A Novel Solution to Query Assurance Verification for Dynamic Outsourced XML Databases

Viet Hung Nguyen, Tran Khanh Dang
Faculty of CSE, HCM University of Technology, Ho Chi Minh City, Vietnam
Email: {hungnv, dtkhanh}@cse.hcmut.edu.vn

***Abstract* – Database outsourcing model is emerging as an important new trend beside the "application-as-a-service" model. In this model, since a service provider is typically not 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, named 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-world datasets prove the efficiency of our proposed solution.

***Index Terms* – outsourced XML database, query assurance.**

I. INTRODUCTION

Database outsourcing is emerging as an important trend beside the "application-as-a-service". In the outsourced database service model (ODBS, see fig. 1), 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 maintenance. Specially, this model gives organizations a cost-effective means of employing outside professional staff that is increasingly expensive. Computer hardware is getting cheaper and more powerful than before. Nevertheless, 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 fully trusted, organizations need to be guaranteed that their 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?" [17]. 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* [2, 17].

- *Data confidentiality*: unauthorized people are not able to see outsourced data, even server operators.
- *User privacy*: server and data owners should not know about the clients' queries and returned results.
- *Data privacy*: clients could not 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.

To ensure *data confidentiality*, data is encrypted before outsourcing using symmetric algorithms (e.g., DES, Rijndael) or asymmetric ones (e.g., RSA). Commonly, symmetric algorithms are preferred due to their tradeoff between security and performance.

The 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 issues is out of the scope of this paper. More information can be found in [3, 7, 9, 15, 16, 17]. This paper focuses on the last one, *query assurance*, for outsourced XML data.

In outsourcing scenario, clients do not fully trust servers. Clients should be able 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. 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. Moreover, most of these research activities have been done theoretically [1, 8]. Recently, in [2, 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. This paper proposes a novel authenticated structure indexing all elements of a XML

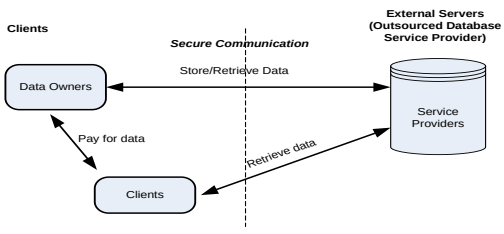

Figure 1. ODBS models

document. Then we introduce a solution to the three mentioned issues for the outsourced XML database.

The rest of the paper is organized as follows. Section II gives background information. Section III briefly reviews related work. Section IV discusses XML data storage format at server. Section V is about our novel structure. Section VI presents analysis and evaluation of the proposed solution. Section VII concludes the paper.

II. FUNDAMENTAL THEORY

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 data's *correctness*. Signature could be 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 one uses a complex structure, called Authenticated Data Structure (AuthDS). Servers use this structure to build a *verification object* $\mathcal{VO}$, and 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 in the next section.

A. Concept

Public-key digital signature schemes [2]**.** A public-key signature is a tool for authenticating the integrity and ownership of signed messages. Digital signature scheme is the combination of asymmetric encryption and collision-free hash function. Hashing the message, senders have its *digest*, and then apply an asymmetric encryption on that digest to produce the signature. Signature is sent along with the original message. When receivers receive the message, they hash the message for the digest, and then verify the accompanied signature. If the verification is successful, receivers could be assured that what they received is originated from the sender. The most popular digital signature scheme is RSA.

Aggregating digital signature schemes [2]. The cost of digital signature computation and verification is expensive, especially, when receivers deal with thousands of signatures. A new technique has been developed to combine these signatures. Thereby, instead of sending these signatures and making corresponding thousands of verifications, a condensed signature is employed and a single verification is done to ensure correctness. The *Condensed-RSA* and the *BGLS aggregated signature* are two popular samples of this scheme [5].

The Merkle Hash Tree [2]**.** The Merkle hash tree (MHT) (see fig. 2), 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 value, and each internal node contains the hash of its two children. The hash function is usually *collision-free*. The verification is done because the hash value of the root is authentically published (authenticity can be established by a digital signature). Beside the returned value, the sender 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.

Fig. 2 [2] shows an illustrative example. The answer here is {5}, and then the server returns additional hash values $\{h_1, h_{34}\}$ to clients. Clients computes $h_2' = h(5)$, $h_{12}' = h(h_1 \| h_2')$, and $root' = h(h_{12}' \| h_{34})$. Since the *root* value is published, if $root = root'$, correctness is achieved

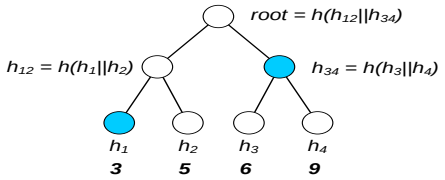

Figure 2. Example of the Merkle hash tree.

B. Methodology

Correctness. *Correctness* means data has not been tampered with as compared to the origin. Digital signature is the most favorite 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 $\mathcal{R}$ returns a set of records $\mathcal{S}$, we have:

$$\mathcal{S} = \{R \mid R \in \mathcal{R}, R.x \geq LB, R.x \leq UB\} \quad (1)$$

In which, *LB* and *UB* are two given boundary values and *R* is a tuple in relation $\mathcal{R}$.

To give a proof, the server includes two additional tuples of relation $\mathcal{R}$ in result set as follows.

$$S_b = \{R_L \mid R_i \in \mathcal{R}, R_L.x = max(R_i.x), R_i.x < LB, \forall i\} \cup \{R_U \mid R_j \in \mathcal{R}, R_U.x = min(R_j.x), R_j.x > UB, \forall j\} \quad (2)$$

If this attribute orders the relation $\mathcal{R}$ and R_L, 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 [2,14]. 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 extracted timestamp from the answers equals the published one.

III. RELATED WORK

There are several researches in this area, and this section will concisely summarize major work here. As presented in section II.B, to achieve *correctness*, each tuple

of a relation is signed by a private-key with a digital signature algorithm [1, 5, 14]. The signature is stored along with the tuple. Returned tuple itself contains a signature; clients re-compute 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 of granularity: whole relation, tuple or attribute [1]. 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 [2, 8], by extending the idea of MHT. This approach provides *correctness* and *completeness* [2, 8] as well as *freshness* [2]. 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 II.B, they could prove both *correctness* and *completeness*. Additionally, the root contains a well-known timestamp value [2]. 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 [5], 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 effective storage cost. The main idea of DSAC is also the same as that of AuthDS, which was detailed in section II.B.

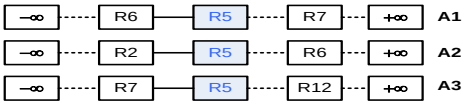

Figure 3. Signature Chain [5].

The relation is sorted by all searchable attributes to have ordered lists. Each tuple has some neighbors (left and right) in lists. In fig. 3, left neighbors of R5 are R6, R2, R7 in three searchable dimensions based on attribute A1, A2 and A3 respectively. Ref. [5] calls left neighbors *immediate predecessor* (IPR). The signature of each tuple contains hashes of its entire IPR (see formula 3).

$$Sign(r) = h(h(r)||h(IPR_1(r))|| \ldots h(IPR_l(r)))_{SK} \quad (3)$$

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. 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 [5].

The above paragraphs 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. It is means these solutions have to analyze the query syntax, and require some extra information (signature or AuthDS) for the proof, and do certain 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 [11]. In this approach, query authenticating is proven for all types of query, regardless their forms. However, the probability of successfully cheating at servers is too high, approximate 33% [11]. A recently research employs the idea of fake records, which is stated to be deal with join query more efficient [6].

IV. Storage of Outsourced XML Databases

XML is a de-factor standard for information interchange. An XML database consists of semi-structured tree data in text format. Therefore, the need of outsourcing XML data is quite natural. Many recent work could be applied to outsourced XML databases to obtain privacy and confidentiality [2, 4, 9]. None of them, however, is about query authenticating for XML database although there are many efforts for the relational one.

To cope with security problems of outsourced XML databases, we could create an algorithm to transform an XML database to a relational one in which security problems are solved well. Alternatively, we need to develop a particular method dealing with the problems. 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, called *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, referred to as *t-node* and *a-node*, respectively). With the schema tree, an XML document could be transformed to 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*, which 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 attribute *a-node*, append to the table a column named as the *a-node*'s name.

Fig. 4 illustrates a sample. From a given XML document (A), we extract its schema tree (B). After labeling,

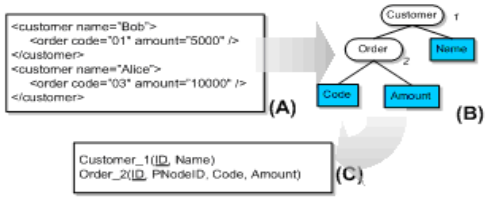

Figure 4. Transform an XML document into tables.

we transform it into relational tables (C).

Once an XML document is transformed, prior work is applied 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* that could be saved in a single table. Each node has enough data to reconstruct the XML text. Schema for *t-node* and *a-node* is as follows.

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 (use labeling like that of *table based*). *PnodeID* refers to parent node's tuple. *lmaID* refers to the left-most attribute of the node. *sibID* refers to the right sibling 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 many records for the tag name and attributes. Thus outsourced database storage cost may become overhead. Furthermore, it is too difficult to utilize the existing indices.

V. QUERY ASSURANCE FOR OUTSOURCED XML DATABASE

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 [4, 18]. However, these structures 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 II.B). Additionally, most of the queries require join-operations that take an expensive cost for proving the query completeness. In order to obtain an effective proof for the completeness (cf. section II.B), all elements should be 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. Therefore, we 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.

A. Nested B$^+$-Tree

As mentioned above, we desire an index structure that sorts all elements by their corresponding combinations (*path, value*) and (*path, parent, value*). Here, we list out all possible paths from root to leaves, and associate each path with a unique integer called *nameid*. To construct this structure, a B$^+$-Tree, named *NameTree*, with the search key is *nameid* (equivalent to *path*) is employed. At each leaves' entry of *NameTree*, instead of record's links, there are two links to two new B$^+$-Trees having *value* and (*parent, value*) as their search keys, respectively. We call these trees *ValueTree* and *ParentTree* (see fig. 5). Leaves of *ValueTree* and *ParentTree* 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. The solution is employing an additional structure called *bucket*, which stores references to all same data records. In addition, an entry of the leaf points to this bucket. Combining these trees, we have Nested B$^+$-Tree (NBT).

B. 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 hashes of its children. These values are calculated as follows.

a-node : $H_{a\text{-}node} = h(nodeid||xtype||...||value)$ (4)

t-node : $H_{t\text{-}node} = h(h(nodeid||...||value)||\cup_i H_{attr})$ (5)

Leaf of *ValueTree*, *ParentTree*: $H_L = h(\cup_i H_{data\text{-}record})$ (6)

Internal node: $H_I = h(\cup_i H_{child\text{-}node})$ (7)

Leaf node of *NameTree*: $H_{L\text{-}N} = h(H_{vtree} || H_{ptree})$ (8)

Root node of *NameTree*: $H_R = h(\varepsilon || \cup_i H_{child\text{-}node})$ (9)

Where, H_{attr} is hash value of an *a-node* of a given *t-node*. $H_{data\text{-}record}$ is either $H_{a\text{-}node}$ or $H_{t\text{-}node}$ that associated to the link. $H_{child\text{-}node}$ is one of H_L, H_I or $H_{L\text{-}N}$. H_{ptree} and H_{vtree} are hash values of the roots of corresponding *ParentTree* and *ValueTree*. ε denotes a timestamp value and *h()* is a one-way non-invertible hash function (e.g, SHA-1, MD5). 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 ε are broadcasted to all clients.

C. 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 (range query). As mentioned in sec. II.B, two additional boundary records are included in the result set. These entries occupy some adjacent leaves, say $\{L_i, L_{i+1},..., L_j\}$. The

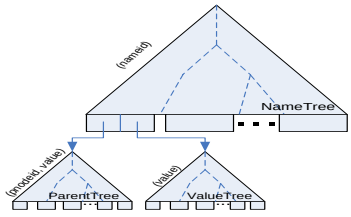

Figure 5. Nested B$^+$-Tree.

server returns data records and hash values of not-in-result entries in L_i and L_j so that clients could re-compute hash values of these leaves. Similarly, the series {Li, Li_{+1},…, L_j} occupies entries in some internal nodes, say {I_x, I_{x+1},…, I_z}. 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, and then verify it with the signature to assure both *correctness* and *completeness*. More detail of *co-path* and *VO* could be found in [8].

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.

D. Select operation

The above shows how to use NMB$^+$-Tree to achieve query authenticity for the simplest situation that returns a single and contiguous record set. It is just a single step among many necessary ones. To answer a query, the server performs some other tasks in a specific order called *execution plan*. Next, let us see some examples of XPath to find out a general method for query processing.

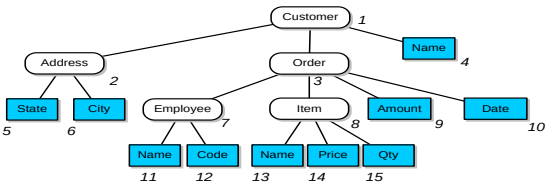

Figure 6. An example of labeled XML schema tree.

As illustrated in fig. 6, the round rectangular nodes stand for *elements* and the sharp corner ones are *attributes*. The number next to node is *nameid* value.

Example 1 is about "List all sold items named 'TV'?". The corresponding query in XPath should be expressed like `/Customer/Order/Item[@name ="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.

These steps could be rewritten as follows:

```
STEP#1 IndexMethod : Vtree, nameID=13
Condition : equal to [TV]
Result level: not included
Retrieval  : node only
StepValue : PNODEID
[For each matched item, perform]
STEP#2 IndexMethod :DirectIDAccess,
  id=ParentStepValue
Result level : 1
Retrieval : node and all its attributes
```

Example 2 deals with a question "List all items bought by Marry?" and its query is `/Customer[@name= "Marry"]/Order/Item`. The server does some similar jobs to obtain satisfied *Customer_1* records. For each *Customer_1*, the server scans on $ParentTree_3$ with *pnodeID* equals *id* of *Customer_1*. The $ParentTree_3$ 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. Similarly, 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 many 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 could 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 a single verification for all returned record sets.

E. 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. The basic idea for this protocol is shown in fig. 7. In general, an update could be treated as a series of delete and insert operations.

Insertion. To insert a new element into database, the owner first serializes it into *t-node* and *a-node*. 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, and 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 data owner. The data owner generates a new timestamp value 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. Then, the owner announces the new timestamp to all clients. If many nodes are inserted at the same time, batch operation could be considered to

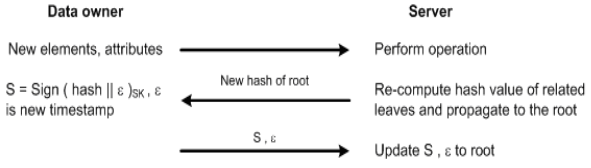

Figure 7. Update protocol.

minimize the root-resigning round.

Deletion. Performing a deletion is similar to an insert. The 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.

VI. ANALYSES AND EXPERIMENTS

Until now, no research work about query assurance of outsourced XML databases has been carried out. Hence, this section theoretically analyzes the solution with respect to storage cost and $\mathcal{VO}$ size. Table I summarizes the notion used in this section.

TABLE I.
FORMULA NOTION.

n	Total number of data item (element and attribute)
s	Number item of result set.
f	*fanout* parameter NMB$^+$-Tree.
$h^T_{min\|max}$	Min/max height of a tree *T*.
$L^T_{min\|max}$	Min/max number of leaves in a tree *T*.
$N^T_{min\|max}$	Min/max of total nodes in a tree *T*.
|sign|	Size of a hash value. (20 bytes for SHA-1)

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 smaller than that of elements, the storage cost for *ValueTrees* and *ParentTrees* dominates the overall storage cost. Formulating the number of *ValueTrees* and *ParentTrees* in a database is very difficult 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 there are one-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. The minimal and maximal numbers of leaves are calculated as formula (10). Hence, we can get the minimum when all leaves are full (*f* entries) and the maximum when they are half-full (*f/2* entries).

$$L_{\min}^{VTree} = \left\lceil \frac{n}{f} \right\rceil, \quad L_{\max}^{VTree} = \left\lceil 2\frac{n}{f} \right\rceil \tag{10}$$

Then we have:

$$h_{\min}^{VTree} = \left\lceil \log_f L_{\min}^{VTree} \right\rceil + 1, \; h_{\max}^{VTree} = \left\lfloor \log_{\frac{f}{2}} L_{\max}^{VTree} \right\rfloor + 1 \tag{11, 12}$$

$$N_{\min}^{VTree} = \left\lfloor L_{\min}\left(\frac{1 - \frac{1}{f^{h_{\min}-1}}}{1 - \frac{1}{f}} \right) \right\rfloor + 1, \quad N_{\max}^{VTree} = \left\lfloor L_{\max}\left(\frac{1 - \left(\frac{2}{f}\right)^{h_{\max}-1}}{1 - \frac{2}{f}} \right) \right\rfloor + 1 \tag{13,14}$$

The overall storage cost is calculated as follows.

$$\begin{cases} C_{\min}^{storage} = N^{NTree} + N_{\min}^{VTree} + N_{\min}^{PTree} \\ C_{\max}^{storage} = N^{NTree} + N_{\max}^{VTree} + N_{\max}^{PTree} \end{cases} \tag{15}$$

In fact, many elements 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 occurrences of *n* in (10) by *n'*. The actual storage cost is achieved by counting the number of index nodes in the database.

VO size. Similarly, only extra information returned to clients to give query authenticity is concerned. The formula (14) is used to calculate the size of a single $\mathcal{VO}$.

$$C^{VO} = |sign| \sum C^{VO}_i, \; i = 1, h^{NMBTre}, \tag{16}$$

Formula of C^{VO}_i is listed in table II.

TABLE II.
FORMULAS ARE USED TO CALCULATE VO SIZE.

	The number of nodes at a depth	**Additional elements in co-path**
h	$L_h = \left\lceil \frac{s+2}{f} \right\rceil$	$C^{VO}_h = f.L_h - s + 2$
h-1	$L_{h-1} = \lceil L_h/f \rceil$	$C^{VO}_{h-1} = f.L_{h-1} - L_h$
…	…	…
h-i	$L_{h-i} = \lceil L_{h-i+1}/f \rceil$	$C^{VO}_{h-i} = f.L_{h-i} - L_{h-i+1}$

Overall cost. The overall cost includes I/O cost, communication cost, CPU cost at both server and client. Experiments are performed in a local environment; therefore, communication cost is omitted. However, this is directly proportional with the $\mathcal{VO}$ 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 the 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: parsing query, planning, fetching, building $\mathcal{VO}$, verifying $\mathcal{VO}$, and re-generating XML text. Parsing phase receives query text from client, builds a corresponding syntax tree, and then passes it to the next state. Planning phase uses the syntax tree came from parsing phase, does some optimization to produce the executing plan. Fetching phase performs the query follow the executing plan, fetches matched data from database. Building $\mathcal{VO}$ phase collects additional information to build the $\mathcal{VO}$, and sends this $\mathcal{VO}$ along with the result set to the client. Verifying $\mathcal{VO}$ phase takes the returned $\mathcal{VO}$ and verifies the result to ensure query assurance. Finally, regenerating phase reconstructs XML text from the answer. The first four phases happen at server, called server-side phases. The last two phases happen at client, called client-side phases. However, the benchmark program ignores two server-side phases, *Parsing* and *Planning*, because they are complex and not affect much to the evaluation.

Experimental results. The experiments are conducted on a P4-2x2.0GHz Windows Vista with 2GB of memory. Benchmarking program is written in VB.NET 2005, .NET Framework 2.0. Data encryption and digital signature are Rijndael and RSA-1024, respectively. The

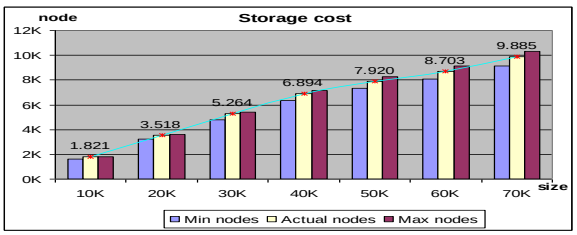

Figure 8. Number of actual nodes compared to min-, max-one.

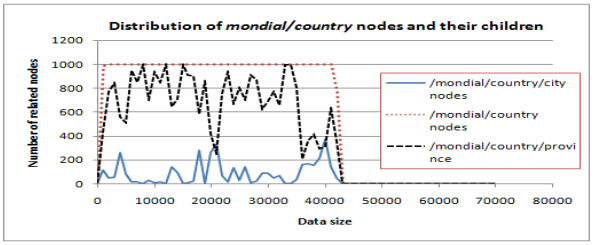

Figure 9. Distribution of *mondial/country* nodes and their children.

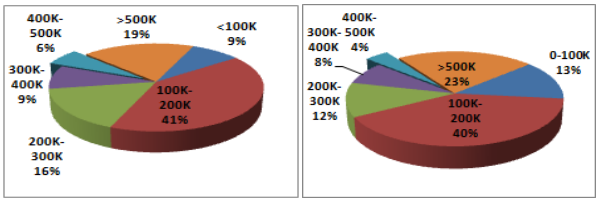

Figure 10. Distribution of */mondial/country/city* (left) and */mondial/country/city* (right) base on their *population* value.

DBMS is MSSQL Server 2005 Express for storing the encrypted database. Moreover, test data are 69,846 items XML document [12], representing world geographic database integrated from the CIA World Fact book, for all the tests. Finally, Windows high-resolution timer APIs are used to measure time cost.

The program reads the XML document, imports its nodes and attributes into an encrypted database. By counting the actual number of tree nodes (both internal nodes and leaves), the program collects the storage cost, which are compared with calculated values show in fig.8 In which, the *Min Nodes* and *Max Nodes* are theoretic nodes calculated by formula (13) and (14).

To evaluate performance, the program carries out certain queries based on the data distribution. Fig. 9 is the datagram of nodes */mondial/country* and their children. The outer dotted line is the number of */mondial/country* nodes, this number is about 60% of the whole document. It is clearly that the query should relate to these nodes. The next dashed line shows the distribution of */mondial/country/province* nodes. The last solid one is that of */mondial/country/city* nodes. Since two children of */mondial/country* nodes, *city* and *province*, are majority, other children are not concerned here. Examining this datagram (fig. 9) helps to choose following queries: */mondial/country/province/city* (name as $\mathcal{P}$) and */ mondial/country/city* (name as $\mathcal{C}$). These queries are performed with three different conditions to assess efficiency of the index structure: *population between 100K and 200K; population between 400K and 500K;* and *population > 500K*. The first condition returns the largest result set; the second produces the smallest one; and finally, the third returns the average one (see fig. 10 for the item distribution based on *population* value).

The program measures three important parameters: result size, I/O cost, and overall execution time. The result size is the number of returned items. Since I/O cost is proportional with amount of data loaded from database, then the number of read nodes could represent this cost. The overall execution time is obtained by subtracting the time when result texts are completely reconstructed by the time when the query processing begins. To understand these measured numbers clearly, we analyze their difference when the database size increases.

Charts in fig.11 show the increase of I/O cost and overall cost in percentage over the database size. These charts have the original at 10K items. Measured values are called V_{10K}. The increases here, of course, are 0%. When database size is up to 20K, measured values are V_{20K}. Then the increases at this point are as follows:

$$\Delta_{20K} = (V_{20K} - V_{10k})/V_{10K} \quad (17).$$

Similarly, we have Δ_{xK}, where x runs from 30K to 70K.

Fig. 11 presents increases of the query $\mathcal{C}$ and $\mathcal{P}$ within three segments of *population* value. They are [100K, 200K], [400K, 500K], and (500K, $+\infty$) which are illustrated from chart A to C, respectively. The *Result size* line is the increase of returned items; the *I/O Cost* line is that of number of items loaded into memory during query processing; and the last *Exec. Query* line is that of overall consumed time to process query. As mentioned in fig. 9, data are distributed from the size 0K to 42K items. Thus, the *Result size* is linear in range (0K, 50K). After that, there is no matched record, so the *Result size* is horizontal. I/O cost and processing time are proportional with result size. So, the *I/O Cost* and *Exec. Query* lines are also linear in range (0K, 50K). In range (50K, 60K), the result size is not vary. However, database size increase does not affect index structure strongly. Thus, the two lines, *I/O Cost* and *Exec. Query*, are nearly horizontal. When the database size is up to 70K, it changes

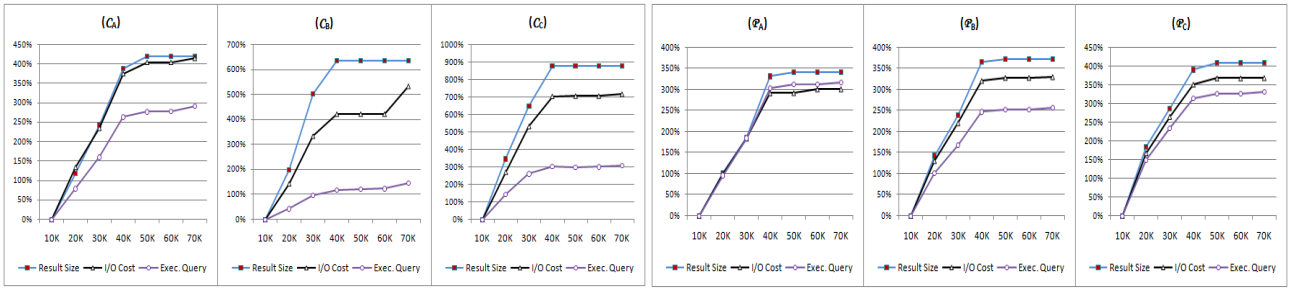

Figure 11. Increases of I/O cost and processing time over database size (query $\mathcal{C}$ and query $\mathcal{P}$)

the index structure. This change makes *I/O Cost* and *Exec. Query* go up a little. However, the *I/O Cost* in chart C_B of fig. 11 has a big jump. This is a special case. Changes of the index structure made by database size increase affect the I/O cost due to specific data distribution.

VII. 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) [13] 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

[1] E.Mykletun, M.Narasimha and G.Tsudik, "Authentication and Integrity in Outsourced Databases", *Proc. ISOC Symp. on Network and Dist. System Security*, USA, 2004.

[2] F.Li, M.Hadjieleftheriou, G.Kollios and L.Reyzin, "Dynamic Authenticated Index Structures for Outsourced Databases", *ACM SIGMOD*, USA, June 2006

[3] H. Hacigümüs, B. Iyer, C. Li and S. Mehrotra, "Executing SQL over Encrypted Data in the Database-Service-Provider Model*", ACM SIGMOD,* USA, Feb 2002

[4] H.Wang, S.Park, W.Fan and P.S.Yu, "ViST: A Dynamic Index Method for Querying XML Data by Tree Structures", *ACM SIGMOD,* USA, June 2003

[5] M. Narasimha and G. Tsudik, "Authentication of Outsourced Databases using Signature Aggregation and Chaining", *Proc. Intl. Conf. on Database Systems for Advanced Applications*, April 2006

[6] M. Xie, H.Wang, J.Yin and X.Meng, "Integrity Auditing in Outsourced Data",*Proc. 33rd VLDB Conf*., Austria, 2007

[7] P. Lin and K.S. Candan, "Hiding Tree-Structured Data and Queries from Untrusted Data Stores", *Proc. 2nd Intl. Workshop on Security in Information Systems*, Portugal, April 2004

[8] P.Devanbu, M.Gertz, C.Martel and S.G.Stubblebine, "Authentic Thrid-party Data Publication", *Proc. IFIP Workshop on Database Security*, The Netherlands, 2000

[9] P.Lin and K.S. Candan, "Secure and Privacy Preserving Outsourcing of Tree Structured Data", *Proc. 1st Workshop on Secure Data Management*, Canada, August 2004

[10] R. Brinkman, L. Feng, J. Doumen, P.H. Hartel, and W. Jonker, "Efficient Tree Search in Encrypted Data", *Information System Security Journal*, 13, 14-21, 2004

[11] R.Sion, "Query Executing Assurance for Outsourced Da-tabases", *Proc. 31st VLDB Conf*., Norway, 2005

[12] Sample dataset World geographic database, http://www.cs.washington.edu/research/xmldatasets/data/mondial/mondial-3.0.xml.

[13] T.K.Dang, "Semantic Based Similarity Searches in Data-base Systems (Multidimensional Access Methods, Similarity Search Algorithms)", PhD thesis, FAW-Institute, University of Linz, Austria, May 2003

[14] T.K.Dang and N.T.Son, "Ensuring Correctness, Completeness and Freshness for Outsourced Tree-Indexed Data", *Information Resources Management Journal (IRMJ)*, Idea-Group Publisher, Jan 2008, in press.

[15] T.K.Dang, "A Practical Solution to Supporting Oblivious Basic Operations on Dynamic Outsourced Search Trees", *Special Issue of International Journal of Computer Systems Science and Engineering (CSSE)*, CRL Publishing Ltd, UK, 21(1), 53-64, Jan 2006

[16] T.K.Dang, "Oblivious Search and Updates for Outsourced Tree-Structured Data on Untrusted Servers", *International Journal of Computer Science and Applications (IJCSA)*, 2(2), 67-84, June 2005

[17] T.K.Dang, "Security Protocols for Outsourcing Database Services", *Information and Security: An International Journal*, ProCon Ltd., Sofia, Bulgaria, 18, 85-108, 2006

[18] T.Shimizu and M.Yoshikawa,"An XML Index on B+ -Tree for Content and Structural Search", 2005, http:/dl.itc.nagoyau.ac.jp/~shimizu/papers/shimizu_dews2005.pdf

Mr. Viet Hung NGUYEN received his IT BEng. degree in 2003 and MSc degree in 2007 from HCMC University of Technology (Vietnam). From 2003-2006, he worked for a local software development company. He experienced in analysis and development of information management system. Since May 2006, he has been working as research assistant in the Faculty of CSE in HCMC University of Technology. His research interests include advanced security and modern information system.

Dr. Tran Khanh DANG obtained his BEng. degree from HCMC University of Technology (Vietnam) in 1998. He got his PhD degree in May 2003 at FAW Institute, Johannes Kepler University of Linz (Austria). From 1998-2000, he worked as a lecturer and researcher in the Faculty of CSE, HCMUT. From 2003-2005, he worked as a lecturer and researcher in the School of Computing Science, Middlesex University in London (UK). He has been working as lecturer and researcher in the Faculty of CSE since Oct 2006. Of late, he founded Advances in Security & Information Systems Lab at the faculty. His research interests include database and information security, similarity search and flexible query answering systems, modern information systems and applications, and distributed systems and parallel processing. He published more than 40 papers in international journal/conferences.