Hiding Tree-Structured Data and Queries from Untrusted Data Stores

Ping Lin and K. Selçuk Candan

In Web and mobile computing, clients usually do not have sufficient computation power or memory and they need remote servers to do the computation or store data for them. Publishing data or applications on remote servers helps improve data availability and system scalability, reducing the client burden of managing data. Application Service Providers (ASPs) and Distributed Application Hosting Services (DAHSs), which rent out storage, (Internet) presence, and computation power to clients with IT needs (but without appropriate infrastructures) are becoming popular. Especially with the emergence of enabling technologies, such as J2EE and .NET, there is currently a shift toward services hosted by third parties.

Those providing data outsourcing service, with their computation power and large memory, can be called data stores or oracles. Typically, these data stores cannot be fully trusted, for we have to realize that with the data outsourced on third-party hosts, entities independent of data/service owners, (1) it is possible for the hosts to illegally utilize sensitive information about data to make a profit; (2) host operators may accept bribes from or be compelled by adversaries to grant insider control or leak sensitive information to them; (3) even if hosts are honest, despite all security meth-

PING LIN is a Ph.D. student in the Computer Science and Engineering Department at Arizona State University. She is interested in information security and distributed information systems. Her current research focuses on data and application security for outsourced services. She has published a book chapter and several conference papers in this research area. She received her M.S. degree from the Institute of Software, Chinese Academy of Sciences in 2000 and her master's thesis was about the implementation of GSM protocols for mobile terminals. She received her B.S. in computer science from Xiangtan University in China in 1993.

K. SELÇUK CANDAN is an associate professor at the Department of Computer Science and Engineering at Arizona State University. He joined the department in August 1997, after receiving his Ph.D. from the Computer Science Department at the University of Maryland at College Park, where he received the 1997 ACM DC Chapter award of the Samuel N. Alexander Fellowship for his Ph.D. work. He has worked extensively on heterogeneous, distributed, and secure data management and integration issues. His research interests also include development of models, query processing, and optimization algorithms for heterogeneous data management systems. He received his B.S. degree, first ranked in the department, in computer science from Bilkent University in Turkey in 1993.

ods (firewalls, etc.) that are employed, chances are that the host can be hijacked, leaking sensitive information about data to the hijacker; or (4) host machines may be physically seized by adversaries. Due to the above possibilities, a data store cannot be treated as trusted party. Thus, in security-critical domains, in addition to traditional security provisions, appropriate methods should be taken to protect sensitive information about data from adversaries at hosts. Without enough security guarantees, data stores cannot survive.

Clients with sensitive data (e.g., personal identifiable data) may require that their data content be protected from such data storage oracles. This leads to encrypted database research, in which sensitive data is encrypted, so the content is hidden from the database. It is defined as content privacy.

Sometimes not only the data outsourced to a data store but also queries are of value, and a malicious data store can make use of such information for its own benefit. This privacy is defined as access privacy. Typical scenarios demanding access privacy include:

- A mineral company wants to hide the locations to be explored when retrieving relevant maps from the IT department map database; and
- In a stock database, the kind of stock a user is retrieving is sensitive and needs to be kept private.

This leads to private information retrieval research, which studies how to let users retrieve information from a database without leaking (even to the server) the location of the retrieved data item.

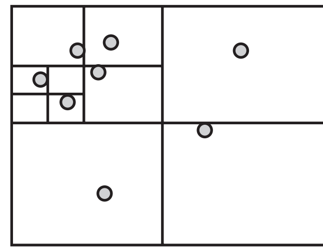
The tree structure is a very important data structure and tree-structured data shows itself in many application domains. In this article, we address outsourcing and hiding of tree-structured data and queries on this data. For this work, we have two motivating applications: hiding XML data that is stored in the form of trees and XML queries in the form of tree paths; and hiding tree-indexed data and queries for the data.

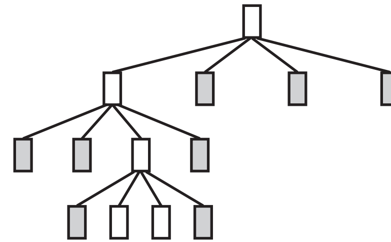
Hiding XML Data and Queries. Some data, such as XML documents, has a natural tree structure. DOM and LORE are two well-known tree data models for XML documents. XML introduces much flexibility into data structures and has become a de facto standard for data exchange and representation over the Internet. Some work has been done on selective and authentic untrusted third-party distribution of XML documents. That work focuses on access control and authentication of document (i.e., query result) source and content. With more and more data stored in XML documents, techniques to hide tree structures (the content and structure of XML documents) from untrusted data stores are in great need. In an XML database, a query is often given in the form of tree paths, such as XQuery. To hide XML queries and the structure of XML documents, clients need to traverse XML trees in a hidden way.

Hiding Index Structures and Queries. In databases, proper index trees are often built for convenient access to data. However, most index structures closely reflect the distribution of the data. For example, the quadtree data structure in Figure 1 demonstrates the close relationship between multidimensional index structures and the underlying data distribution. Such is the case with the B-tree, a tree generally adopted for indexing in DBMSs, which can be seen as a one-dimensional quadtree. Multidimensional index structures are often used in spatial and multimedia databases. Thus, in order to hide the data and data distribution from the database, tree structure hiding techniques must be adopted to protect index trees from oracles. To retrieve an indexed data item, a client needs to traverse the index tree to find the data's location. In order to hide the query and the index tree, the traverse path needs protection.

We concentrate here on hiding tree-structured data and traversal of trees from oracles. Noticing that existing private information retrieval techniques require either heavy replication of the database onto

FIGURE 1 Data Distribution and Its Effect on the Quad Tree: (a) Data Distribution in 2D Space; (b) Corresponding Quadtree Structure


(a) Data distribution in 2D space


(b) Corresponding quadtree structure

multiple noncommunicating servers or large communication costs, we give a single-server tree-traversal protocol that provides a balance between the communication cost and security requirements. To protect the client from the malicious data store, some tasks (such as traversing the tree structures) are delegated to the client.

In the proposed technique, clients learn how to traverse a remotely stored tree structure to locate and retrieve data, while the tree structure and the traversal are hidden from the server. Client responsibilities include encryption and decryption of the data received from the data store during the traversal of the tree. The computation capability required at the client side can be achieved by assistant hardware equipment, such as smartcards, which are cheap (generally no more than several dollars) and now commonly used in mobile environments. In this article, we analyze the overhead incurred by the proposed technique, including communication cost, encryption, decryption cost, and locking cost. Although Chor et al.[4] have argued that information-theoretical private information retrieval cannot be achieved without a significant communication overhead, we discuss methods to minimize those costs in a computational hiding sense.

Related Work: Private Information Retrieval

Private information retrieval aims to hide the address of the target data to be retrieved from the database server in an information-theoretic sense. Consequently, it is closely related to our problem. The concept of private information retrieval was first introduced by Chor et al.[4] They found that if a single copy of the database is used, sending the whole database to the client is the only way to hide information retrieval in the information-theoretic sense. Consequently, replication of the whole database to multiple servers that are not able to cooperate with each other is necessary if less than the whole database is transferred. Later on, Ostrovsky et al.[9] presented a technique by which a user can privately write data into the database. Based on basic private information retrieval techniques, Chor et al.[4] proposed a private information retrieval with a keyword technique to utilize search tree structures to transform keywords into data addresses.

The existing private information retrieval protocols share some limitations and constraints: the heavy communication cost and the need of replication of databases. A trade-off exists between communication and replication: to reduce the communication cost to subpolynomial complexity measured in terms of the size of the database it is necessary to replicate more than a constant number of copies of the database. Furthermore, no communication between the copies is allowed (otherwise copies can cheat in coalition to learn about the data). This is almost impossible in the real world. Protocols built on one-way functions claim

that a single database scheme could be achieved with polylogarithmic communication complexity by lowering the security requirement from information-theoretic privacy to computational privacy. However, these algorithms are very complicated, involving heavy computation at both the client and server. Furthermore, they are built on a binary-bit model of the database, which is difficult to be transformed into a realistic database system.

In this article, we discuss technical challenges to privately retrieve hidden tree-structured data, especially how to let a client traverse a tree structure to find the desired data node, while minimizing the leakage of the tree structure and the target data. We propose a protocol and provide tree traversal algorithms. We show that using the protocol, by adjusting the security and communication cost parameters, we can achieve the required level of computational security with an acceptable communication cost.

Organization of the Article

We first present a general overview of the framework and the outline of the private tree data retrieval protocol. Next we discuss how redundancy enables oblivious traversal of a tree structure, and address the underlying technical challenges, providing an oblivious traversal algorithm. We then give a quantitative analysis of the protocol and discuss how to tune the various system and security parameters to optimize the performance. We show how to apply the protocol for private accessing of outsourced XML documents. We implement the protocol, analyze experiment results, and discuss the amount of security the protocol can achieve, and suggesting ways to improve the security of the protocol in the future. Finally, we conclude.

OVERVIEW OF THE HIDING FRAMEWORK

In this section, we first give a general overview of the hiding framework. We then provide an outline of the proposed hidden data retrieval protocol.

There are three types of entities with different roles in the system: data owners, licensed users, and a data store (oracle). The data owners and licensed users are thin clients (as explained before). A data owner has the right to publish its data in the oracle, and a licensed user has the permission granted by some data owner to retrieve information from the data owner's data storage space in the oracle. The oracle manages data storage spaces, where data and tree structures are stored in a hidden way.

Clients run data encryption algorithms and have initial secret keys for decryption. Encryption algorithms are used to encrypt data and tree structures before sending them to the oracle to ensure that the data content and the data structure are hidden from the oracle. If clients are accessing an outsourced index tree, they have point- or range-queries. If they are accessing outsourced XML trees, they have query patterns. Query patterns are used to traverse a tree structure along paths described by some regular-like expressions. These tasks are accomplished efficiently by thin clients with the help of specialized embedded hardware, such as smartcards, distributed to the licensed user by data owners. Smartcards have often been used in mobile computing. They are relatively cheap, costing no more than several dollars. Such embedded hardware also helps in solving the secret key distribution problem; that is, by distributing smartcards that contain secret keys, a data owner distributes keys to licensed users.

Data Store Architecture

The storage space of the data store is divided into partitions, each owned by a different data owner. We focus here on a single partition.

The unit of storage and access is a node. Each node is identified by a unique Node ID (NID). A client retrieves a node from the data store or server by sending a request that includes the NID of the node.

Because the IDs of the nodes that constitute a tree structure are not known a priori to the clients, there is a special entry node

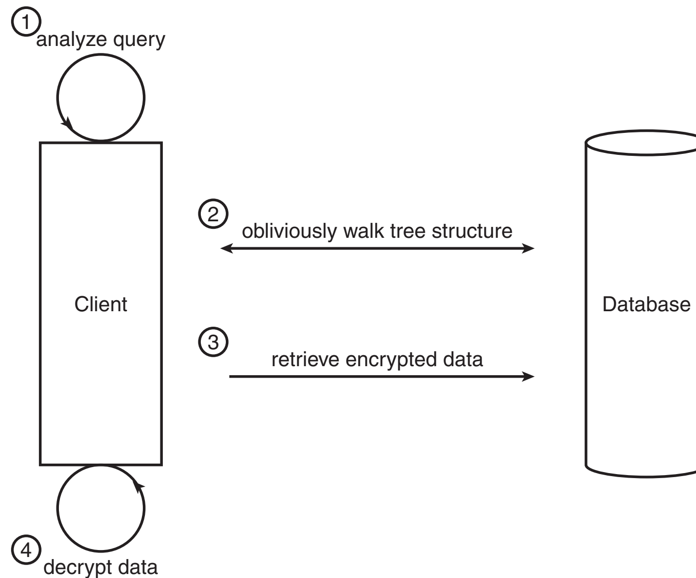


FIGURE 2 Outline of the Protocol

called snode whose NID is known to all legal data owners and licensed users. It stores pointers to roots of tree structures. This node is encrypted by a fixed secret key known to all legal clients. We discuss its fields in greater detail later.

Note that the NID of the snode may be distributed to licensed users in the same way as keys by data owners.

Outline of Private Tree Data Retrieval Protocol

An outline of the private tree data retrieval protocol is depicted in Figure 2.

Every time the data owner wants to insert new data into the tree structure or delete a data item from it, the owner

- Encrypts the data with a secret key;
- Walks the tree structure in an oblivious manner so that the traversal path is hidden to the data store;
- Locates the node of interest (either for insertion or deletion); and
- Updates the tree structure by inserting or deleting encrypted tree nodes in proper positions, in an oblivious way with respect to the data store.

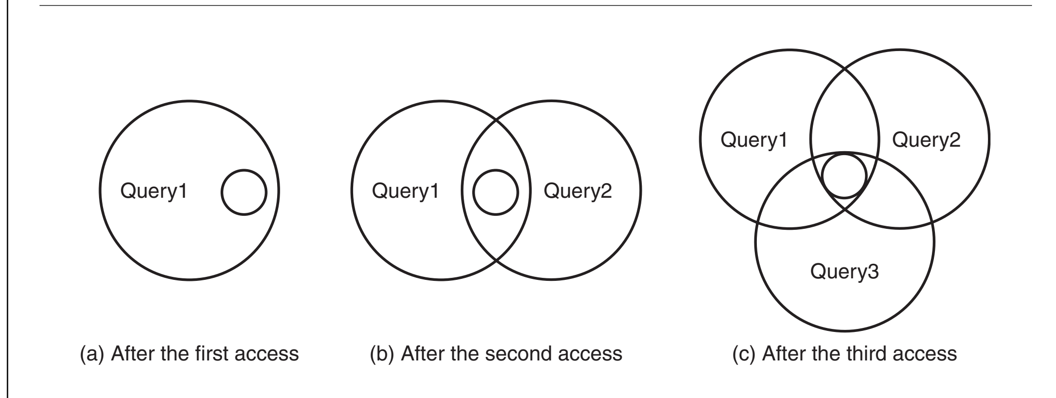
By walking or updating the tree structure in an oblivious way with respect to the data store, we mean minimizing the leakage of information about the data and the tree structure as much as possible; the details of how to walk and update tree structures in an oblivious way are described later.

Client traversal of the tree for retrieving information is similar to update, as in order to prevent the database server from differentiating between read and write operations, a read operation is always implemented as a read followed by a writing back of the contents.

OBLIVIOUS TRAVERSAL OF THE TREE STRUCTURE

Nodes of a tree structure are always encrypted before they are passed to the data store. Consequently their content is already hidden from a malicious store. However, if a client traverses the tree structure in a plain way, the relationships between nodes in the tree, and therefore the tree structure as well as the user's query, are revealed. The paper "Private Information Retrieval with Keywords"[10] presents an oblivious way to traverse a tree structure: it models every

FIGURE 3 Repeated Accesses Reveal the Position of the Target Node: (a) After the First Access; (b) After the Second Access; (c) After the Third Access



level of a database as a separate array, which is then hidden using the basic private information retrieval technique. This scheme is simple, but like other information-theoretical private information retrieval approaches, it needs replication of the whole database across multiple noncooperating data stores and the resulting communication cost is very expensive. Our protocol is to provide computational hiding guarantees with a single data store, while minimizing the communication cost. We propose two adjustable techniques to achieve oblivious traversal of tree structures: access redundancy and node swapping.

Access Redundancy

Access redundancy requires that each time a client accesses a node, instead of simply retrieving that particular node, it asks from the server a set of randomly selected $m - 1$ nodes in addition to the target node. Consequently, the probability with which the data store will guess the intended node is $1/m$. m is an adjustable security parameter. We discuss how to choose the value of m later and define this set as the redundancy set of the target node. We also discuss how a client can choose those m nodes with minimal knowledge about the organization of the whole data storage space.

The problem with redundancy sets, on the other hand, is that their repeated use can leak information about the target node. For example, if the root node's address is fixed, then multiple access requests for the root node reveal its position (despite the use of redundancy) because the root is always in the first redundancy set any client asks. By intersecting all the redundancy sets, the data store can learn the root node. If the root is revealed, there is a high risk that its children (and also the whole tree structure) may be exposed. The situation is depicted in Figure 3 where large circles represent redundant sets for different queries that retrieve the same target node and small circles denote the target node.

Node Swapping

Consequently, in order to prevent the server from using an attack based on intersecting repeated or related requests, we have to move nodes each time they are accessed. Preferably, the move should have minimal impact on the tree structure and should not leak information about where a given node is moved. To achieve this, each time a client needs to access a node from the server, it asks the server for a redundancy set consisting of m nodes that includes at least one empty node along with the target node. The client then

1. Decodes the target,
2. Swaps it with the empty, and
3. Re-encrypts the redundancy set and writes it back.

FIGURE 4 Swapping Enables Movement of the Target Node: (a) After the First Access; (b) After the Second Access; (c) After the Third Access

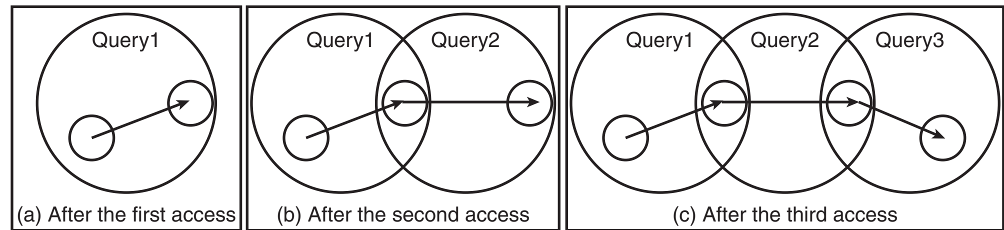


Figure 4 shows how this approach prevents information leakage: Figure 4(a) shows that after the first access, the position of the target node is moved (the arrow shows the node's movement). Figures 4(b) and (c) show that after the second and the third accesses, the position of the target node is moved again. As shown in Figure 4, during the course of an access, the oracle has the chance to know the position of the node only if the redundancy set for the access has little intersection with the set of the previous access so that the position where the node moved to after the previous access is revealed. But because the node moves again once the nodes are written back after the access, such leakage is of no use to the server. In this way, the possible position of the target node is randomly distributed in the data storage space and thus the repeated-access-attack is avoided.

Re-Encryption to Enable Node Swapping

Node swapping requires re-encryption of nodes before they are rewritten to the server. Re-encryption should employ a new encryption scheme/key, because if the same encryption scheme is used, by comparing the content of nodes in the redundancy set after rewriting with their original content, the server can easily identify the new position of the node. This means that a client has to identify how each node is encrypted. We achieve this by adding a new field that contains the secret key for that particular node. This field is always encrypted using a single/fixed secret key. This way, the client can decrypt this field to learn how to decrypt the rest of the node.

Empty Nodes

The node-swapping technique we described uses empty nodes to move the content. This, however, requires allocation and (hidden) maintenance of empty nodes. Obviously, an alternative technique would be to swap the content of two existing nodes, instead of using an empty node. The main disadvantage of the second approach, on the other hand, is that it would require maintenance of parent–child relationships for both swapped nodes. As we discuss in the next section, maintaining node/parent-node relationships is not a trivial task, and maintaining empty nodes is easier than maintaining the parent relationship of both swapped nodes.

HIDDEN TREE TRAVERSAL ALGORITHM

To implement oblivious traversal of a tree structure, some critical issues have to be solved.

- How does a client know NIDs of the empty nodes?
- In order to randomly distribute the target node in the database, other nodes in addition to the target one and the empty one in the redundancy set should be randomly chosen from the database. How can a client randomly choose them?
- After moving one node, in order to maintain the integrity of the tree structure, the parent's pointer to this node has to be

updated accordingly. How can this be performed without revealing parent–child relationships on the tree structure?
- How can consistency of a tree structure be kept when there are many clients accessing it concurrently?
- How can we choose the values of various system parameters, such as the size of the redundancy set?

In this section, we provide techniques to address the first four of these challenges, and provide hidden retrieval algorithms based on them and the underlying protocol. In the next section will discuss the choice of system parameters in greater detail.

Keeping Track of the Empty Nodes

Empty nodes are stored in hidden linked lists. Each node (data + pointer) in such a list is encrypted, so the server can identify neither the nodes that are the empty, nor the structure of the linked list. The encrypted snode, whose address is well known to all legitimate clients, keeps three kinds of pointers: *ehead*s point to the heads of empty node lists, *etail*s point to the tails of empty node lists, and *root*s point to the roots of tree structures. Because all the nodes including the snode are always encrypted when passed back to the server, the server neither learns the empty nodes nor the linked list. Furthermore, because every time an empty node is required, a different empty node is used, the server cannot identify empty nodes through repeated use.

Random Choice of Nodes in Database

When a client requests a node, it asks for a set of m nodes, including the target one. If node swapping is applied, those nodes should include at least one empty one. All nodes other than the target and the empty one in a redundancy set must be randomly chosen from the data storage space so that maximum independence is achieved. To enable a client to choose nodes randomly, the snode should record the range of NIDs of nodes in the data storage space. So by generating $m - 2$ random NIDs within the range and ensuring that among them there is no NID of the target node and the empty node, the client generates a redundancy set.

Maintaining Integrity of Tree Structures

As to the challenge of maintaining node/parent-node relationships after node swapping, we propose two solutions.

The first solution is to find the empty node to be swapped with the child node and update the parent node correspondingly before actually moving the child node; that is, when a client has got a redundancy set for a parent node, and wants a child node, it

- Identifies the NID of the next empty node in the empty node list of the next level; this empty node is the one to be used for swapping the child once it is fetched from the server;
- Changes the parent node's corresponding child pointer to the NID of this empty node;
- Uses another encryption scheme to encode the nodes of the parent's redundancy set and writes them back into server; and
- Asks for a new set of m nodes, which includes the child node, the empty node, and $m - 2$ randomly selected nodes from the next level; they constitute a redundancy set for the child node.

This way, parents are always updated considering the future locations of their children.

The second solution is to let the client keep track of all nodes on the path while it walks down the tree structure, deferring all the updates to a later time. Once the node containing the data to be retrieved is accessed, the client visits the entire path in the reverse order and swaps all nodes on this path.

The benefit of the first solution lies in that client does not need to keep track of the nodes on the path, thus requiring less memory than the second solution. The benefit of the second solution is that, because the client updates the nodes on the tree structure at

the end of the data retrieval (and especially because it updates a parent node right after moving its child node, without any other processing in between) it is easier to maintain the consistency of the database in the case of aborted transactions. Those two solutions also introduce different concurrency penalties. We discuss them in the following subsection.

Concurrency Control Without Deadlocks

The proposed protocol will be applied to Web-based mobile computing environments with a large number of clients. In order to keep consistency of the tree structure with many clients accessing tree structures simultaneously, proper concurrency control must be used at the server side. There has been intensive study about index locking so that maximum concurrency is achieved while preserving the integrity of the tree structure. Because there is no pure read operation in the scheme (each node, after being read, should be written back), only exclusive locks are needed. To prevent deadlocks, we organize nodes in a data owner's data storage space into d levels. Each level of a data owner's data storage space requires an empty node list to maintain empty nodes at this level. The client always asks for locks of parent-level nodes before asking for locks of child-level nodes, and always asks for locks of nodes belonging to the same level in some predetermined order (e.g., in the order of ascending NIDs). In this way, all nodes in a data owner's data storage area are accessed by all clients in a fixed predetermined order. This ensures that circular waits cannot occur. Hence deadlocks are prevented.

These two solutions introduce different concurrency costs. Considering the first solution in which a parent node is updated before its child node is moved, the exclusive locks on nodes of the redundancy set of the parent node can be released immediately after the locks on the nodes of the redundancy set of the child node are gained. Then another client may have the chance to access the parent. In the second solution, because the whole path from the root to the node where data is retrieved is updated in the reverse order of their being accessed, the locks on nodes along the path should also be released in the reverse order of being gained. This means the lock on the root is the first to be gained and the last to be released; thus other clients are prohibited from accessing the tree structure at the same time. Therefore, from the concurrency perspective, the first solution is more desirable.

Pseudo Code of an Oblivious Tree Traversal Algorithm

According to the first tree integrity maintenance solution, we provide the pseudo code of an oblivious traversal algorithm in the following paragraph. The time complexity for this algorithm is $O(d \times m)$, with d denoting the depth of tree storage space and m denoting the redundancy set size, with space complexity $O(m)$.We omit the details of the second solution, but time complexity for it is $O(d \times m)$ and the space complexity is $O(d \times m)$.

Oblivious Trav**ersal Algorithm.** Input: feature values of target data item and the identifier of the data owner from whose tree structure the client wants to retrieve data.

Output: pointer to the node that contains the data if it exists; or null pointer.

1. Lock and fetch the public entry node to the data store, let it be PARENT, find the root, and let it be CURRENT.
2. Select a redundancy set for the CURRENT, lock nodes in the set, and let the empty node in the set be EMPTY.
3. Update the PARENT's pointer to refer to the EMPTY, and release locks on the PARENT level.
4. Swap the CURRENT with the EMPTY.
5. If CURRENT contains the data, return CURRENT

   else

6. Let CURRENT be PARENT, find the child node to be traversed next, let it be CURRENT, and repeat steps 2 through 5.

IDENTIFYING APPROPRIATE VALUES FOR THE SYSTEM PARAMETERS

Choosing the appropriate design parameter values for a hiding system depends on various system constraints, including the acceptable communication cost and the required degree of hiding. We model a data owner's data storage space as d levels. Suppose the tree structure is an l-level tree. Then the following parameters and constraints have to be considered.

- The maximum probability δ for the server to be able to find the actual node that the client is asking from a redundancy set gives us $(1/m) < \delta$.
- The maximum probability λ for the server to find the path along which a client walks the tree structure gives us $(1/m^l) < \lambda$. We emphasize here that although it is easy for the data store to guess the target node from the redundancy set if m is small, it becomes much harder to guess the parent–child relations between sequential node accesses. And the probability of discovering a path is reduced exponentially with the increase of length of the path and hence should be slim even with a small value of m.
- The total communication cost clients are allowed to make for each data retrieval gives us $((\text{read}(m) + \text{write}(m)) \times l \leq \varepsilon$; here read($m$)/write($m$) denotes communication cost to read/write m nodes from the server.
- A node may contain multiple data points. We denote the node size (i.e., the number of data points a node is able to contain) as s. The value of s can be determined by considering the following. Let c denote the function of one round-trip communication cost for data points to be received from and sent to the server, e and d denote the encryption and decryption cost function, and w and r denote the write and read cost function. Theoretically, they are linear functions. Then:

  total_cost_for_data_retrieval
  = tree_depth × m × (communication + decryption + encryption + read + write cost_per_node)
  $= l \infty m \infty (c(s) + d(s) + e(s) + r(s) + w(s))$.

As node size s increases, tree depth l decreases and costs per node increase. If all other parameters are known, we can calculate optimal node size to minimize the total cost. However, as s increases, the probability for the data store to find a path, which is $1/m^l$, increases. Therefore, the value of s should be carefully chosen to ensure that the security requirement is satisfied and the total cost is minimized as much as possible. Note that most of the above constraints are linear; hence an appropriate parameter setting can be easily identified using efficient algorithms.

ENABLING PRIVATE ACCESS TO OUTSOURCED XML DOCUMENTS

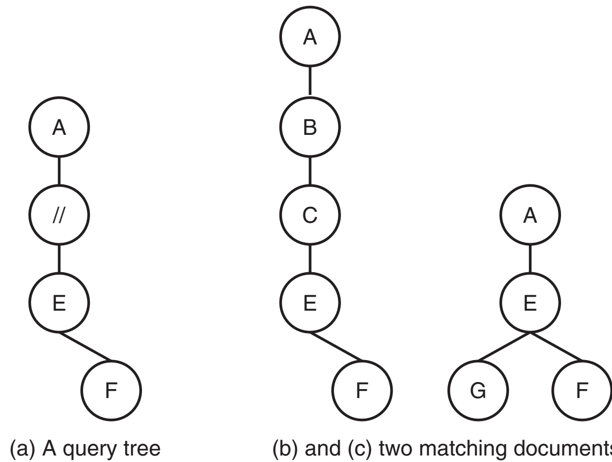
In this section, we describe how our protocol can be extended to provide private access to XML documents.

In an XML document, each node of the tree corresponds to an element or an attribute of an element in the XML document. The root node contains the document's root. A child node corresponds to a subelement or an attribute of the parent node. For each child of a node, in addition to the pointer to the child, there can be a tag in the node that indicates the name of the child node. If the child is a subelement, the name is the element tag of the subelement. If the child is an attribute, the name is the attribute name.

XML query processing is concerned with finding the instances of a given pattern tree (query tree or twig) in a given target tree (XML document or document collection). Given directed labeled trees Q and T, the query tree Q matches the target tree T at a node x iff there exists a one-to-one mapping from the nodes of Q to the nodes of T (with respect to the node x) which preserves the labels, the order, and the ancestor/descendant relationships of the nodes. However, the wildcard symbol * can be used in a query to match any character in the alphabet and // can be used to match any sequence of symbols (Figure 5).

A naive approach to execute tree queries is to navigate from root to leaves to look for

FIGURE 5 (a) Query Tree (\\ Means Match Any Number of Elements); (b) and (c) Two Matching Documents



tree matches. A more advanced approach is to use inverted indexes to access nodes of the trees and then use join operations to match the tree structures.

Unfortunately, structural joins can be very costly. A variety of algorithms, such as ViST, use intervals to index nodes based on element names and use these intervals to perform structural join operations. Our protocol can be extended to both types of accesses.

Hiding Navigational Accesses to XML Data

The first kind of access is the naive navigational access that involves explicit traversal of the tree structure. We consider a path-based query that may be expressed using a regular expression plus predicates that filter nodes. Figure 6 gives an example XML document.

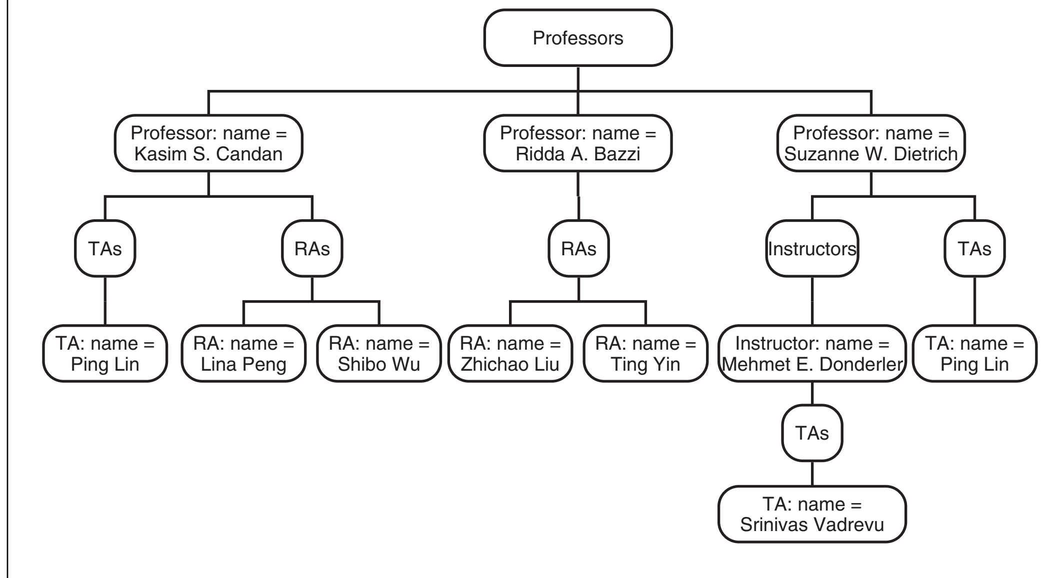
Suppose a licensed user has an XQuery with the Xpath description: "*professors/Professor*[*name*=*"K.S.Candan"*]*/RAs/RA*" which aims to find all RAs of the professor named "K.S. Candan." For this document, the user checks the root element and finds that the root node matches the element *professors*. The user would then retrieve the children of the root and check if any is of type *Professor*. If any of the children satisfies this condition, then the user would further evaluate the filtering predicate and pick the *Professor* nodes with the name attribute "K.S.Candan." The user would then continue navigating down the tree, checking at each step the element types and filtering conditions, until the required *RA* elements are retrieved.

Navigational access is very similar to the tree traversal we described earlier. To hide this type of access, the client would first find where the root node is by checking the snode. Then it would generate a redundancy set for the root. It would retrieve the nodes in the redundancy set from the data store and would identify the candidate nodes to be evaluated at the next stage. This way the navigation (or tree traversal) process would continue until the user visited all relevant nodes in the tree.

Hiding Join-Based Accesses to XML Data

The second kind of access is through structural join operations. The general idea of a structural join is: (1) label the XML tree node so that it is straightforward to tell the structural relationship (parent/child or ancestor/decedent) between any two nodes by labels. A typical example of such a label is the interval. By interval labeling, XML trees are traversed in pre-order and each node is assigned an interval such that every

FIGURE 6 Example XML Document



child's interval is contained by the parent's interval. (2) Build an index tree on element tags or labels to facilitate fast retrieval of required XML nodes. (3) For a structural join XML query, split it into the proper set of subqueries and execute them. (4) Use labels to structurally join subquery results to gain the XML query result.

Different structural join algorithms differ in (a) how they assign intervals to nodes, (b) how they split queries into its subqueries to minimize query cost, (c) how they select the join order, (d) how they index the tags and intervals, and (e) how they perform the structural join operations. However, a common property of most approaches is that, to speed up the structural join operations and to process tree queries efficiently, these algorithms use tree-based index structures, such as B^+-trees, to store node intervals. Such index structures facilitate efficient retrieval of data. Improperly accessing the index, not only the structure of the index and its contents, but also the data distribution, for most indexes generally closely reflect the distribution of the data.

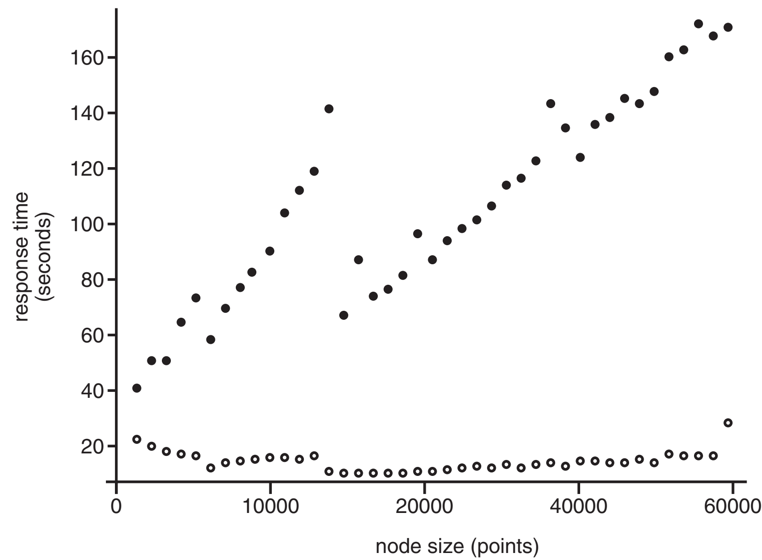
Thus, in order to hide the data and data distribution from the database, tree-structure hiding techniques must be adopted to protect the access of index trees by oracles. To retrieve an indexed data item, a client needs to traverse the index tree to find the data's location. In order to hide the query and the index tree, the traversal path needs protection. Therefore, access to tree-structured index structures can also benefit from the oblivious tree traversal techniques we have previously described.

EXPERIMENT RESULTS

To validate the protocol, we simulated the protocol with the first tree traversal algorithm and conducted some experiments. We do not present results here regarding the second algorithm, as its concurrency cost is much higher. Therefore, we focus on the better of the two. The computing environment consisted of a Linux server acting as a data store and a 1.0 Ghz/256 M laptop generating client requests. They were connected via a wireless LAN system. We implemented a two-dimensional $k - d$ tree as the index structure due to its simplicity. This simple structure enables us to observe experiment results more effectively.

In the article, we do not experiment with range queries inasmuch as we focus on path traversal. We point out that using this protocol, range queries can be implemented as

FIGURE 7 Response Time and Node Size



multiple path traversals without deadlocks. We generated 40,000 data points that were uniformly distributed in the region (0, 0) to (1,000,000, 1,000,000), and stored them in a data storage space with 30,000 node capacity. The size of the redundancy set *m*, is set to 8.

Response Time and Node Size

We executed a set of experiments to show the relationship between node size and response time, that is, the time between a client sending a data retrieval request and getting the response.

Figure 7 shows the two sets of experiment results. The dark points denote the results of experiments with encryption/decryption implemented by software. This set shows that when node size is set to around 50 data points, the minimum response time (about 38 s), is achieved. This phenomenon verifies the theoretic observation that there must be an optimal node size. Considering the probability for the malicious server to find the path (denoted as path probability, which is a function of page size, $1/m^{\log(num/s)}$. Here m is the redundancy parameter, *num* is the total number of data points stored, and s denotes node size), a suitable node size can be chosen to satisfy security requirements and minimize response time.

The set of white points depicts experiments with efficient hardware encryption/decryption. From the result, we found that encryption and decryption constitute heavy cost and with assistant hardware, response time can be greatly reduced to about 8 s.

To compare our protocol with the one-server Private Information Retrieval (PIR) technique, we also simulated PIR by transferring the whole database to a client. It takes about 3643 s to finish transferring. We can claim that our protocol is much more efficient.

Another interesting phenomenon we observe from Figure 7 is that although the two sets of points have a big difference in their values, they have a similar zigzag pattern. This shows that the discontinuities and sharp variances in response time values are mainly determined by other costs (communication cost $c(s)$, write cost $w(s)$, and read cost $r(s)$) than encryption/decryption.

We also notice that the response time for the set of black points has a strong tendency to increase with the node size, whereas it is

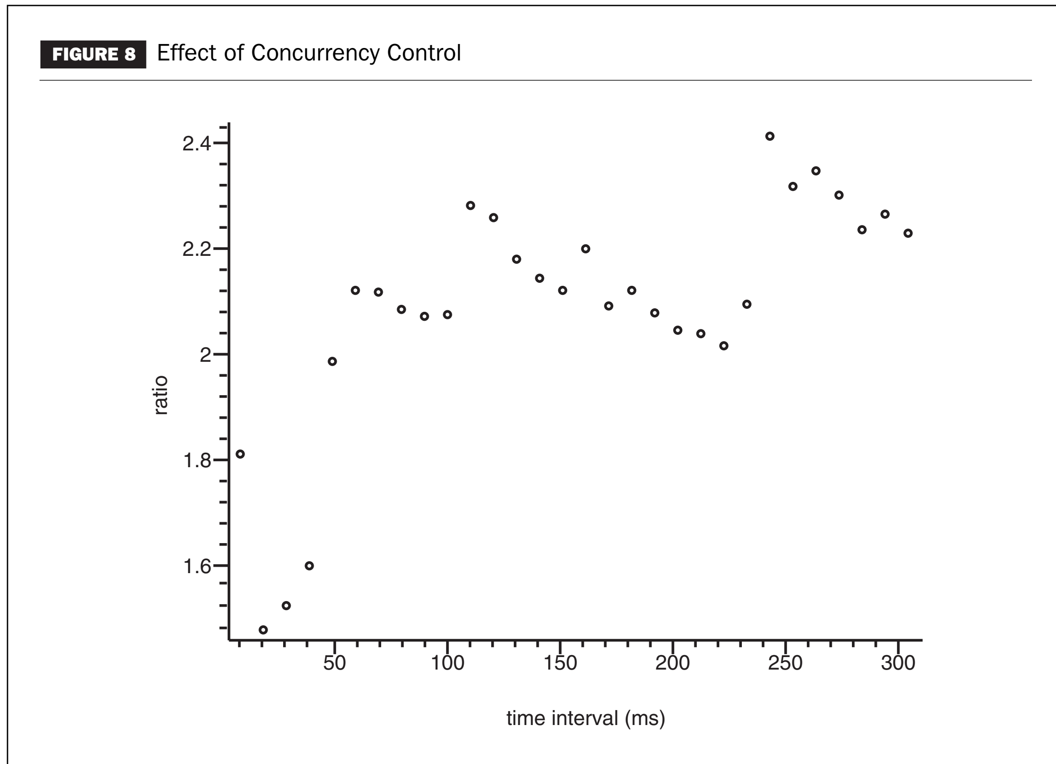


FIGURE 8 Effect of Concurrency Control

very slight for the white points. This can be explained by the significant parts encryption/decryption play in the total cost and their linear increase with the node size.

Concurrency Control

Furthermore, we conducted a set of experiments to show the effect of concurrency control. In this set of experiments, 50 retrieval requests for independently selected random data points were launched one by one at varying frequency every 10 ms to every 300 ms. In the experiment results, we found no deadlocks. We also found that the total time to finish all the requests was much less than letting the server process those retrievals sequentially. To give a sample result, when requests were launched every 20 ms, the total time required to finish them was 734.8 s, and the time to process them sequentially was 1442.9 s.

Figure 8 gives the ratio of the time required to process sequentially and the time required by our protocol with concurrency control. We can see that the ratio is about 2. This means we gain 100 percent savings with concurrency control. Figure 8 also shows that this ratio increases with the time interval. This is consistent with the common knowledge that the efficiency of the data store is reduced with more clients simultaneously accessing trees.

SECURITY ANALYSIS AND FUTURE WORK

To ensure computational private security, the protocol should be able to protect queries and the tree data structure from a polynomial-time server. To study the security guarantee the protocol provides, suppose that the server keeps a history of all redundancy sets users retrieved, and the server makes inferences about queries and data by statistical analysis of the history. We define each redundancy set as a call, and the history as a view of the server. The required amount of security is defined as:

- For any two different queries $Q1$ and $Q2$ posed in the view, the distributions of their sequences of calls are indistinguishable in polynomial time; and
- For any two queries $Q1$ and $Q2$ posed in the view, it is hard to tell if they are identical by observing their sequences of calls.

We first assume that:

- Queries are uniformly distributed; that is every tree node is accessed by clients as the target data to be retrieved with the same probability; and
- Every client can pose calls anonymously; that is the identity of the clients can be hidden from the data store.

We remove the first assumption later. As to the second one, anonymous access is achievable with anonymous access protocols. Our proof that the protocol satisfies both levels of the security definition is based on the following proposition.

Proposition. If the data storage space is randomly initialized and queries are uniformly posed, tree nodes will always be uniformly distributed in each layer of the data storage space.

Proof. Suppose there are a total of N nodes in a level of the data storage space, and there are n tree nodes at the layer. We use the N-bit random variable $X(t)$ to represent the distribution of tree nodes in the level at t so that:

$X(t)_i$ = 1, if the ith node in the level stores the tree data
0, otherwise, for $i \in [1..N]$.

Here $X(t)_i$ denotes the ith bit of $X(t)$. It is obvious that $\sum_i X(t)_i = n$. Let S be $X(t)$'s state space; that is, $S = \{s \mid$ s is an N bit binary number such that $\sum_i s_i = n\}$. There should be C_N^n elements in S. Now we prove by induction that if queries are uniformly distributed and data storage space is randomly initialized, tree nodes will always be uniformly distributed in each level after queries.

First, **after** the data storage space is randomly initialized, tree nodes are distributed uniformly at each level. Let $X(0)$ be the initial state of the level, then:

$$PROB[X(0) = s] = 1/C_N^m, \quad \forall\ s \in S.$$

Suppose that $X(t)$ (i.e., the state of the level at t) conform to uniform distribution and there is a uniformly distributed random query to switch its target tree node with a randomly selected empty node at the level. Hence at time $t + 1$:

$$PROB[X(t+1) = s] = \sum_{s' \in S} PROB[X(t) = s'] \times PROB[X(t+1) = s | X(t) = s'].$$

Let $D(s,2) = \{s' \mid s' \in S$ and the hamming distance between s' and s is 2. It includes all possible states of $X(t)$ if $X(t+1)=s$. Hence:

$$PROB[X(t+1) = \sum_{s' \in D(s,2)} PROB[X(t) = s'] \times PROB[X(t+1) = s | X(t) = s'].$$

Consider that $X(t)$ conforms to uniform distribution, $|D(s,2)| = C_n^1 \times C_{N-n}^1$, and given an s' in $D(s,2)$, there are $C_n^1 \times C_{N-n}^1$ numbers of ways to perform a single random switch, and only one of them enables transition of the state of the level from s' to s, the equations becomes:

$$PROB[X(t+1) = s] = \frac{1}{C_N^n} \times C_n^1 \times C_{N-n}^1 \times \frac{1}{C_n^1 \times C_{N-n}^1} = \frac{1}{C_N^n} \quad \forall s \in S.$$

Corollary. If the data store is randomly initialized and queries are uniformly distributed, redundancy sets are also uniformly distributed.

The proof of the corollary is quite similar to the proof of the proposition and we omit the details here.

Under the assumption of uniform distribution of queries, for two different queries, if

their query path lengths are equal, the distribution of their sequences of calls (redundancy sets) are identical (uniform distribution), and hence indistinguishable in polynomial time; if their query path lengths are not equal, clients can execute dummy calls at deeper levels to always make the same number of calls.

As to the second security requirement, if two identical queries are posed consecutively without any interfering calls, their calls at the same level will always intersect; hence it seems possible for data stores to deduce some hints about identical queries from intersections. However, under the assumption of uniform distribution of queries, the probability for two identical queries to occur consecutively, which is $1/N^d$ (here d denotes the depth of the tree), is smaller than the probability for any two sequences of random sets to intersect, which is $(1 - C_{N-m}^{m} / C_{N}^{m})^d$. Because every call includes some randomly selected nodes, the intersections introduced by identical queries will be perfectly hidden by intersections introduced by random nodes.

However, under other assumptions of query distribution, our protocol may not provide the required security guarantee. For example, if a query Q occurs at a very high frequency, the intersections introduced by random nodes cannot hide the intersections between consecutive Qs. We study how to improve the protocol under other query distributions by methodically introducing dummy calls and intersections to make other query distributions appear uniform in the view.

CONCLUSION

In this article, we propose a simple, adaptive, and deadlock-free protocol to hide tree-structured data and traversal paths from a data store. Because many data such as XML have a tree structure and queries can be expressed as traversal paths, this protocol can be utilized to hide such data and queries. XML documents can be seen as trees. Research has been done to study access control, content hiding, and authentication of XML documents, but none has been done on hiding XML queries and structures. We believe that this is the first protocol to address this need. Compared with existing private information retrieval techniques, our protocol does not need replication of databases and it requires moderate communication, and is thus practical. We show how to apply it to hide XML documents and tree-path-based queries. We conduct experiments and observe that the proposed techniques achieve hiding without generating unacceptable concurrency problems. Finally, we give a security analysis of the protocol and point out future research directions.

Note

This work is supported by the AFOSR grant #F49620-00-1-0063 P0003.

References

1. Hacigümüs, H., Iyer, B.R., Li, C., and Mehrotra, S. (2002). Executing SQL over Encrypted Data in the Database-Service-Provider Model. In *Proceedings of 2002 ACM SIGMOD International Conference on Management of Data,* Madison, WI, June 3–6, 2002, 216–227.
2. Oracle Corp. (1999) Database Security in Oracle8i, 1999. Retrieved on February 26, 2004, from http://otn.oracle.com/depoly/security/oracle8i/index.html.
3. Smith, S. W. and Safford, D. (2001). Practical Server Privacy with Secure Coprocessors. *IBM Systems Journal,* 40(3), 683–695.
4. Chor, B., Goldreich, O., Kushilevitz, E., and Sudan, M. (1995). Private Information Retrieval. In *Proceedings of the 36th IEEE Conference on the Foundations of Computer Sciences,* Milwaukee, WI, October 23–25, 1995, 41–50.
5. Bouganim, L. and Pucheral, P. (2002). Chip-Secured Data Access: Confidential Data on Untrusted Servers. In *Proceedings of the 28th Very Large Data Bases Conference,* Hong Kong, 2002, 131–142.
6. Bayer, R. and Schkolnich, M. (1977). Concurrency of Operations on B-Trees. *Acta Informatica,* 9, 1–21.
7. Mohan, C. (1996). Concurrency Control and Recovery Methods for B+-Tree Indexes: ARIES/KVL and ARIES/IM. In V. Kumar, (Ed.) *Performance of Concurrency Control Mechanisms in Centralized Database Systems,* Prentice-Hall, Englewood Cliffs, NJ, 248–306.
8. Mohan, C. (2002). An Efficient Method for Performing Record Deletions and Updates Using Index Scans. In *Proceedings of the 28th Very Large Data Bases Conference,* Hong Kong, 2002, 940–949.

9. Ostrovsky, R. and Shoup, V. (1997). Private Information Storage, In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing,* El Paso, TX, May 4–6, 1997, 294–303.
10. Chor, B., Gilboa, N., and Naor, M. (1997). Private Information Retrieval by Keywords, Technical Report TR CS0917. Technion, Israel.
11. Lin, P. and Candan, K.S. (2003). Data and Application Security for Distributed Application Hosting Services. In M. Fugini and C. Bellettini (Eds.), *Information Security Policies and Actions in Modern Integrated Systems.* 273–316.
12. Lin, P., Candan, K.S., Bazzi, R., and Liu Z. (2003). Hiding Data and Code Security for Application Hosting Infrastructure. In H. Chen et al. (Eds.), *Intelligence and Security Informatics,* 388.
13. Lin, P. and Candan, K.S. (2003). Hiding Traversal of Tree Structured Data from Untrusted Data Stores. In H. Chen et al. (Eds.), *Intelligence and Security Informatics,* 385.
14. Candan, K. S., Jajodia, S., and Subrahmanian, V. S. (1996). Secure Mediated Databases. In *Proceedings of the IEEE Conference on Data Engineering,* 490–501.
15. Paparizos, S., Patel, J.M., Srivastava, D., Wiwatwattana, N., Wu, Y., and Yu, C. (2003). TIMBER: A Native XML Database for Querying XML. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data,* San Diego, CA, June 9–12, 2003, 672.
16. Wang, H., Park, S., Fan, W., and Yu, P.S. (2003). ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data,* San Diego, CA, June 9–12, 2003, 110–121.