Query Assurance Verification for Dynamic Outsourced XML Databases

Viet Hung Nguyen, Tran Khanh Dang, Nguyen Thanh Son                      Josef Küng
Faculty of Computer Science and Engineering,                                 FAW Institute
University of Technology,                                                     Johannes Kepler University of Linz,
Ho Chi Minh City, Vietnam                                                      Austria
{hungnv, dtkhanh, sonsys}@cse.hcmut.edu.vn                                    jkueng@faw.uni-linz.ac.at

Abstract

With rapid developments of network technologies, database outsourcing is emerging as an important new trend beside the “application-as-a-service”. In this model, data owners ship their data to external service providers. Service providers do data management tasks and offer their clients a mechanism to manipulate outsourced databases. Since a service provider is not always fully trusted, security and privacy of outsourced data are significant issues. These problems are referred to as data confidentiality, user privacy, data privacy and query assurance. Among them, query assurance takes a crucial role to the success of the database outsourcing model. To the best of our knowledge, however, query assurance, especially for outsourced XML databases, has not been concerned reasonably in any previous work.

In this paper, we propose a novel index structure, Nested Merkle B*-Tree, combining the advantages of B*-tree and Merkle Hash Tree to completely deal with three issues of query assurance known as correctness, completeness and freshness in dynamic outsourced XML databases. Experimental results with real datasets prove the efficiency of our proposed solution.

1. Introduction

With rapid improvement of network technologies, database outsourcing is emerging as an important trend beside the “application-as-a-service”. In the outsourced database service model (ODBS), data owners ship their data to external service providers. Service providers do data management tasks and offer their clients a mechanism to manipulate outsourced database. This helps organizations cut down the cost of hardware, software investment and maintainance. Specially, this model gives organizations a cost-effective means of employing outside professional staff that is increasingly expensive. Thank to the ongoing development of the chip industry, computer hardware is getting cheaper and more powerful than before. Nevertheless, we easily realize that that cost of software and experienced staff are increasing rapidly. By outsourcing, organizations could take the benefits of new software and have their system maintained, upgraded professionally with a feasible price.

Information is a valuable asset. Since a service provider is not always fully trusted, organizations need to be guaranteed that their sensitive data is protected. This kind of security requirement is different from that of traditional in-house databases. The question is “how is the client’s data protected against sophisticated attackers?” [1]. Attackers here mean both intruders and server operators. Because a server operator has rights to execute all database operations, traditional security barriers are useless with these malicious insiders. This big challenge of the model is the source of numerous researches in the community. These problems are stated as data confidentiality, user privacy, data privacy and query assurance [1, 14].

- **Data confidentiality**: data owners do not want anyone, except valid clients, to be able to see their outsourced data, even server operators.

- **User privacy**: server and even data owners should not know about the clients’ queries and returned results.

- **Data privacy**: clients are not allowed to get more information than what they are querying from the server.

- **Query assurance**: server has to prove that the returned results are original, complete and up-to-date.

In ODBS model (see Figure 1 [1]), data owners store their data at a server managed by an external company called service provider. Data owners manipulate their data through a secure communication. Clients, who pay for data access permission, if needed, could request data directly from service providers. Naturally, data owners themselves are clients.
To ensure data confidentiality, data is encrypted before outsourcing using symmetric algorithms (e.g., DES, Rijndael) or asymmetric algorithms (e.g., RSA). Commonly, symmetrical algorithms are preferred due to their tradeoff between security and performance.

Figure 1. ODBS model.

The next two requirements, user privacy and data privacy, are well done in many researches for both relational and tree-structured data. A detailed discussion of these two issues is out of the scope of this paper. More information can be found in [1,2,3,4,5,19,20]. In this paper, we focus on the last one, query assurance, for outsourcing XML data.

In outsourcing scenario, clients do not fully trust servers. Data owners have to give clients the ability to authenticate the results they received from servers. In that respect, query authentication has three important issues: correctness, completeness and freshness. Correctness means clients must be able to validate returned records to ensure that they have not been tampered with. Completeness guarantees that there is no matched record excluded from the answers. Finally, freshness shows that the answers are based on the latest version of the dynamic outsourced database.

Previous work often assumes that the database is read-only and the results are always fresh. Hence, only correctness and completeness are considered. In addition, most of these research activities have been done theoretically [6,12]. Recently, in [13,14], the authors mentioned about freshness, and carried out the evaluations with real datasets. However, none of them says about query assurance for outsourced XML database.

Our contributions. In this work, we propose a novel authenticated structure indexing all elements of a XML document, which includes both xml elements and xml attributes. Basing on this structure, we introduce a solution to the three mentioned issues for the outsourced XML database.

The rest of the paper is organized as follows. Section 2 gives some background information. Section 3 briefly reviews related work. Section 4 discusses XML data storage format at server. Section 5 is about our novel structure and how to use this structure to support query assurance. Section 6 presents analysis and evaluation of the proposed solution. Finally, section 7 concludes the paper.

2. Preliminaries

Basically, the idea of existing solutions mainly falls into two approaches:

- The first approach relies totally upon digital signature scheme. By signing on each record of each relation, clients can verify the correctness of the data. Signature is embedded with other information to give a proof of completeness, this kind of information will be discussed later. However, no research in this approach has mentioned about freshness.

- The second approach depends on a complex structure, also known as Authenticated Data Structure (AuthDS). Servers use this structure to build a verification object \( \mathcal{VO} \), then pass it along with the answers. After that, clients use the \( \mathcal{VO} \) to verify the returned results.

Besides, there are some researches in this area not belong to any approaches above. A brief discussion will be presented later.

2.1. Concept

Collision-free hash function \[13\]. For our purpose, a hash function \( \mathcal{H} \) is an efficiently computable function that takes a variable-length input \( x \) and returns a fixed-length output \( y = \mathcal{H}(x) \). Collision-free states that it is unable to find two inputs \( x_1 \neq x_2 \), such that \( \mathcal{H}(x_1) = \mathcal{H}(x_2) \), in a computational time.

Public-key digital signature schemes \[13\]. A public-key signature is a tool for authenticating the integrity and ownership of signed messages. Sender hash the message for digest, and then apply an asymmetric encryption on that digest to produce the signature. Signature is passed along with the original message. When receivers receive the message, they hash the message for the digest, and then verify it with the accompanied signature. If the verification is successful, receivers could be assured that what they received is originated from the sender.

Aggregating digital signature schemes \[13\]. The cost of digital signature computation and verification is quite expensive, especially, when receivers have to deal with thousands of signatures. A new technique has been developed to combine these signatures into a single aggregated one. Thereby, instead of sending and verifying thousands of signatures, a condensed signature is employed to ensure correctness of all messages \[9,18\].

The Merkle Hash Tree \[13\]. The Merkle hash tree (MHT), first proposed by R.C. Merkle, is one of the most popular authentic data structure. It is a binary tree, where each leaf contains the hash of the data val-
ue, and each internal node contains the hash of its two children. The hash function is usually collision-free. The verification is done based on the fact that the hash value of the root is authentically published (authenticity can be established by a digital signature). Beside the returned value, the sender also attaches several additional hash values of some nodes so that receiver could reconstruct hash value of the root. If this value equals the published value, the data is authenticated. By this mechanism, the Merkle hash tree provides an authenticity for simple point queries.

2.1. Methodology

Correctness. Correctness means data has not been tampered with as compared to the origin. Digital signature is the most favourite solution. If the data is signed by owners’ private keys and their public keys are well known, clients could easily verify that returned results are truly from the owner and have not been altered.

Completeness. There are a lot of methods to give proof of completeness. Most of the researches concentrate on range query and point query. In general, point query is the query that returns records whose attributes equal the conditional value and range query returns records whose attributes are within two given boundary values. Because point query is a specific case of range query, if we achieve completeness for range query, then we have it for point query in the same way.

Now, suppose that a range query \( Q \) on relation \( R \) returns a set of records \( S \), we have:

\[
S = \{ R | R \in R_l, R.x \geq LB, R.x \leq UB \} \tag{1}
\]

In which, \( LB \) and \( UB \) are two given boundary values and \( R \) is a tuple in relation \( R \).

To give a proof, the server includes two additional tuples of relation \( R \) in result set as follows.

\[
S_l = \{ R_l | R_l \in R_l, R.l.x = \max(R_x), R.l.x < LB, \forall \}
\]

\[
\cup \{ R_u | R_u \in R_u, R.u.x = \min(R_x), R.u.x > UB, \forall \} \tag{2}
\]

If this attribute orders the relation \( R_l \) and \( R_u \) are proven to be satisfied formula (2), there is no record falling into \( R_l \) and \( \min(S), R_u \) and \( \max(S) \); and so the completeness proof is achieved.

Freshness. Outsourced data is embedded with unique time information called timestamp [13,14]. Timestamp is widely published to all clients. When data owners change the data, they update the timestamp and announce it again. Freshness is achieved if clients could extract timestamp from the answers and this value equals the published value.

3. Related Work

There are several researches in this area, and we concisely summarize major work here. As presented in section 2.2, to achieve correctness, each tuple of a relation is signed by a private-key with a digital signature algorithm [6,9,14,18]. The signature is stored along with the tuple. Returned tuple itself contains a signature; clients recompute the hash and then verify the signature to ensure correctness. Another interest is the granularity that relates to which level of relation should be signed. There are three levels: whole relation, tuple or attribute [6]. Signing the whole table requires the server to return all tuples in the answers, which is unfeasible. If signature is at row-level, the answers will include all attributes of the rows so the data privacy is violated. In case of column-level, the computation and storage cost at both server and clients are overhead due to expensive large-integer calculations for a signature. However, by employing aggregated signature technique, several signatures are combined into the only one passed to clients. Clients do a single verification for all returned data, column-level granularity could be taken into account.

A different approach uses an extra data structure called authenticated data structure (AuthDS), proposed in [12,13], by extending the idea of MHT. This approach provides both correctness and completeness [12,13] as well as freshness [13]. In this scheme, data is sorted at leaves of a tree. Leaves also contain hash of data; internal nodes contain hash of their children that calculated by hashing the combination of children’s hashes. These calculations ensure the order of data in the sorted list. Therefore, as mentioned in section 2.2, they could prove both correctness and completeness. Additionally, the root contains a well-known timestamp value [13]. Clients will extract the timestamp from the answers; compare it with this value to know how much up-to-date the answers are.

AuthDS is usually large and complex. It takes an extra cost for storing and maintaining these structures. In [9,18], the authors introduced a new signature-based method called Digital signature aggregation and chaining (DSAC), which gives a concise proof of correctness and completeness with a smaller-impossible storage cost. The main idea of DSAC is also the same as that of AuthDS, which was detailed in section 2.2.

![Figure 2. Signature Chain](image)

The relation is sorted by all searchable attributes to have ordered lists. Each tuple has some neighbors (left and right) in lists. For example, in Figure 2 [9], left neighbors of R5 are R6, R2, R7 in three searchable dimensions based on attribute A1, A2 and A3 respectively. We call left neighbors immediate predecessor.
The signature of each tuple contains hashes of its entire immediate predecessor (IPR).

\[ \text{Sign}(r) = h(h(r))\| h(IPR_1(r))\| \ldots h(IPR_l(r))\| h(R) \]

(3)

Return to our example, we have signature of R5 computed as:

\[ \text{Sign}(R5) = h(h(R5))\| h(R6)\| h(R2)\| h(R7) \]

With signatures chained in the above fashion, the server answers a range query by releasing all matching tuples; the boundary tuples which are just beyond the query range (to provide a proof of completeness) as well as the aggregated signature corresponding to the result set. The signature chain proves to the querier that the server has indeed returned all tuples in the query range. Specifically, the querier verifies that the values in the boundary tuples are just beyond the range posed in the query. At the same time, the querier verifies that there are no other tuple between the boundary tuples and the satisfied ones. This is because boundary values are linked to the first and the last tuple. Therefore, the querier obtains a concise proof of completeness [9,18].

The above paragraphes discuss two main approaches in query authenticating: the signature based approach and the AuthDS approach. However, these two approaches concentrate only on range queries and do not address aggregated ones. Moreover both of them are query-understanding that means these solutions have to analyze the query syntax, requiring some extra information (signature or AuthDS) for the proof, and do some specific tasks in query processing process. This causes a high cost with respect to complex query processing.

To overcome these problems, a new approach extending the ringer protocol of distributed computing has been developed [7]. In this approach, query authenticating are proven for all types of query, regardless their forms. However, the probability of successfully cheating at servers is too high, approximate 33% [7].

4. Xml Databases

XML is now a de factor standard and widely used over the internet. Therefore, the need of outsourcing XML data is quite natural. Some research work done recently could be applied to outsourced XML databases in order to obtain privacy and confidentiality [2,4,8]. None of them, however, is about query authenticating for XML database although there are a lot of efforts for the relational database.

To cope with security problems of outsourced XML databases, we could create an algorithm to transform an XML database to a traditional relational database in which security problems are quite well-solved. Alternatively, we need a special method dealing with security issues in outsourced XML databases. Hence we should first answer the question “how do we store XML data?” The answer strongly influences to the way we deal with this problem.

**Table based.** Each XML document has a schema, we call it *schema tree* that defines the relation between nodes and their attributes (parent-child relationship). A schema tree consists of two node types: *element node* and *attribute node*. Hereafter, we name *element node* as *t-node* and *attribute node* as *a-node*. With the schema tree, we could easily change an XML document into relational tables as follows. First, label all t-nodes with integers incrementally so that their labels are unique in the whole XML document. For each *t-node*, create a table named by suffixing t-node’s name with its label. The table should have one more column, called *ID*, that is its primary key. If a *t-node* has a parent, we append a column named *PNodeID* to point to its parent’s corresponding table. Finally, for each child node *a-node*, append to the table a column named as the *a-node*’s name.

![Figure 3. Transform an XML document into tables.](image)

Figure 3 illustrates a sample. From a given XML document (A), we extract its schema tree (B). After labeling, we transform it into relational tables (C).

Once an XML document is transformed into traditional tables, we could apply prior work to achieve query authenticating. However, an XML database schema is changed easily. That causes the table schema to change also, and sometime forces the data to be re-outsourced.

**Node based.** XML document is structured as a tree with *t-nodes* and *a-nodes*. We could save *t-nodes* and *a-nodes* as tuples in a single table. Each node has enough information to reconstruct the XML text. We propose the following schemas for *t-node and a-node*.

| t-node(nodeid, xtype, datatype, nameid, pnodeid, lmaid, value) |
| a-node(nodeid, xtype, datatype, nameid, pnodeid, sibid, value) |

In which, *Xtype* is used to distinguish *t-node* and *a-node* among the records. *Datatype* determines type of the value (text, numeric, etc.). *NameID* is the node’s identity (using the labeling in the same manner of table based). *PNodeID* refers to parent node’s tuple. *ImaID* refers to the left-most attribute of the node. *SibID* refers to the right sibling of the attribute. *Value* is the value of the node/attribute.

For security reasons, this information is serialized into an encrypted binary string before outsourcing. Storing XML documents under such a node-based format conforms to tree-structured data. Changing an XML document schema does not dramatically affect
the outsourced data because it just appends the modified node to the database. However, each element is stored by a lot of records for the tag name and attributes. Thus outsourced database storage cost may become overhead. Furthermore, it is difficult to utilize existing indices. This problem extremely affects the overall performance.

5. Query Assurance

An important factor for a feasible solution to query assurance of outsourced XML databases depends on index structures, which are employed to manage the storage and retrieval of the data. By embedding extra information into this structure, we could achieve query assurance. Most related work about XML indexing could be found in [15,16]. These structures, however, are not suitable for query authenticity purpose because we expect an ordered list of all XML elements (both xml-tags and xml-attributes) for giving completeness proof (cf. section 2.2). Additionally, most of the queries require join-operations that take an expensive cost for proving completeness. A simple way of minimizing these costs is to sort XML elements by their parent. As mentioned in section 2.2, in order to obtain an effective completeness proof, XML elements are sorted by two criteria: (path, value) and (path, parent, value), where path is the path from the tree’s root to a given node. However, existing data structure could not help this. In the next sections, we will introduce a novel index structure, called Nested Merkle B’-Tree, to facilitate the storage and retrieval as well as to ensure the query correctness, completeness and freshness for outsourced XML databases.

5.1. Nested B’-Tree

As mentioned above, we desire an index structure that can sort all elements by their corresponding combinations (path, value) and (path, parent, value); where path is the path from the root to a given node. Here, we list out all possible paths in the XML document by a tree traverser algorithm, associating each path with an unique integer. We call this number nameid. To construct our desired structure, we first employ a B’-Tree structure, NameTree, with the search key is nameid (equivalent to path). At each leaves’ entry of NameTree, instead of record’s links, we store two references to two new B’-Trees having value and (parent, value) as their search keys, respectively. We call these trees ValueTree and ParentTree (see Figure 5). At all leaves of ValueTree and ParentTree, we store links to a-node or t-node records (hereafter, referred to as data records). Note, if many data records have the same key then they are in a same entry. This entry points to an additional structure called bucket which stores references to all same data records. Combining these trees, we have our desired structure named Nested B’-Tree (NBT).

5.2. Nested Merkle B’-Tree

Basing on the idea of the MHT, we attach some information to NBT for giving proof of query assurance. Each node of NBT has some extra hashes of its children. The hash values are calculated as follows.

\[ a\text{-node} : \text{H}_{a\text{-node}} = h(\text{nameid}||x\text{type}||...||\text{value}) \]

\[ t\text{-node} : \text{H}_{t\text{-node}} = h(\text{nameid}||...||\text{value})||\text{HL}_{\text{child-node}} \]

\[ \text{Leaf of ValueTree, ParentTree: } \text{H}_{v} = h(\text{attr} \text{H}_{\text{data-record}}) \]

\[ \text{Internal node: } \text{H}_{i} = h(\text{H}_{\text{child-node}}) \]

\[ \text{Leaf node of NameTree: } \text{H}_{\text{LN}} = h(\text{H}_{\text{tree}}||\text{H}_{\text{freed}}) \]

\[ \text{Root node of NameTree: } \text{H}_{\text{R}} = h(\epsilon \text{H}_{\text{child-node}}) \]

Where, \text{H}_{\text{attr}} is hash value of an a-node of a given t-node. \text{H}_{\text{data-record}} is either H_{a\text{-node}} or H_{t\text{-node}} that associated to the link. H_{\text{child-node}} is one of H_{1}, H_{2} or H_{\text{LN}}. H_{\text{tree}} and H_{\text{freed}} are hash values of the roots of corresponding ParentTree and ValueTree. \epsilon denotes a timestamp value and \text{h}() is a secure hash function (e.g. SHA). Additionally, we sign the root of NameTree by private-key of a digital signature scheme (such as RSA, DSA). The corresponding public key and timestamp \epsilon are broadcasted to all clients.

5.3. Providing query assurance

Correctness and Completeness. Assume that the result set consists of leaf entries in ValueTree that contain links to records fallen into a given range. As mentioned in section 2.2, two additional boundary records are also included in the result set. These entries occupy some adjacent leaves, say \{L_{i}, L_{i+1},..., L_{j}\}. The server returns data records and hash values of not-in-result entries in L_{i} and L_{j} so that clients could recompute hash values of these leaves. Similarly, the series \{L_{i}, L_{i+1},..., L_{j}\} occupies entries in some internal nodes, say \{I_{i}, I_{i+1},..., I_{j}\}. In addition, hash values of the remained entries in internal nodes are returned. Recursively, the server returns the root with its signature and timestamp. These not-in-result entries (in both leaves and internal nodes) are called co-path. Actual results and co-path are packaged in a structure called VO (verification object) and sent to clients. The clients will recalculate hash value of the root, then verify it with the signature to assure both correctness and completeness. More detail of co-path and VO are in [12].

Freshness. Along with the result set, the server returns timestamp value of the root. After verifying the root’s signature, clients compare returned timestamp to the well-known timestamp broadcasted by the data owner before. If two values are equal, clients are guaranteed about freshness of the result.
5.4. Select operation

The above shows the simplest situation that returns a single, contiguous record set. It is just a single step among many ones. To answer a query, the server has to perform some other tasks in a specific order called execution plan. Next, let’s see some examples of XPath to find out a general method for query processing.

As illustrated in Figure 4, the rectangular nodes stand for elements and the sharp corner ones are attributes. The numbers next to node are their nameid we mentioned above.

Example 1. The first question is “Find out all sold items named ‘TV’". The server scans NameTree with nameid = 13 to get the ValueTree. Then, the server scans on the found ValueTree with value = ‘TV’ to list out all satisfied attributes name_13 and builds a VO for authenticity. With the pnodeid field in each found name_13, the server reads and appends these Item_8 into the VO. Because there is only one Item_8 for each name_13, the server needs not to provide any information to prove query assurance. For each Item_8, the server returns two remain attributes.

Example 2. The second example deals with a question “List out all items bought by Marry?”. The server does some similar jobs to obtain satisfied Customer_1 records. For each Customer_1, the server scans on ParentTree, with pnodeID equals id of Customer_1. The ParentTree, is the ParentTree located in the leaf entry that nameid = 3 of the NameTree. A VO is built to give proof of query assurance for these records. In the same manner, the server returns expected Item_8 records and their VOs.

Unifying VOs. For each matched Customer_1 record, the server returns an Order_3 record set that have pnodeid point to the Customer_1 record and, similarly, an Item_8 record set for each found Order_3. Therefore, the server returns a lot of record sets and each of them requires a VO. Thus, the number of VOs is linear with the product of matched records in Customer_1 and Order_3. This will dominate communication and computation cost at clients. Hence, instead of building multiple VOs, the server only builds a single VO for all returned record sets by making a slight modification on the co-path generating algorithm. Thus, the clients only have to carry out one verification for all returned record sets.

The above deals with simple queries. However, by applying this idea, we could answer more complex questions, but we do not detail it here due to space limitation.

5.5. Update operations

Since the database could be changed even if it has been outsourced, feasible solutions should have an acceptable cost for update operations. Update operations refer to insertion, update, and deletion. Data owners could make changes to a local copy of the database then outsource it again. Here, we refer to another scheme that data owners do not have any local copy and/or re-outsourcing is impossible.

The most important issue of an update operation is to recalculate embedded security information. In this scheme, we should recalculate the data nodes relevant and their associated index nodes. In general, an update could be treated as a series of delete and insert operations. We summarize them as follows.

Insertion. To insert a new element into database, the owner first serializes it into t-nodes and a-nodes. These nodes are sent to server to insert into the database and update the index structure. This will change the hash value of a leaf, then the change is propagated to its ancestors (up to the root). The server recalculates these hash values and returns the new hash of the root to the owner. The owner generates a new timestamp ε, combines it with the hash value then signs on it with the private key. The signature and ε are sent to the server to update the root. Finally, the owner announces the new timestamp to all clients. If many nodes are inserted at the same time, batch operation could be considered to minimize the root resigning round.

Deletion. Performing a delete is quite similar to an insert operation. The only difference is the first round of the protocol. Instead of inserting, server has to locate deleted node in the index tree then removes it from the index and database. The remains are the same as that of the insertion process.

6. Analysis and Experiments

Until now, no research work about query assurance of outsourced XML databases has been carried out. Hence, we will theoretically analyze our proposed solution with respect to storage cost and VO size. The table 1 summarizes the notions used in this section.

The experiments are conducted on a P4-2.8GHz Windows XP SP-2 with 512MB of memory. Benchmarking program is written in VB.NET 2005. Data encryption and digital signature are Rijndael and RSA-1024, respectively. The DBMS is MSSQL 2005 Express for storing the encrypted database. Moreover, test
data is 69,846 items dataset [17], representing about world geographic database integrated from the CIA World Factbook, for all the tests. Finally, Window high-resolution timer is used to measure time cost.

Table 1. Formula notions.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>Total number of data item (element and attribute)</td>
</tr>
<tr>
<td>s</td>
<td>Number item of result set.</td>
</tr>
<tr>
<td>f</td>
<td>fanout parameter NMB+-Tree.</td>
</tr>
<tr>
<td>L_{h_{\min}}</td>
<td>Min/max height of a tree $T$.</td>
</tr>
<tr>
<td>L_{l_{\min}}</td>
<td>Min/max number of leaves in a tree $T$.</td>
</tr>
<tr>
<td>N_{\text{min}}</td>
<td>Min/max of total nodes in a tree $T$.</td>
</tr>
<tr>
<td>$</td>
<td>\text{sign}</td>
</tr>
</tbody>
</table>

Storage cost. Storage cost here is referred to as the cost for additional security information. In our scheme, this is the cost of storing the NMB’-Tree. The NMB’-Tree consists of one NameTree, several ValueTrees and ParentTrees. Because the number of nameid is small compared to that of elements, the storage cost for ValueTrees and ParentTrees dominates the overall storage cost. We, however, could not formulate the number of ValueTrees and ParentTrees in a database because it depends on the XML document structure. Therefore, we suppose that only one type of node in the XML document, hence there is only one nameid in the whole database. This means we have an element NameTree, one ValueTree and one ParentTree. Thus the storage cost is cost of storing n-elements ValueTree and ParentTree.

In addition, we assume that these nodes are distinguished by their values. We could easily find that the minimal and maximal number of leaves is calculated as formula (10). Here we can get the minimum when all leaves are full ($f$ entries) and the maximum when all leaves are half-full ($f/2$ entries).

$$h_{\text{Tree}}^{\text{min}} = \left\lfloor \frac{n}{f} \right\rfloor, \quad h_{\text{Tree}}^{\text{max}} = \left\lfloor \frac{2n}{f} \right\rfloor$$

Then we have:

$$h_{\text{Tree}}^{\text{min}} = \left\lfloor \frac{\log_f L_{\text{Tree}}^{\text{min}}} \right\rfloor + 1, \quad h_{\text{Tree}}^{\text{max}} = \left\lfloor \frac{\log_f L_{\text{Tree}}^{\text{max}}} \right\rfloor + 1$$

$$N_{\text{Tree}}^{\text{min}} = L_{\text{min}} \left( \frac{1 - \frac{1}{f^{h_{\text{Tree}}^{\text{min}}}}}{1 - \frac{1}{f}} \right) + 1, \quad N_{\text{Tree}}^{\text{max}} = L_{\text{max}} \left( \frac{1 - \frac{1}{f^{h_{\text{Tree}}^{\text{max}}}}}{1 - \frac{1}{f}} \right) + 1$$

Hence, the overall storage cost is calculated as (13).

$$C_{\text{storage}} = N_{\text{Tree}}^{\text{min}} + N_{\text{Tree}}^{\text{max}} + h_{\text{Tree}}^{\text{Tree}}, \quad C_{\text{storage}} = N_{\text{Tree}}^{\text{max}} + N_{\text{Tree}}^{\text{max}} + N_{\text{Tree}}^{\text{max}}$$

In fact, there are many elements that may have the same (nameid, value) or (nameid, parentID, value). These elements will occupy only one entry in the index tree. So, before applying these formulas, we should determine a value $n$ that is the number of distinct elements by mentioned conditions. Then, we replace any occurrence of $n$ in formula (10) by $n'$. The actual storage cost is achieved by counting number of index nodes in the database. Figure 6 shows detailed information about the storage cost for our sample dataset.

VO size. Similarly, we only concern about extra information returned to clients to give query authenticity. Formula (14) is used to calculate size of a single $\psi_0$.

$$C_{\psi_0} = \sum_{i=1}^{h} N_{\text{Tree}}^{\text{min}}, \quad i = \log_f L_{\text{Tree}} - s + 2$$

Overall cost. The overall cost includes I/O cost, communication cost, CPU cost at both server and client. Our experiments are performed in a local environment, therefore, communication cost is omitted. But this is directly proportional with the $\psi_0$ size. We simulate the overall cost by measuring query processing time since a query is submitted to server until a completed XML result text is reconstructed from the returned data. To know the I/O cost, for each query, we measure number of nodes that have been loaded from the database into memory during the query processing process. There are six phases of the query processing process: parse the query; plan the execution; fetch data from database; build the $\psi_0$; verify the $\psi_0$; and generate XML text. The first two phases, parse and plan, are ignored in our benchmarking program. To make a comparison, we import a same XML document into SQL Server and execute the same query on it. Experimental results are shown in the next paragraphs.

Experimental results. Figure 6 presents the storage cost in tests. The MinNodes and MaxNodes are theoretic nodes calculated by formula (13),(14). Figure 7 shows the increase of the execution time, I/O cost over the result size and database size. In which, result size is number of items in the result set. Executing time is the overall time, including time for fetching data from the database into memory; time for building $\psi_0$; time for verifying the $\psi_0$ at client; and time for reconstructing XML text. SQL Time is the executing time SQL Server used to execute the query. I/O Cost is number of nodes being loaded into memory during the process.

7. Conclusions and Future Work

This work explored the problem of query assurance of query replies in outsourced database. In particular, we developed a novel index structure called Nested Merkle B’-Tree by which we could completely achieve query assurance for dynamic outsourced XML data,
ensuring the query correctness, completeness and freshness. Our proposed solution is among the first efforts in this area. We also implemented a benchmarking program for experiments and carried out evaluations with real datasets to show the efficiency of the proposed solution.

In the future, we will further investigate other complex forms of XPath/XQuery, especially aggregated function ones. Moreover, another approach could be taken in mind is to employ multi-dimensional access methods (MAMs) [21] for the storage and retrieval management of outsourced XML data. Although MAMs bring in many advantages for the indexed data, they introduce challenging issues related to security, especially for outsourced databases. Thus, further research in this direction will be interesting.

References


[7]. R.Sion, “Query Executing Assurance for Outsourced Da-tabases”, Proc. 31st VLDB Conf., Norway, 2005


[10]. R. Sion and B. Carbunar, “ Conjunctive Keyword Search on Encrypted Data with Completeness and Computational Privacy”, Cryptology ePrint Archive, 2005


