML document is not destroyed and only one index table is rebuilt. It is obvious that an insertion operation is not as efficient as deletions or updates.

The second indexing schema used in the proposed model is Dewey numbering schema. Table 3 shows our sample clinical document with Dewey indexes of the nodes and Table 4 shows the Dewey indexes of the attributes.

In Dewey numbering a child's Dewey number starts with the parent's Dewey number (i.e. it has the parent's number as a prefix).

Table 3. Sample index table for nodes with Dewey Number indexing

ID	Node Name	Dewey Number	Value	Root to Path
1	Root	1	NULL	Root
2	CDA	1.1	NULL	Root/CDA
3	Patient	1.1.1	NULL	Root/CDA/Patient
4	Name	1.1.1.1	John	Root/CDA/Patient/Name
5	Surname	1.1.1.2	Parker	Root/CDA/Patient/Surname
6	Birthday	1.1.1.3	6/7/1963	Root/CDA/Patient/Birthday
7	CDA	1.2	NULL	Root/CDA
8	Patient	1.2.1	NULL	Root/CDA/Patient
9	Name	1.2.1.1	Caron	Root/CDA/Patient/Name
10	Surname	1.2.1.2	Brown	Root/CDA/Patient/Surname
11	Birthday	1.2.1.3	4/5/1980	Root/CDA/Patient/Birthday

Table 4. Sample index table for attribute Dewey Number indexing

ID	Attribute Name	Node ID	Value	Root to Path
1	ID	1.1.1	123	Root/CDA/Patient/ID
2	ID	1.2.1	124	Root/CDA/Patient/ID

When an update operation is executed on Dewey numbering, only the corresponding entry is modified and the other nodes do not need to be re-indexed. Thus update operation is as efficient as in left and right bounds indexing.

Insertions with Dewey number indexing are more efficient than those in left and right bounds indexing because the target node will just be the last node of the corresponding subtree. For example, assume that we also want to store the telephone number of John Parker. His telephone number will be inserted to the end of his subtree having the Dewey index 1.1.1.4.

However, deletions are not as efficient as those in left and right bounds indexing because of the fact that some number of nodes may need to be re-indexed after a deletion operation. For instance, if we delete the name entry with Dewey number 1.1.1.1 in Table 3, the surname's Dewey number has to change to 1.1.1.1 and that of birthday's to 1.1.1.2. Hence, in order to delete a node, first the client sends a selection query for all the nodes that are at the same level with the target node, then updates their Dewey numbers with an update query and finally deletes the target node with all of its contents and children.

3.2.1. Adding bogus data to indexes

To improve security against attacks, we propose to add bogus data to the original index tables. This requires an extra column that serves as a flag indicating whether the data are real or not.

The entries of this column are obtained by using a hash function which gives odd values for the real data and even values for bogus data. After adding bogus entries and encrypting the indexes, the data are sent to the server. While query processing, the server processes the real data as well as the bogus data. This may be considered as an extra overhead for the server, but it is necessary in order to improve the system's security. After the results of the queries are returned to the client, the client first decrypts the values in the flag column and then uses the hash function to obtain odd or even values. The client understands that the data have to be discarded, if the value in the flag column is even. A detail to consider at this point is that the client should also specify a seed (a prime number) to the hash function. While encrypting or decrypting the database, the client will give this seed as a multiplicative factor to the hash function and process accordingly. The seed is important because of the fact that an attacker may attain the implementation of the codes in the system where the hash function is written.

One way of adding bogus data is to make all the values to occur with the same frequency to improve security against frequency based attacks. However, this approach may cause the database to become so large that the gain from parallel processing may diminish. A more efficient approach, which we use in our proposed system, is to group the data first and then add bogus data in such a way that the data in each group occur with the same frequency. The group sizes may vary. Table 5 shows an example including a comparison between the former and latter approaches.

Table 5. Number of items after adding bogus data

A	B	C	D	E	F
IER	7	1560	1553	50	43
ASJDHG	8	1560	1552	50	42
USD	12	1560	1548	50	38
SJKDF	15	1560	1545	50	35
UER	22	1560	1538	50	28
ISDUYF	23	1560	1537	50	27
SUFDT	45	1560	1515	50	5
SSKLDF	123	1560	1437	165	42
WOERI	130	1560	1430	165	35
PSODF	133	1560	1426	165	31
WTER	134	1560	1427	165	32
QYTWE	145	1560	1415	165	20
PDFOG	150	1560	1410	165	15
AYS	1500	1560	60	1716	216
ADRS	1560	1560	0	1716	156

Column A in Table 5 represents the encrypted node names; column B represents the original number of items; column C represents the number of items after making each data item occur with the same frequency; column D represents the number of bogus items needed to be added to make each data item occur with the same frequency; column E represents the number of items after applying the proposed algorithm and finally column F represents the number of bogus items needed to be added in the proposed algorithm. The total number of original items in this dataset is 4007 whereas the total number of bogus items needed to be added to the dataset to make the data occur with the same frequency is 19393. The total items become nearly 5 times the original data causing a big burden on the system. On the other hand, when the proposed algorithm is used, the total number of items needed to be added is 765. Hence the efficiency of the system does not decrease too much in this case.

In order to group the data, we make use of a statistical outlier detection method well known in the literature whose principle idea is explained in the following. We begin with analyzing the input domain. We make a histogram of the number of distinct input values and sort the number of values in the histogram. Hereafter, starting with the first element, we apply outlier detection algorithm. If the n^{th} element is an outlier when started from the first element, then group first n-1 elements and start from the beginning again. As a result k distinct groups are formed. In each group, get the maximum number of occurrences for each element, multiply it with a coefficient slightly bigger than 1 and then start adding bogus data until the number of each element reaches the maximum number of its group times the specified coefficient. At the end, the frequency of occurrence of data items is uniformly distributed in each group.

In order to detect the outliers, the mean and the standard deviation of the frequency of occurrence of the data items are calculated. Afterwards the data are standardized by subtracting the mean from each data item's frequency of occurrence and dividing the result to the standard deviation. If the calculated value does not fall in the range of -1 to 1, then the value is considered to be an outlier. The reason we choose -1 and 1 as the boundaries of the interval is because they are nearly the optimum values in determining the outliers. If we choose a number greater than 1, the frequency values which are grouped together usually fall in the same range, the group sizes expand and the distribution nearly becomes uniform. If we choose a number smaller than -1, then there occurs a vast amount of groups with small sizes which is not good for security.

Once the bogus data are inserted, an attacker cannot infer the exact relationship between ciphertext and plaintext values. Even if he knows the exact frequency distribution of the data items, since we grouped the values, he again cannot infer that a certain group of plaintexts corresponds to a group of ciphertexts. Alternatively, if he knows the exact number of occurrences of a word in the input domain, since we added bogus data, he cannot find out the ciphertext value that the word corresponds to.

The third case is that the attacker may know k distinct values occurring in the domain but since we grouped the values, the number of candidate databases is too large for the attacker to guess. One more case to consider is that the attacker may know the maximum and minimum number of occurrences in the input domain. After multiplying with the coefficient and adding bogus data, the extrema are indistinguishable.

3.3. Third offline subphase: encrypting

Since AES is more secure and efficient than DES as mentioned in Section 2.3, we chose to use AES in our proposed architecture. After indexing the XML document with one of the indexing schemas and adding bogus data, the client encrypts the XML indexes with AES. In the node index table, only the node names and node values are encrypted. In attribute index table, attribute names and attribute values are encrypted. Left bounds, right bounds and Dewey numbers are not encrypted. One can think that this approach may reveal the structure of the XML document but it should not be forgotten that we add bogus data into the input set. By this means, the number of candidate databases becomes too high for the service provider or an attacker to determine the true database structure easily.

3.4. Online phase: query processing

After splitting, indexing and encrypting the data query processing takes place according to the W3C standards. The user creates a query which is translated into its encrypted form by the query translator and the service provider processes the query and sends the results back. The client decrypts the results and eliminates the bogus data to get the actual results.

4. Performance study and evaluation

We have implemented the proposed system in .Net 2.0. We executed sample queries for different size of datasets and different number of computation nodes. The results of these experimental executions will be given in this section.

All the experiments are done on an Intel ®Pentium® M processor 2.13GHz PC with 2GB RAM running under Windows XP. The relational indexes are implemented in MS SQL Server 2005. We used 20MB, 50MB and 100MB synthetic datasets and ran the queries on 5 and 10 computation nodes simultaneously for both indexing methods. We repeated each experiment 10 times and took the average response times. The measurements are done in terms of query processing times. We will list the graphical results of 5 processor systems as well as a summary of performance gain percentage for both 5 and 10 processor

systems. In the graphs, processor number 0 represents the case where there is only one processor in the system. Other processor numbers represent the number of processors in systems. A performance gain percentage given in this section specifies the average of the performance gain percentages of the 10 executions for each query. A performance gain percentage for one execution of a query q is computed as follows. Let p1 be the time it takes to execute query q using 1 processor and without using our methodology. Let pm be the time it takes to process query q using 5 or 10 processors using our methodology. The performance gain percentage is calculated by (p1-pm) * 100 / p1.

Table 6. Query1

List the attribute IDs of provider organizations whose name is 'HOLD'
//CDA/recordTarget/patientRole/providerOrganization/[class code='HOLD']

Query 1 is specified in Table 6. This query is a selection query from the attributes table. Translated version of the query is the same for both indexing methods so the execution times are identical. Total processing times of Query1 is small because there is only selection from a relational table which has its own indexes in it.

Figure 1 shows the total query processing time of Query1 on 20-50-100MB documents with 1 to 5 processors.

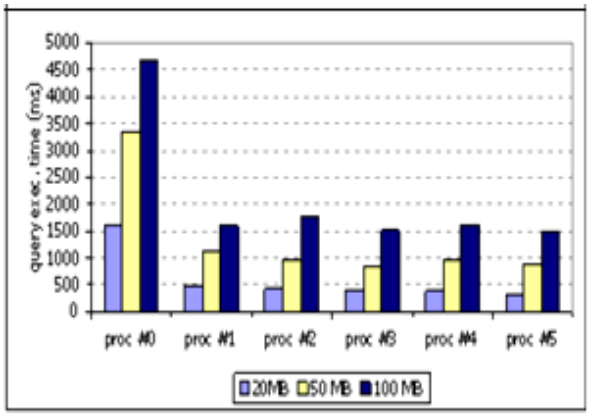

Figure 1. Query1 execution time using 1-5 processors

Summary of the performance gain percentage for Query1 is given in Table 7.

Table 7. Summary of performance gain percentage for Query1

Document Size	5 Processors	10 Processors
20 MB	70	80
50 MB	66	79
100 MB	62	79

Table 8. Query2

List the location of the patients whose disposition code is 'Token121'
//currentEncounter[//DispositionCode[code='Token132']][//location]]

Query2 is specified in Table 8. This query is join type. Therefore its total processing time is longer when compared to that of Query 1. Figure 2 shows the total processing time of Query2 with Left and Right Bounds indexing using 1 to 5 processors for 20-50-100MB documents.

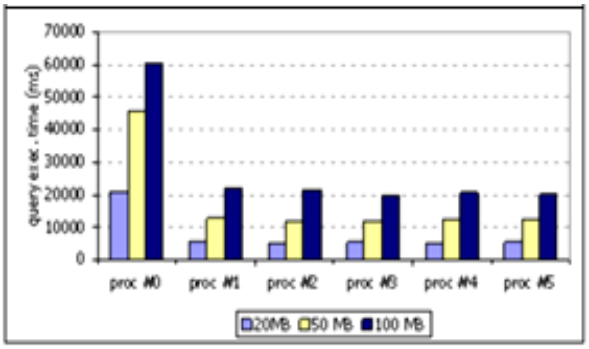

Figure 2. Query2 execution time with 1-5 processors using Left and Right Bounds Indexing

Summary of the performance gain percentage for Query2 with left and right bounds indexing is given in Table 9.

Table 9 Summary of performance gain percentage for Query2 with Left and Right Bounds Indexing

Document Size	5 Processors	10 Processors
20 MB	72	85
50 MB	72	82
100 MB	64	68

Figure 3 shows comparison of the total query processing time of Query2 using Dewey numbering with 1 to 5 processors for 20-50-100MB documents.

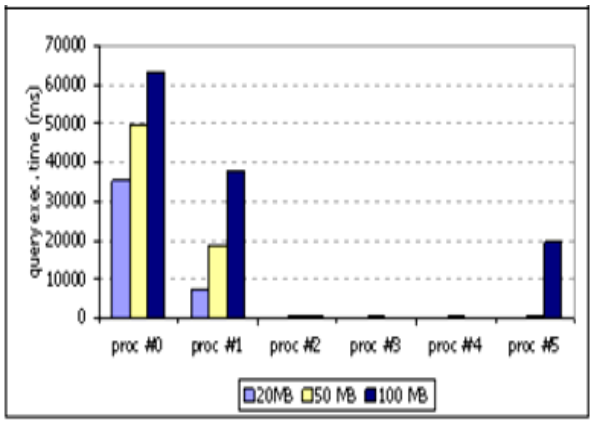

Figure 3. Query2 execution time with 1-5 processors using Dewey Numbers

In parallel processing, we executed this query in a manner that most of the results are given by processor 1 so the execution time in processor 1 is much greater than those of the other processors. However the performance gain is still considerable. Summary of the performance gain percentage for Query2 using Dewey numbers is given in Table 10.

Table 10. Summary of performance gain percentage for Query2 using Dewey Numbers

Document Size	5 Processors	10 Processors
20 MB	93	80
50 MB	81	62
100 MB	69	40

Query3 is specified in Table 11. This query contains both selection and projection operations.

Table 11. Query3

List the telecom of the customers whose organization is 'CUSTNAME'?
//CDA/custodian/assignedCustodian/[/name='CUSTNAME']/telecom

Figure 4 shows the comparison of the total query processing time of Query3 using Left and Right Bounds indexing with 1 to 5 processors for 20-50-100MB documents.

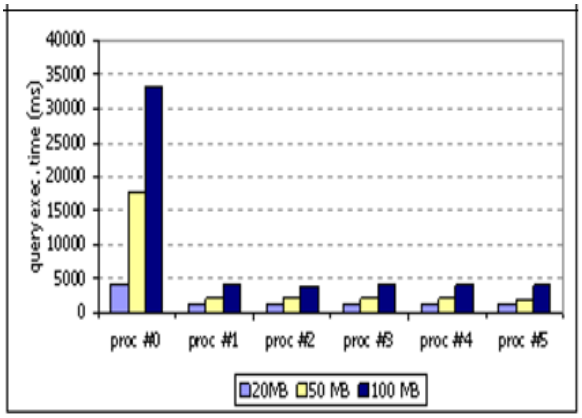

Figure 4. Execution time of Query3 with 1-5 processors using Left and Right Bounds Indexing

Summary of the performance gain percentage for Query3 using Left and Right bounds indexing is given in Table 12.

Table 12. Summary of the performance gain for Query3 using Left and Right Bounds Indexing

Document Size	5 Processors	10 Processors
20 MB	90	77
50 MB	75	61
100 MB	68	40

Figure 5 shows the comparison of the total query processing time of Query3 using Dewey numbering with 1 to 5 processors for 20-50-100MB documents.

Summary of the performance gain percentage for Query3 with Dewey numbering is given in Table 13.

Table 13. Summary of the performance gain for Query3 using Dewey Numbers

Document Size	5 Processors	10 Processors
20 MB	67	72
50 MB	88	91
100 MB	88	93

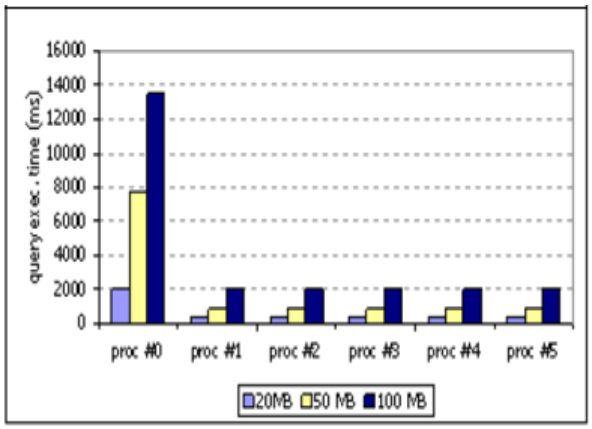

Figure 5. Query3 execution time with 1-5 processors using Dewey numbering

5. Conclusions

Encrypted query processing is a time consuming process. The methodology presented in this paper decreases the time spent for query processing. Query processing time decreases significantly and proportionally to the number of processors used. Moreover, we add extra security to the indexes stored at server side by adding bogus data and flagging it by an extra encrypted index so that it becomes difficult for an attacker to decrypt the content. The methodology presented is independent of the encryption algorithm used.

One of the important application areas of XML document systems is clinical documents [30] which may contain large amounts of data. They contain personal information and records that have to be kept private. The methodology proposed enables the clinic authorities to query the document without revealing the content to the service provider. We are not aware of any other methodology in the literature for parallel processing of encrypted XML documents in database as a service concept.

References

[1] **H. Hacıgümüş, S. Mehrotra, B. Iyer.** Providing Database as a Service. *Proceedings of the* 18*th International Conference on Data Engineering*, 2002, 29-40.

[2] **E. Mykletun, G. Tsudik**. Incorporating a Secure Coprocessor in the Database-as-a-Service Model. *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, 2005, 38-44.

[3] **H. Wang, L. Lakshmanan.** An Efficient Secure Query Evaluation over Encrypted XML Databases. *Proceedings of the* 32*nd International Conference on Very Large Databases*, 2006, 127-138.

[4] **O. Ünay, T.I. Gündem**. A Survey on Querying Encrypted XML Documents for Databases as a Service. *ACM SIGMOD Record, Vol*. 37, 2008, 12-20.

[5] **Y. Yang, W. Nig, H. Lau , J. Cheng.** An Efficient Approach to Support Querying Secure Outsourced XML Information. *Proceedings of the* 19*th International Conference on Advanced Information Systems Engineering,* LNCS 4001, 2006, 157-171.

[6] **B.Carminati, E. Ferrai.** Confidentiality Enforcement for XML Outsourced Data. *Proc. of the Second International EDBT Workshop on Database Technologies for Handling XML Information on the Web,* LNCS 4254, 2006, 234-249.

[7] **E. Damiani, S. Wimercati, S. Paraboschi**. A Fine-Grained Access Control System for XML Documents. *ACM Transactions on Information and System Security*, 2002, *Vol*.5*, Issue* 2, 169-202.

[8] **S. Hada, M. Kudo.** Provisional Authorization for XML Documents. *http://www.trl.ibm.com/projects /xml/xss4j/docs/xacl-spec.html.*

[9] **E. Bertino, E. Ferrari.** Secure and selective dissemination of XML documents. *ACM Transactions on Information and System Security*, 2002, *Vol*. 5, *Issue* 3, 290-331.

[10] **M.M. Kocatürk, T.I. Gündem**. A Fine-grained Access Control System Combining MAC and RBACK Models for XML. *Informatica,* 2008, *Vol*.19, *Issue* 4, 517-534.

[11] **G.I. Davida, D.L. Wells, J.B. Kam.** A Database Encryption System with Subkeys. *ACM Transactions on Database Systems,* 1981*, Vol*.6, *Issue* 2, 312-328.

[12] **D.X. Song, D. Wagner, A. Perrig.** Practical Techniques for Searches on Encrypted Data. *Proceedings of the IEEE Symposium on Security and Privacy*, 2000, 44–55.

[13] **H. Hacıgümüş, B. Lyer, C. Li, S. Mehrotra.** Executing SQL over Encrypted Data in Database-Service-Provider Model. *Proceedings of the* 2002 *ACM SIGMOD International Conference on Management of Data,* 2002, 216-227.

[14] **R. Agrawall, J. Kiernan, R. Srikant, Y. Xu.** Order Preserving Encryption for Numeric Data, *Proceedings of the* 2004 *ACM SIGMOD International Conference on Management of Data*, 2004, 563-574.

[15] **G. Ozsoyoglu, D.A. Singer, S.S. Chung.** Antitamper databases: Querying Encrypted Databases. *Proc. of the* 17*th Annual IFIP WG* 11.3 *Working Conference on Database Applications and Security,* 2003, 133-146.

[16] **S.S. Chung, G. Ozsoygolu.** Anti-tamper databases: Processing Aggregate Queries over Encrypted Databases. *Proc. of the* 22*nd International Conference on Data Engineering Workshops*, 2006, 98.

[17] **E. Damiani, S.D.C. Vimerati, S. Jajodia, S. Paraboschi, P. Samarati.** Balancing confidentiality and efficiency in untrusted relational DBMSs. *Proceedings of the* 10*th ACM Conference on Computer and Communications Security*, 2003, 93-102.

[18] **B. Hore, S. Mehrotra, G. Tsudik.** A Privacy Preserving Index for Range-Queries. *Proceedings of the* 30*th International Conference on Very Large Data Bases,* 2004, 720-731.

[19] **T. Immamura, B. Dillaway, E. Simon.** XML Encryption Syntax and Processing. *http://www.w3.org/TR/xmlenc-core.*

[20]. **J. Reagle.** XML Encryption Requirements. *http://www.w3.org/TR/xml-encryption-req.*

[21] **L.Feng, W. Jonker.** Efficient Processing of Secured XML Metadata. *On the Move to meaningful Internet Systems: OTM Workshops,* LNCS 2889, 2003, 704-717.

[22] **P. Buneman, B. Choi, W. Fan, R. Hutchison, R. Mann, S. Viglas.** Vectorizing and querying large XML repositories. *Proceedings of the* 21*st International Conference on Data Engineering,* 2005*,* 261–272.

[23] **J. Cheng, W. Ng.** XQzip: Querying compressed XML using structural indexing. 9*th International Conference on Extending Database Technology,* 2004, 219–236.

[24] CDA Schema. *http://xml.coverpages.org/CDA-ReleaseTwoSchema-200408.xsd.*

[25] CDA Instance. *http://xml.coverpages.org/CDA-ReleaseTwoSample200403.xml.*

[26] Encryption. *http://en.wikipedia.org/wiki/XML_Encryption.*

[27] **H. Kurita, K. Hatano, J. Miyazaki, S. Uemura.** Efficient Query Processing for Large XML Data in Distributed Environments. 21*st International Conference on Advanced Information Networking and Applications,* 2007, 317-322.

[28] **O. Li, B. Moon.** Indexing and Querying XML Data for Regular Path Expressions. *Proceedings of the* 27*th International Conference on Very Large Databases*, 2001, 361-370.

[29] **I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, C. Zhang.** Storing and Querying Ordered XML Using a Relational Database System. *Proceedings of the* 2002 *ACM SIGMOD international conference on Management of data*, 2002, 204-215.

[30] **N. Aksu, T.I.Gündem.** Indexing of Medical XML Documents Stored in WORM Storage. *Information Technology and Control,* 2009, *Vol*.38, *No*.1, 72 - 80

Received November 2009.

DOI: 10.5755/j01.itc.39.4.12381