International Journal of Computer Science & Applications
Vol. II, No. II, pp. 67 - 84

Oblivious Search and Updates for Outsourced Tree-Structured Data on Untrusted Servers

DANG Tran Khanh
School of Computing Science
Middlesex University
London, United Kingdom
k.dang@mdx.ac.uk

Abstract

Although tree-based index structures, such as B+-trees, R-trees, have proven their advantages to both traditional and modern database applications, they introduce numerous research challenges as database services are outsourced to untrusted servers. In the outsourced database service (ODBS) model, the crucial security research questions mainly relate to data confidentiality, data and user privacy, authentication and data integrity. To the best of our knowledge, however, none of the previous research has radically addressed the problem of preserving privacy for basic operations on such outsourced search trees. Basic operations of search trees/tree-based index structures include *search* (to answer different types of queries) and *updates* (modification, insert, delete). In this paper, we will discuss security issues in outsourced databases that come together with search trees, and present techniques to ensure privacy in the execution of these trees' basic operations on the untrusted server. Our techniques allow clients to operate on their outsourced tree-structured data on untrusted servers without revealing information about the query, result, and the outsourced data itself.

Keywords: Outsourced search trees, data and user privacy, oblivious search and updates, encrypted data, untrusted servers.

1. Introduction

Advances in the networking technologies and continued growth of the Internet have triggered a new trend towards outsourcing data management and information technology needs to external service providers. Database outsourcing is a recent manifestation of this trend [29]. In the outsourced database service (ODBS) model, clients rely on the premises of the provider, which include hardware, software and manpower, for the storage, maintenance, and retrieval of their data. This ODBS model introduces numerous research challenges and thus has rapidly become one of the hot topics in the research community [15, 31, 24, 6, 13].

As mentioned, in ODBS model, a client (e.g., an organization) stores its private data at an external service provider, who is typically not fully trusted. Therefore, securing outsourced data, i.e. make it confidential, is one of the foremost challenges in this model. Basically, regardless of the untrusted server at the provider's side, the final goal that clients want is that they can use the outsourced database service as an in-house one. This includes a requirement that clients can operate on their outsourced data without

worrying about leak of their sensitive information. This requirement in turn poses several additional challenges related to privacy-preserving for client's queries as well as for the outsourced data during the execution of operations at the untrusted server. Overall, with an assumption that client's side is trusted[1], the following security requirements must be met:

- *Data confidentiality*: outsiders and even the server's operators (database administrators) cannot see the client's outsourced data contents in any cases (including when the client's queries are performed on the server).
- *User privacy*: clients do not want the server to know about their queries and the returned results.
- *Authentication and data integrity*: clients must be ensured that data returned from the untrusted server is originated from the data owner and has not been tampered with.

The above security requirements are different from the traditional database security issues [7, 34] and will in general influence the performance, usability and scalability of the ODBS model. Among the three, the last security objective (i.e. authentication and data integrity) is out of the scope of this paper and we refer interested readers to a recent publication [29] for more details. In this paper, we concentrate on addressing the first two security objectives for the outsourced databases that come together with search trees as discussed below.

These first two security objectives are obviously related to each other. To deal with the data confidentiality issue, outsourced data is usually encrypted before being stored at the external server. Although this solution can protect the data from outsiders as well as the server, it introduces difficulties in querying process: It is hard to protect the user privacy as performing queries over encrypted data. The question is *"how will the server be able to perform client's queries effectively, efficiently and obliviously over encrypted data without revealing any information about both data and queries?"*. The problem has been quite well-solved (even without the help of some special hardware[2]) if the outsourced data contains only encrypted records in tables and no tree-based index structures are used for the storage and retrieval purposes [10, 16, 23, 6, 15, 33]. However, no solution has been developed to radically solve the problem in case such search trees are employed although some preliminary proposals have been made as [13, 28, 17, 16]. In our previous work on this topic [13], we did propose an extreme protocol for this ODBS model based on private information retrieval (PIR)-like protocols [2]. However, it will become prohibitively expensive in case only one server is used to host the outsourced data [8]. In Damiani et al.'s work [17, 16], they gave a solution to query outsourced data indexed by B+-trees. Their approach, however, does not provide an oblivious way to traverse the tree and this may lead to compromise the security objectives [13, 28]. Of late, Lin and Candan [28] introduced an approach to solve the problem with a computational complexity security algorithm and the experimental results reported are sound. Unfortunately, their solution only supports oblivious *search* operations on the outsourced search trees, but *insert*, *delete*, and *modification* ones. That means their solution can not be applied to dynamic outsourced search trees where new items may be inserted into and removed from, or existing data can be modified. More importantly, as we will discuss later, applying their solution directly to such dynamic trees may lead to leak of information about the queries and the tree structure so the security objectives are compromised. In this paper, we will analyse and introduce techniques to solve the concerned problem completely.

The rest of this paper is organized as follows: Section2 briefly introduces basic aspects of search trees and their important role in modern database applications. Section 3 summarizes Lin and Candan's approach to oblivious traversal of the outsourced

[1] As presented in [20, 15, 13] there are some cases in which clients are not allowed to receive data that does not belong to the query result. In these cases, clients are considered as untrusted ones and more *data and user privacy-preserving* requirements must be taken into account (cf. section 5).

[2] IBM has been developing special security hardware equipment called secure coprocessors that can support secure computations at both client and server sides [31].

search trees: access redundancy and node swapping techniques. Section 4 is dedicated to presenting our contributions to solve the problem radically. Section 5 presents other related work and discussions. Eventually, we give conclusions and present future work in section 6.

2. Tree-based Index Structures and Applications

Although different types of search trees/tree-based index structures do exist, their basic structure is essentially organized into *nodes*, which are usually stored in disk pages each at the storage media. Each node in the tree, except for a special one called the root, has one parent node and several (may be zero) child nodes. The root node has no parent and a node that does not have any child nodes is called a leaf node. A node is called an internal node if they are neither the root nor a leaf node. Within each node, there are *pointers* connecting this node to others. The number of child nodes in an internal node is called the *fanout* of that node. The role of leaf nodes is to keep data object identifiers or even data objects themselves. Besides, a node usually contains some kind of stored information used to guide the search for a particular data item.

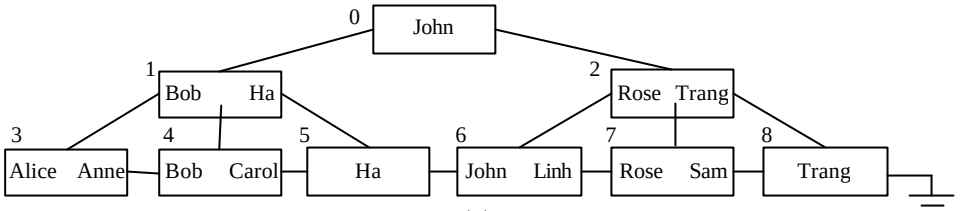
Generally, a search tree is a typical type of tree that is used to guide the search for data items, given some search criteria. Basically, the search trees are multilevel indexes which can be employed to cluster many types of data, ranging from one-dimensional to multi-dimensional/spatial data sets. They have played a fundamental and vital role in both traditional and modern database application domains. As an example, B+-trees among the most traditional search trees have become a commonly accepted default structure for generating indexes on demand in most DBMSs. Coming along with the development of real world applications as well as theoretical research problems, many multi-dimensional access methods (MAMs) have also been invented [19, 4, 11, 12]. They are an indispensable factor for modern database applications domains due to their efficiency in managing the storage and retrieval of data. Some specific examples of typical real-world application areas comprise information retrieval (IR), data mining, multimedia databases, digital libraries, data compression, time-series databases, CAD database systems, geographical information systems (GISs) and tourist information systems (TISs), and many others come together with the fast continued development of computer science (see [12] for more detailed discussions).

Basic operations on the search trees include search (to answer different types of queries) and updates (modification, insert, delete). In practice, insert and delete operations are the most critical operations, which heavily form the structure of the resulting search trees and the achievable performance [4]. An update activity is in common viewed as a consequence of insertions or/and deletions except for advanced update operations such as dynamic bulk-loading operation [12]. In conclusion, the insert and delete operations of (dynamic) search trees used for indexing *dynamic databases* influence the performance, usability and scalability of the applications depending on those databases. As we will show later, the previous work of Lin and Candan fails to protect privacy for the users and such dynamic outsourced databases.

3. Oblivious Traversal of Outsourced Search Trees: Access Redundancy and Node Swapping Techniques

As private data is outsourced, it should be encrypted to ensure the data confidentiality. When this outsourced data is indexed using some search tree, although the private data is in general stored at leaf nodes, information of the whole tree structure and the outsourced data should all be confidential. Otherwise, relations between the data items as well as the data distribution will be revealed, hence the data confidentiality and user privacy will be compromised. As shown in [16], encrypting each tree node as a whole is preferable because protecting a tree-based index by encrypting each of its fields would

disclose to the untrusted server the ordering relationship between the index values. Lin and Candan's approach also follows this solution. Moreover, like other approaches [13, 17, 16], the unit of storage and access in their approach is also a tree node. Each node is identified by a unique node identifier (NID). The original tree is then stored in the server as a table with two attributes: NID and an encrypted value representing the node content. A client retrieves a node from the server by sending a request including the NID of the node. Let's see an example: Figure 1(a) shows a B+-tree built on an attribute *CustomerName* with sample values; Figure 1(b) shows the corresponding plaintext and encrypted table used to store the B+-tree at the external server. As we can see, that B+-tree is stored at the external server as a table over schema B+EncryptedTable= {NID, EncryptedNode}.



(a)

B+Table

NID	Node
0	(1,John,2,-,-1)
1	(3,Bob,4,Ha,5)
2	(6,Rose,7,Trang,8)
3	(Alice,Anne,4)
4	(Bob,Carol,5)
5	(Ha,-,6)
6	(John,Linh,7)
7	(Rose,Sam,8)
8	(Trang,-,-1)

B+EncryptedTable

NID	EncryptedNode
0	D0a1n2g3Kh75nhs&
1	T9&8ra§ÖÄajh³q91
2	H&$uye"µnÜis57ß@
3	L?{inh*ß²³&§gnaD
4	Wh09a/[%?Ö*#Aj2k
5	j8Hß}[aHo$§angµG
6	#Xyi29?ß~R@€>Kh
7	~B³!jKDÖbd0K3}%§
8	T-§µran&gU19=75m

(b)

Figure 1: An example of the B+-tree on an attribute *CustomerName* with fanout 3 (a) and the corresponding plaintext and encrypted table used to store the B+-tree at the external server (b)

Based on the above settings, Lin and Candan proposed an approach to oblivious traversal of outsourced search trees using two adjustable techniques named access redundancy and node swapping. We summarize these two techniques and related important procedures, and then discuss issues arisen from this approach below.

Access Redundancy: This technique requires that whenever a client accesses a node, called the target node, it asks for a set of m-1 randomly selected nodes in addition to the target node from the server. By this access redundancy, the probability that the server can guess the target node is 1/m. Here, m is an adjustable security parameter and the authors also gave discussions about how to choose the value of this parameter in the paper. The access redundancy technique can be viewed as a computational security version of computational PIR-like protocols [2], where information-theoretical security objectives can be traded off against the performance [1]. This technique is also different from those presented in [17, 16], where only the target node is retrieved (this may lead to reveal the tree structure as shown in [13]).

Apart from the redundancy in node access, this technique also bears another weakness: it can leak information about the target node. This is easy to observe: multiple access requests for the root node will reveal its position by simply calculating the intersection of the redundancy sets of the requests. If the root position is disclosed, there is a high risk that its child nodes (and also the whole tree structure) may be exposed [28]. This deficiency is overcome by secretly changing the target node's address after each time it is accessed.

Node Swapping: Each time a client requests to access a node from the server, it asks the server for a redundancy set of m nodes consisting of at least one *empty* node along with the target one. The client then (1) decrypts the target node; (2) manipulates its data; (3) swaps it with the empty node; and (4) re-encrypts nodes in the redundancy set and writes them back to the server. As presented in [28], with this technique, the possible position of the target node is randomly distributed over the data storage space at the untrusted server, and thus the weakness of the access redundancy technique is overcome.

Note that, in order to prevent the server from differentiating between read and write operations, a read operation is always followed by a write operation for all nodes in the redundancy set back to the server. This technique, in turn, requires re-encryption of nodes using a different encryption scheme/key before they are rewritten to the server. Otherwise, the server can easily identify the new position of the target node (and also of the empty node swapped with the target one). More detailed discussions of the re-encryption technique are presented in the original paper.

Additional Procedures: To realize oblivious traversal of outsourced search trees, some more critical issues must be addressed. We briefly discuss these issues as follows:

- *Managing root node address:* In order to traverse a search tree, clients must first know the root node address, which can be dynamically changed by the node swapping technique. However, this information is not known a priori to clients. The authors proposed a solution by employing a special entry node called SNODE whose NID is known to all clients (and even to the server). This node is encrypted by a fixed secret key known to all legal clients (but not to the server). It keeps pointers ROOTS pointing to the root nodes of all outsourced search trees that the client can access.
- *Managing empty node lists:* Empty nodes are stored in hidden linked lists. Each of the empty nodes (data + pointer) is also encrypted. To help clients find out the empty nodes, two other types of pointers are also stored in the SNODE: EHEADS and ETAILS point to the heads and the tails of empty node lists, respectively[3].
- *Managing random choice of the redundancy set:* When a client requests a node, it asks for m nodes consisting of the target node, an empty node, and m-2 randomly selected nodes. To enable clients to do this, the SNODE records the range of NIDs of nodes in the data storage space at the server. The client will then be able to generate m-2 random NIDs within the range that are different from NIDs of both target node and selected empty node.
- *Managing the tree structure integrity:* This aims at maintaining node/parent-node relationships after the node swapping. The authors proposed two solutions to this issue. Shortly, the first solution is to find the empty node to be swapped with the child node and update the parent node accordingly before actually swapping the child node. The second solution is to let clients keep track of all nodes from the root down, deferring all the swaps until the node containing the data is accessed. Each solution has its pros and cons but we are not going to detail them here due to the space limitation. More details of these two solutions can be found in the original paper and will also be clear in the rest of this paper.
- *Concurrency control in the multi-user environment:* The authors also presented a simple solution to concurrency control without deadlocks. Basically, their solution is to organize nodes in the data storage space at the server into d levels. Each level requires an empty node list to store empty nodes at this level. Besides, the client always asks for *exclusive locks* (because there is no pure read operations, but all read operations are followed by write ones) of parent-level nodes before that of child-level ones, and always asks for locks of nodes in the same level using some predefined

[3] We should note that Lin and Candan did not describe how to update these empty node linked lists as the swapping operations are carried out, although this is a vitally important factor for the storage space utilization, the performance and concurrency degree of the whole system. We will extend their work and address this issue as described later.

order. Therefore, all clients access nodes in some fixed predetermined order, ensuring deadlock-free accesses in a multi-user environment.

Searching on the Tree: As pointed out in the paper, the first solution to the tree structure integrity maintenance is better than the second one in terms of the space complexity, O(m) vs. O(d x m), and the concurrency perspective. Both of them have the time complexity of O(d x m), where d denotes the depth of the tree storage space and m denotes the redundancy set size. We summarize below the pseudo code of an algorithm that can be used for oblivious traversal (search operations) of outsourced search trees based on the first solution to the tree structure integrity maintenance as mentioned above (this algorithm was given in [28]):

Algorithm 1: Oblivious Traversal of Outsourced Search Trees (by Lin and Candan)

1. Lock and fetch the SNODE, 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 the CURRENT contains the needed data, return CURRENT. Otherwise:
6. Let the CURRENT be PARENT, find the child node to be traversed next, let it be CURRENT, and repeat steps 2 through 5.

Note that, according to the above algorithm, at the same time, a client has to keep two redundancy sets of both PARENT and CURRENT after step 2. The exclusive locks on nodes of the redundancy set of the PARENT can only be released after the locks on nodes of the redundancy set of the CURRENT are gained and necessary information of the PARENT is updated (e.g., the pointer to the EMPTY).

Issues and Limitations of Lin and Candan's Approach: Although experimental results reported by the authors are sound, especially with helps of special cheap encryption/decryption hardware, their work has critical issues and weaknesses. We identify and discuss these issues and weaknesses below and present our solutions to solve all of them in section 4:

- *Maintenance of empty node lists and inefficiency in storage space usage:* As presented, after each node swapping operation, the target node is swapped with an empty node. Nevertheless, the authors did not show how the empty node list is updated or how such target nodes can be reused. From the presented work we can only see that all of those target nodes, whose contents are no longer important to the outsourced database after being swapped, will be abandoned. This is a vitally important issue because pointers EHEADS and ETAILS need to be updated accordingly whenever each swap happens to reflect the change and to ensure the correctness for the next node swapping operation in the same tree level. Obviously, failing to do so will lead to totally *destroy* the correctness of the whole system (e.g. wrong results will be returned to the client).
- *Insert and delete operations on outsourced search trees:* In dynamic databases, data items can be inserted into or removed from the database. In case such databases are outsourced, the system has to facilitate these basic operations. The work presented by Lin and Candan fails to support such basic operations. However, insert and delete are not simple operations in search trees [12], and the previous work cannot be applied to these operations directly. In these trees, there is a parameter, called *minimum fill factor* k, for tree nodes that is used as the *lower bound* to ensure the storage utilization in the tree. If an item is deleted from a node and makes this node *underfull*, i.e. the current item number in that node is less than kM (where M denotes the maximum number of data items/objects that this node can keep), the node then needs to be deleted or its remaining items need to be redistributed among the

siblings. More seriously, deleting this node may lead to delete its parent node as well and this phenomenon can propagate up to the root. Besides, as a new item is inserted into a node and makes this node *overfull*, the node needs to be split. Similarly, the node splits can propagate up to the root of the search tree (in this case, the tree height will be increased by one). We will present new algorithms based on the access redundancy and node swapping techniques to deal with insert and delete operations on outsourced search trees.

- *Maintenance of the tree structure integrity:* It is easy to observe that the first solution to the tree structure integrity maintenance can not be applicable to insert and delete operations in dynamic outsourced databases due to the split and deletion of nodes as mentioned above. More concretely, we need to deal with the split or deletion propagation up to the parent nodes. We will present new solutions to addressing these issues. Specially, for the insertions, our solution in most cases does not have to lock all nodes along the path up to the root node, but only part of its.
- *Using more empty nodes for a redundancy set:* The node swapping technique that uses only one empty node for each redundancy set is not suitable for cases in which an overfull node is split into two new nodes.
- *Modification of indexed data:* Whenever attributes or feature values of a data object are modified, the object location in the search tree may be changed as well. The modified object must therefore be deleted and reinserted into a correct position in the tree. This basic operation relates closely to inserts and deletes, which are not supported by the previous work as well.

4. Oblivious Search and Updates on Outsourced Search Trees

In this section, we will present new/modified techniques/algorithms to ensure privacy for basic operations on outsourced search trees. First, in section 4.1, we introduce a full-fledged solution to maintaining empty node lists and managing the target nodes that have been swapped with empty ones during the execution of the operations, which has not been addressed by the previous work.

4.1. Node Swapping and Managing Empty Nodes

Our main aim is twofold: (1) to provide a solution to keep information about empty nodes up-to-date after a node swapping operation is carried out, and (2) to provide an efficient use for the database storage space at the server. For the ODBS model, the latter is not less important because it relates directly to costs that the data owner has to pay for the service provider.

In the same line as [28], we also use hidden linked lists to keep empty nodes and information about these lists (i.e., EHEADS and ETAILS) is stored in the SNODE. The difference is that when the target node is swapped with the empty node (all nodes are in the same redundancy set – cf. section 3), information in the SNODE must be updated accordingly to reflect the changes. Concretely, as a client receives a redundancy set of m nodes, including an empty node ENODE, for a certain target node TNODE, it will then have to perform the following tasks:

1. Decrypts TNODE.
2. Manipulates its data as needed.
3. Swaps TNODE with ENODE.
4. Inserts ENODE (this is the *new* empty node with the NID is the old target node's NID) into the end of the corresponding empty node list and amends the head pointer of the list so that it points to the next member of the list, i.e. corresponding information in the SNODE must be updated.
5. Re-encrypts the redundancy set and write it back to the server.

As we can see above, there is a new task the client must tackle: As TNODE is swapped with ENODE, the new ENODE, i.e. the old TNODE, becomes useless for the current tree structure because the new TNODE, i.e. the old ENODE, now takes its role, and thus the new ENODE is inserted into the end of the empty node list (so the ETAILS must be updated) to reuse later. Moreover, information about the first member in the corresponding empty node list, which is the old ENODE itself, must be changed as well (so the EHEADS must be updated). With these changes, our main aim as stated above is satisfied, hence the corresponding issue in the previous work is solved.

There is another issue in this approach: Because the (new) ENODE is inserted into the end of the *corresponding* empty node list at a predefined level (nodes are organized into d levels - cf. section 3), it may provide a clue for some statistical attacks afterwards. To deal with this issue, we propose not to insert the new ENODE into the same empty node list from which the old ENODE was picked up, but choose another list to put it in. This can be easily done because information about all empty node lists is stored in the SNODE (by the pointers EHEADS and ETAILS).

4.2. Search Operation

Basically, besides the changes to the node swapping technique and the maintenance of empty nodes as noted above, the oblivious search algorithm remains quite similar to Algorithm 1. We present our algorithm for this operation as follows:

Algorithm 2: Oblivious Search on Outsourced Search Trees

1. Lock and fetch the SNODE, let it be PARENT. Find the root and let it be CURRENT.
2. If PARENT <> SNODE, lock and fetch the SNODE (will not be performed for the first loop).
3. Select a redundancy set for the CURRENT, lock nodes in the set, and let the empty node in the set be EMPTY.
4. Update information in the SNODE to reflect changes to the empty node lists as described above in section 4.1 (here, the CURRENT and EMPTY take the role of the TNODE and ENODE, respectively).
5. If PARENT <> SNODE, write the SNODE back to the server and release the lock on it.
6. Update the PARENT's pointer to refer to the EMPTY (to maintain the tree structure integrity), re-encrypt nodes of the PARENT redundancy set, write them back to the server, and release locks on the PARENT level.
7. Swap the CURRENT with the EMPTY.
8. If the CURRENT contains the needed data, manipulate that data as required (get it in this case), then re-encrypt nodes of the CURRENT redundancy set, write them back to the server, and release locks on the CURRENT level. Otherwise:
9. Let the CURRENT be PARENT, find the child node to be traversed next, let it be CURRENT, and repeat steps 2 through 8.

For the sake of clarification, we just present the pseudo-code of the algorithm, omitting details of the implementation stage. In general, whenever a client asks for a data object (to get its details), the outsourced search tree must be traversed from the root downwards to the leaf node in which that object should reside.

In step 1, our algorithm locks and fetches the special public entry node to the outsourced database, SNODE, and names it PARENT. As we described in section 3, the SNODE keeps the root node position of the outsourced search tree in the pointer ROOTS and the algorithm names it CURRENT. To manage empty nodes as presented before the algorithm has to lock and fetch the SNODE if it has not been done (step 2). Actually, this step will not be performed in the first loop because the SNODE itself is

also the PARENT. However, from later loops (step 9), step 2 is very important for the purpose of managing empty nodes as shown in section 4.1. Furthermore, step 3 can only be done after step 2, i.e. after information of the EHEADS and ETAILS is safely obtained. When we have already locked and fetched nodes in the redundancy set of the CURRENT (step 3), the algorithm has to update information related to the empty node in the SNODE as discussed in the previous section (step 4). At this time, the EHEADS and ETAILS must be updated to reflect the fact that the EMPTY no longer belongs to the empty node lists, and the CURRENT becomes a new member of one of those lists. The SNODE is then encrypted (using a fixed key known to all clients), written back to the server and the exclusive lock on it is released (step 5). Of course, step 5 will not be performed for the first loop because the algorithm still needs the PARENT (also the SNODE during the first loop) for step 6. In step 6, the PARENT's pointer referring to the CURRENT will be changed to refer to the EMPTY, which will take the role of the CURRENT after the swap is carried out in step 7. For the first loop, this PARENT's pointer is also one of the pointers ROOTS in the SNODE. As the PARENT's pointer has already changed in step 6, the algorithm will have to re-encrypt and write back all nodes of the PARENT redundancy set to the server, then release locks on these nodes at the end of step 6. After being swapped, the CURRENT is checked to see if the needed data object is residing in there to further process as required by the client and, similarly to step 6, all nodes of the CURRENT redundancy set will be re-encrypted and written back to the server (step 8). Otherwise, a new loop is triggered, repeating steps 2 through 8, with new PARENT and CURRENT nodes are assigned as shown in step 9.

We should note that the number of child nodes to be traversed next in step 9 may be more than one in some situations:

- The client asks for more than one data object.
- The (spatial) data object asked by the client overlaps more than one node along the search path.
- Nodes in a search tree can overlap (see [12, 19, 4, 11]) and the asked object overlaps the intersection of some nodes.
- Some outsourced search trees may store a data object over several leaf nodes, for example R+-trees [32], UB-trees [3].

In such situations, the algorithm must keep all necessary information of the child nodes to be traversed next in order to pass it on to step 9 repeatedly. In other words, several loops from step 2 to 8 must be performed to complete the search on these child nodes.

Our algorithm as presented above also follows the first solution to the tree structure integrity maintenance (cf. section 3). Let d denote the depth of the tree storage space and m denote the redundancy set size, obviously the space complexity of the algorithm is O(m) and the time complexity is O(d x m).

4.3. Insert Operation

As a new data object needs to be inserted into a search tree, we first have to look for a leaf node that is *most suitable* to keep the new object. It is out of the scope of this paper to discuss which leaf node can be considered most suitable, and we refer interested readers to [19, 12] for detailed discussions. After a leaf node is determined and if it still has at least one free entry, the new object will be inserted into that leaf. Otherwise, i.e. the leaf is already a full node, it needs to be split into two leaf nodes, and the new object will be inserted into one of them. In this case, corresponding updates must be performed on the parent node to reflect the split, and such updates may propagate upwards to the higher level of the tree if the parent node is also already full. In the worst case, the splits

propagate up to the root node, and if the root is also full, it must be split and a new root is created, so the tree height increases[4].

The scenario is similar in the case of outsourced search trees. However, with the node swapping technique, we have to lock nodes along the traversal path on the tree. If the nodes are not locked, later necessary updates on these nodes are impossible because their NIDs can be changed by other oblivious operations. Furthermore, as mentioned before, because there are not pure read operations existing with the node swapping technique so we have only exclusive locks on tree nodes. Obviously, exclusively locking all nodes along the traversal/search path from the root will penalise the concurrency degree of the outsourced database. In this section, we will present a new locking scheme for outsourced search trees, which can be used for *oblivious insert* operations (with a modified node swapping technique) but does not have to lock *all* nodes along the traversal path. In our locking scheme, only a part of nodes on that path, from the root to the chosen leaf, needs to be locked.

The philosophy of our locking scheme is as follows: Lock and fetch nodes in the root redundancy set in exclusive mode and then find the appropriate child node of the root to access next. Lock and fetch nodes in the redundancy set of this child node and check if the child node is not full. If it is, then the locks on nodes of the root node redundancy set can be released. Otherwise, continue traversing down the tree and whenever a non-full node on the traversal path (would be the chosen leaf itself) is found, the locks on all nodes in *all* previous redundancy sets can be released. The reason behind this locking scheme is that if a node on the traversal path is not full, the insertion of a new data object into a leaf node in its sub-tree will not cause changes to higher level index nodes[5], even in case there are splits in nodes on the traversal path in this sub-tree. In database textbooks, this approach is called the *conservative approach* for concurrency control in indexes. Figure 2 illustrates this idea more clearly.

In Figure 2 we have three examples with an assumption that the traversal path contains nodes from 0 to 4, where node 0 is the root, node 4 is the chosen leaf to keep the new data object, and nodes 1, 2, and 3 are internal nodes. Figure 2(a) shows a case when the selected leaf node 4 is not full. In this case, as soon as we can verify that node 4 is a non-full leaf, we can release locks on all nodes in the redundancy sets of nodes 2 and 3. Note that, in this example, locks on all nodes in the redundancy sets of nodes 0 and 1 were already released as soon as we verified that the internal node 2 was not full. In Figure 2(b), only nodes in the redundancy sets of nodes 0 and 1 are released because all nodes on the traversal path below node 2 are full. In this case, all nodes in the redundancy sets of nodes 2, 3, and 4 are locked exclusively. Figure 2(c) shows a case when all nodes along the traversal path below the root are full, and thus every node in the redundancy sets of all nodes belonging to the path must be locked. As discussed in [12], however, this is not a frequent circumstance because index structures usually do not gain the maximum storage utilization of 100%, hence we can usually find at least a free entry in some nodes along the traversal path.

[4] Note that this discussion does not apply to some special MAMs like SH-trees [14], well-known k-d trees and its variants. Such tree-based indexes are typically *unbalanced* trees, and thus the updates due to the spits usually do not propagate up to the root (for example, in the SH-tree, only the balanced node above the split leaf node needs to be updated and split)

[5] This is usually not true for MAMs: whenever a new spatial object is inserted into a MAM, all nodes along the search/traversal path need to be updated to reflect the fact that the new object is resident in their sub-partitions. Fortunately, such updates can be done as traversing down the MAM, but do not have to postpone until the new object is actually inserted into the selected leaf node. We will address this issue as well in our algorithm introduced later. Again, see [12] for more details of MAMs and other related references.

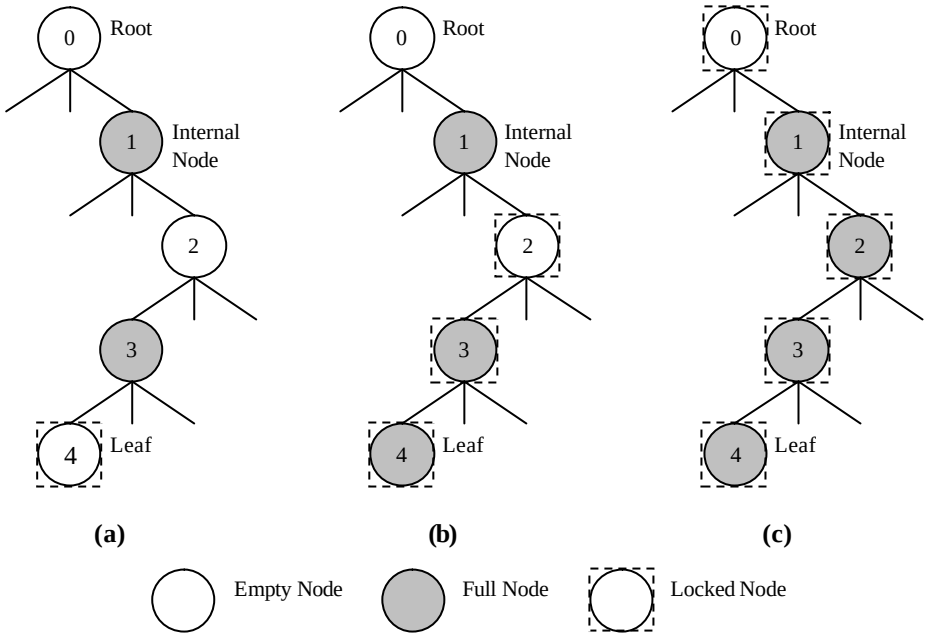


Figure 2: Examples for the *conservative* locking scheme on outsourced search trees

We present below the high level pseudo-code of an algorithm for the oblivious insert operations, which can be applied to a wide range of outsourced search trees. With some special search trees, such as SH-trees [14], UB-trees [3], R+-trees [32], the algorithm can be easily adapted, but we will not go further into such details because of the space limitation.

Algorithm 3: Oblivious Insert Operations

1. Find the root, let it be CURRENT.
2. Initialize a stack variable called STACK.
3. Get a redundancy set for the CURRENT so that this set contains the target node and:
   - at least *three* empty nodes if the CURRENT itself is the root
   - at least *two* empty nodes, otherwise.
4. If the CURRENT is not full, empty the STACK.
5. If the CURRENT is an internal node:
   - 5.1. Update some meta-information in the CURRENT if necessary.
   - 5.2. Push its redundancy set in the STACK.
   - 5.3. Find its child node to be traversed next, let it be CURRENT and go back to step 3.
6. Else (i.e. the CURRENT is a leaf node):
   - 6.1. If it is not full (so the STACK is empty now because of step 4): insert the new object into it, swap it with one of the empty nodes in the redundancy set, and finalize the algorithm.
   - 6.2. Else (i.e. this leaf node is full, and thus either the leaf itself is the root node or the STACK is not empty), do the following tasks:
     - 6.2.1. Insert (M+1) data objects, M in the CURRENT and a new one, into the two empty nodes (E1, E2) in the redundancy set of the CURRENT. Note that M denotes the maximum number of objects that a leaf node can keep.
     - 6.2.2. If the STACK is empty (i.e. the CURRENT itself is the root node): empty out the CURRENT, insert E1, E2 into the CURRENT, swap the CURRENT with the third empty node (E3) in its redundancy set, and finalize the algorithm. Otherwise:

6.2.3. Pop an item (a redundancy set in this case) out from the STACK. Let the target node (an internal node) in this redundancy set be PARENT.

6.2.4. If the PARENT is not full (so the STACK is empty now because of step 4): delete the CURRENT from the PARENT, insert E1 and E2 into the PARENT, swap the PARENT with one of the empty nodes in its redundancy set, and finalize the algorithm.

6.2.5. Else (i.e. the PARENT is full, and thus either the PARENT itself is the root node or the STACK is still not empty):

6.2.5.1. Delete the CURRENT from the PARENT (so the PARENT now consists of M'-1 entries, where M' is the maximum number of entries that an internal node can keep).

6.2.5.2. Insert (M'+1) entries, M'-1 in the PARENT and E1, E2 from the CURRENT redundancy set, into the two empty nodes (E1', E2') in the redundancy set of the PARENT.

6.2.5.3. Let the CURRENT be PARENT, let E1 and E2 be E1' and E2', respectively, then go back to step 6.2.2

Although the above presented algorithm is not difficult to follow, we have some important notes as follows. First, for each redundancy set, we need to lock and fetch at least two (or three if the target node itself is the root) empty nodes in order to support the possible node splits afterwards (see step 3). If no split is required for a certain node (see steps 6.1 and 6.2.4), one of these empty nodes is employed to swap with the target node as usual as described in the node swapping technique (cf. section 4.1). On the contrary, the target node is split and its entries as well as the new objects (see steps 6.2.1 and 6.2.5.2) are inserted into these two empty nodes. With these changes, all desired security objectives of the node swapping technique are still preserved, while we can apply this modified technique to the oblivious insert operation as presented above. Note that, in a special case where the root is full and needs to be split: After its entries and the new objects are distributed over the first two empty nodes, it is emptied out and these two empty nodes are inserted back into the root, then the root itself is swapped with the third empty node before the algorithm is finalized (see step 6.2.2).

Second, as mentioned in step 5.1, because the algorithm does not keep the full search path (from the root to the leaf) so, for several MAMs, some meta-information stored in the internal/root nodes, which are no longer kept track of once the nodes are removed from the STACK, needs to be updated soon before the new item is actually inserted. For example, information about the boundary of the subspace, that is stored in an internal node, as the new object is inserted into some leaf node needs to be updated as soon as this internal node is visited as shown in step 5.1. Some MAMs carry out this kind of update automatically whenever a node is visited during the execution of the insertion method (e.g., SH-trees).

Last, as mentioned before, the main advantage of our approach is that it is unnecessary to lock all nodes visited along the search/traversal path as the second solution to the tree structure integrity maintenance in [28] (cf. section 3), hence the higher concurrency degree can be achieved whistle the tree structure integrity is still ensured. This advantage relies on the fact that we are able to find some non-full nodes along the traversal path from the root down the tree to the target leaf node. In most practical cases, as discussed, this fact is affirmative, and thus our conservative approach introduced in this section is quite practical.

4.4. Delete Operation

When a data object needs to be removed from a search tree, the leaf node possibly consisting of this object first has to be located. If the object is found there, it will be

eliminated and necessary updates on parent nodes at higher levels need to be conducted in order to maintain the tree structure integrity and keep the search keys as specific as possible. In some cases, as shown below, the target leaf node and some internal nodes may have to be removed and their entries are reinserted after that. In general, deletions are usually very complicated and costly operations on search trees. On outsourced search trees, where the node swapping and access redundancy techniques are employed, only exclusive locks are used for nodes being accessed. This makes the problem more difficult and expensive.

With a minimum fill factor k chosen for a particular search tree, when an object is deleted from a leaf, more issues are arisen if that leaf becomes under-full (i.e., its remaining item number is less than kM). There are several solutions to this problem [22]: An under-full leaf node can be merged with whichever sibling that needs to have the least enlargement or its objects can be scattered among sibling nodes. Both of them can cause the node splits, especially the latter can lead into a propagated splitting. Therefore, R/R*-trees [22, 5] and many new MAMs, such as SR-trees [27], SS-trees [35], Hybrid-trees [9], etc., employ re-insertion policy instead of the two ones as mentioned above. With the re-insertion policy, all objects in an under-full leaf are reinserted after this leaf is removed from the tree structure (and added to an empty node list in the case of our outsourced search trees). Note that the same issue can happen to internal nodes if one of its entries is eliminated and the same technique can be employed to solve this issue.

Some other modern MAMs employ a "delete-borrow-reinsert" policy: after an object is deleted from a node and if this node becomes under-full, it tries to "borrow" some objects from its siblings and if this is impossible (e.g., each sibling just has an enough number of entries, i.e. kM), then the re-insertion policy is applied. As shown in [12], this "delete-borrow-reinsert" policy is desired for the storage utilization but not usually good for the search performance. Again, as we discussed before in section 4.3, this will lead to decrease the concurrency degree for the insertions of outsourced search trees. So we will follow the re-insertion policy to deal with under-full nodes in R-trees and the GiST [25] in order to control under-full nodes in outsourced search trees. We present below a pseudo-code algorithm for oblivious delete operations on outsourced search trees based on the node swapping technique:

Algorithm 4: Oblivious Delete Operations

1. Search for the leaf node L containing the deleted object O. Lock all nodes in the redundancy sets along the search path from the root node. Stop if O is not found. Note that, each redundancy set needs to have only one empty node E, not two or three as in Algorithm 3.
2. Remove O from L.
3. Let CURRENT be L. Set Q, the set of deleted nodes, to be empty.
4. If the CURRENT is the root, go to step 8. Otherwise, let PARENT be the parent of the CURRENT, and let E_N be the CURRENT's entry in the PARENT.
5. If the CURRENT has fewer than kM entries (i.e. it is an under-full node):
   5.1. Delete E_N from the PARENT and add the CURRENT to set Q.
   5.2. Add the CURRENT to an empty node list.
6. If E_N has not been removed from PARENT, adjust the meta-information in E_N and at higher levels accordingly to reflect the deletion of O.
7. If E_N has been removed from PARENT: Let the CURRENT be PARENT, and go to step 4.
8. Reinsert all entries of nodes in set Q. Entries from deleted leaf nodes are reinserted into tree leaves as described in Algorithm 3, but entries from higher level nodes must be placed higher in the tree at the corresponding levels.
9. If the root has only one child, make the child the new root.

10. Release the locks on nodes and finalize the algorithm. Note that, swapping the node with the empty node in the corresponding redundancy set may need to be conducted in this step.

Although Algorithm 4 above is not for B+-trees, it can be easily adapted to support the B+-tree and its variants. To do this, we need to change step 5 of the above algorithm to support the "borrow or coalesce" policy of B+-trees on dealing with under-full nodes. Again, to support special trees like SH-trees, R+-trees, UB-trees, and k-d tree family, Algorithm 4 will also have to be modified. Furthermore, as argued in [26], in some implementations of tree-based indexes it is preferable to leave the under-full nodes after a delete in the expectation that it will be filled up soon thereafter. To support such behaviour, we can replace all steps 3 through 10 in Algorithm 4 with only one step to carry out the three following tasks at each *locked* target node along the search path: (1) Adjust the meta-information stored in the node accordingly to reflect the deletion of object O, (2) Apply the node swapping technique as described in section 4.1 to the redundancy set of this node, and (3) Re-encrypt nodes in the redundancy set, write them back to the server, and release the locks on them. Note that, similarly to Algorithm 3, in this case we also do not have to lock all nodes along the search path, but just nodes that their meta-information should be updated if O is deleted from the tree. We present the pseudo-code algorithm for this case as follows:

Algorithm 5: Oblivious Delete Operations (as under-full nodes are accepted in the tree)

1. Search for the leaf node L containing the deleted object O. Lock nodes in the redundancy sets along the search path if the target node in that redundancy set should be updated in the case O is actually removed from the tree. Stop if O is not found. Note that, each redundancy set needs to have only one empty node E, not two or three as in Algorithm 3.
2. Remove O from L.
3. With each locked target node from L's parent node up the tree:
   3.1. Adjust the meta-information stored in the node accordingly to reflect the deletion of object O in step 2.
   3.2. Apply the node swapping technique as described in section 4.1 to the redundancy set of this node.
   3.3. Re-encrypt nodes in the redundancy set of this node, write them back to the server, and release the locks on them.

As concluded in the original paper, a critical weakness of the node swapping technique is that if a query Q occurs at a very high frequency, the intersections introduced by random nodes (in the redundancy sets) cannot hide the intersections between Qs. We observe that the above assumption can be slightly changed, but the consequence: It is easy to see that reinserting a large number of *near* objects consecutively may lead to reveal information about the tree structure. The main reason is that there is a high probability that a part of the traversal paths as inserting two near objects is the same because near objects are usually stored in the *same or near* nodes (e.g., nodes having the same parent) [19]. Therefore, if a large number of near objects are inserted consecutively, part of the tree structure may be disclosed with a high probability. Attackers may make use of this aspect to find out the whole tree structure. In [28], the authors mentioned that dummy calls and intersections can be introduced to overcome this issue. Although the authors have not realized this idea, it can be foreseen that this solution would be prohibitively expensive. Nevertheless, it is easy to see that our Algorithm 5 above can be employed to overcome this critical weakness of the node swapping technique because the insertions are completely omitted.

4.5. Modification Operation

If an indexed data object is modified so that its feature values (or attributes) related to its location in the search tree are changed, the object must be deleted, updated, and then reinserted, so that it will find its way to the right place in the tree. For example, when the shape of an object in the R-tree is changed, its covering rectangle will be changed as well, and this leads to the possible movement of its location in the tree. This basic operation relates closely to inserts and deletes, and in the context of outsourced search trees it can be done easily with the support of oblivious insert and delete operations as introduced in previous sections.

5. Other Related Work

Protecting data and user privacy in the digital era is not trivial work and a number of research activities have been carried out. We give a brief summary of some related research other than the work presented in [28]:

Most similarly to our work, there are two approaches aiming at protecting the data confidentiality for outsourced indexed data [16, 23]. Both approaches protect the outsourced data from intruders and the server's operators by some encryption method. To process queries over encrypted data, they introduced two different solutions. In [23], the authors proposed storing, together with the encrypted data, additional indexing information. This information can be used by the untrusted server to select the data in response to a user's query. The main idea to process a query in this scheme is to split the original query into: (1) A corresponding query over encrypted relations to run on the untrusted server; and (2) a client query for post processing the results returned from the server query. The major challenge in this scenario is how to compute and represent index information. Specially, the relationship between indexes and data should not open the door to inference and linking attacks that can compromise the protection granted by encryption. However, as stated in [16, 18], although the *index of range* technique proposed in [23] which relies on partitioning domains of client tables' attributes into sets of intervals is suitable for both exact match and range queries, it introduces difficulties in managing the correspondence between intervals and the actual values present in the database as well as limitations in such protection. In the same line as [23], the authors of [16] also introduced a method to query a tuple-level encrypted database but with a better security level for the outsourced data. For exact match queries, they analyzed some potential inference and linking attacks, and proposed a hash-based indexing method. In order to execute interval-based (range) queries in the ODBS model, they proposed a solution employing B+-trees. However, as shown in [13], their solution does not provide an oblivious way to traverse the B+-tree and this can be exploited by the untrusted server to carry out inference and linking attacks. Besides, none of the two above approaches supports outsourced MAMs.

In a quite different scenario, when the data owners outsource their private data and allow other clients to access the outsourced data, the *data privacy* may also become important [20, 15, 13]. By the data privacy, the client is not allowed to receive data that does not belong to the query result. In this scenario, the data owner needs to protect their data from both clients and the server. Clients, in turn, may not want to reveal queries and results to both the data owner and the server. Therefore, additional data and user privacy-preserving issues must be taken into account. In [20], the authors introduced an approach to the data privacy in PIR schemes but this approach is not for the outsourced data. Du and Atallah [15] introduced a solution to this outsourcing model, but their approach cannot be applied to outsourced search trees. In our recent work [13], we proposed a solution resorting to a *trusted* third party in order to bring this outsourcing model back to the ODBS model considered in this paper.

In [8], Chor et al. first time introduced the PIR protocol and it has been investigated by many researchers thereafter [2]. In general, the PIR protocol supports the user privacy, allowing a client to access a database without revealing to the server both the query and the result. In [13], we also introduced a solution to the ODBS model based on PIR-like protocols. Although this approach will become prohibitively expensive if there is no replication for the outsourced data and an information-theoretic PIR protocol is employed, it can become practical in case some efficient computational PIR protocol, such as [1], is used instead.

Last, apart from the software-based approaches as shown above, hardware-based approaches to the problem of secure computations have also been investigated and developed at IBM [30, 31]. In their project, IBM has been developing special security hardware equipment called *secure coprocessors* that can support secure computations at both client and server sides. Although this hardware-based solution may satisfy security objectives, there is still a matter of argument [21, 30].

6. Conclusions and Future Work

The outsourced database service (ODBS) model has been emerging as an efficient replacement solution for traditional in-house DBMSs. In this model, however, the private data is stored at an external provider, who is typically not fully trusted. Therefore, dealing with security issues in the ODBS model has rapidly become one of the most active topics in the research community. In this paper, we presented full-fledged solutions to the problem of preserving privacy for basic operations on outsourced search trees, which take a fundamental and important role in both traditional and modern database application domains.

The basic operations of search trees include search (to answer different query types) and updates (insert, delete, and modification). Based on the access redundancy and node swapping techniques introduced in [28], we proposed practical algorithms for privacy-preserving search/traversal, insert, delete, and modification operations that can be applied to a variety of outsourced search trees. Moreover, we also proposed a complete solution to maintaining empty node lists and managing the target nodes swapped with the empty ones during the execution of the oblivious operations, which was not clarified in [28]. To the best of our knowledge, none of the previous work has dealt with the oblivious updates on outsourced search trees. Our presented work therefore provides the vanguard solutions for this problem.

Our future work will focus on evaluating the efficiency of the proposed solutions in real-world applications (e.g. GISs, TISs, multimedia databases, etc.). Specially, evaluating the efficiency of Algorithms 5 and investigating its impact on various outsourced search trees will be of great interest because, as introduced in [26], the technique used in this algorithm has been designed to work only with B-trees. Furthermore, comparing our approach in this paper with the one introduced in [13] using some efficient computational PIR protocol as mentioned in section 5 will be interesting. Last but not least, because there are various search trees existing, so developing generalized oblivious tree algorithms will also be worth considering.

References

[1] D. Asonov, J.C. Freytag. Repudiative Information Retrieval. Proc. of the ACM Workshop on Privacy in the Electronic Society, USA, 2002

[2] D. Asonov. Private Information Retrieval - An Overview and Current Trends. Proc. of the ECDPvA Workshop, Informatik 2001, Austria, 2001

[3] R. Bayer. The Universal B-Tree for Multidimensional Indexing: General Concepts. Proc. of the International Conference on Worldwide Computing and Its Applications (WWCA'97), Japan, 1997

[4] C. Boehm , S. Berchtold , D.A. Keim. Searching in High-Dimensional Spaces: Index Structures for Improving the Performance of Multimedia Databases. ACM Computing Surveys (CSUR), 33(3), September 2001, 322-373
[5] N. Beckmann, H-P. Kriegel, R. Schneider, B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. Proc. of the ACM SIGMOD International Conference on Management of Data, USA, May 23-25, 1990, pp. 322-331
[6] L. Bouganim, P. Pucheral. Chip-Secured Data Access: Confidential Data on Untrusted Servers. Proc. of VLDB 2002
[7] S. Castano, M.G. Fugini, G. Martella, P. Samarati. Database Security. Addison-Wesley and ACM Press 1994, ISBN 0-201-59375-0
[8] B. Chor, O. Goldreich, E. Kushilevitz, M. Sudan. Private Information Retrieval. Proc. of IEEE Symposium on Foundations of Computer Science, 1995
[9] K. Chakrabarti, S. Mehrotra. The Hybrid Tree: An Index Structure for High Dimensional Feature Spaces. Proc. of the 15th International Conference on Data Engineering – ICDE 1999, Sydney, Australia , March 23-26, 1999, pp. 440-447
[10] Y-C. Chang, M. Mitzenmacher. Privacy Preserving Keyword Searches on Remote Encrypted Data. Cryptology ePrint Archive: Report 2004/051
[11] E. Chávez, G. Navarro, R. Baeza-Yates, J.L. Marroquín. Searching in Metric Spaces. ACM Computing Surveys (CSUR), 33(3), September 2001, 273 – 321
[12] T.K. Dang. Semantic Based Similarity Searches in Database Systems (Multidimensional Access Methods, Similarity Search Algorithms). PhD Thesis, FAW-Institute, Johannes Kepler University of Linz, Austria, May 2003
[13] T.K. Dang. Extreme Security Protocols for Outsourcing Database Services. Proc. of the 6th International Conference on Information Integration and Web-based Applications and Services - iiWAS 2004, Jakarta, Indonesia, September 27-29, 2004, pp. 497-506
[14] T.K. Dang, J. Kueng, R. Wagner. The SH-tree: A Super Hybrid Index Structure for Multidimensional Data. Proc. of the 12th International Conference on Database and Expert Systems Applications - DEXA 2001, Munich, Germany, September 3-7, 2001, pp. 340-349
[15] W. Du, M.J. Atallah. Protocols for Secure Remote Database Access with Approximate Matching. Proc. of the 7th ACM Conference on Computer and Communications Security, the First Workshop on Security and Privacy in E-Commerce, Greece, 2000
[16] E. Damiani, S.D.C. Vimercati, S. Jajodia, S. Paraboschi, P. Samarati. Balancing Confidentiality and Efficiency in Untrusted Relational DBMSs. Proc. of the 10th ACM Conference on Computer and Communication Security, USA, 2003
[17] E. Damiani, S.D.C. Vimercati, S. Paraboschi, P. Samarati. Implementation of a Storage Mechanism for Untrusted DBMSs. Proc. of the 2nd International IEEE Security in Storage Workshop, USA, 2003
[18] K.C.K. Fong: "Potential Security Holes in Hacigümüs' Scheme of Executing SQL over Encrypted Data", http://www.cs.siu.edu/~kfong/research/database.pdf (2003)
[19] V. Gaede, O. Guenther. Multidimensional Access Methods. ACM Computing Surveys (CSUR), 30(2), June 1998, 170 – 231
[20] Y. Gertner, Y. Ishai, E. Kushilevitz, T. Malkin. Protecting Data Privacy in Private Information Retrieval Schemes. Proc. of the 30th Annual ACM Symposium on Theory of Computing, 1998
[21] O. Goldreich, R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. Journal of the ACM, 1996
[22] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. Proc. of ACM SIGMOD Conference, Boston, Massachusetts, USA, June 18-21, 1984, pp. 47-57
[23] H. Hacigümüs, B.R. Iyer, C. Li, S. Mehrotra. Executing SQL over Encrypted Data in the Database-Service-Provider Model. Proc. of the ACM SIGMOD International Conference on Management of Data, USA, 2002
[24] H. Hacigümüs, S. Mehrotra, B.R. Iyer. Providing Database as a Service. Proc. of the 18th International Conference on Data Engineering, 2002
[25] J.M. Hellerstein, J.F. Naughton, A. Pfeffer. Generalized Search Trees for Database Systems. TechnicalReport #1274,University of Wisconsin at Madison, July 1995
[26] T. Johnson, D. Shasha. Inserts and Deletes on B-trees: Why Free-At-Empty is Better Than Merge-At-Half. Journal of Computer Sciences and Systems, 47(1), August 1993, 45-76
[27] N. Katayama, S. Satoh. The SR-Tree: An Index Structure for High Dimensional Nearest Neighbor Queries. Proc. of the ACM SIGMOD International Conference on Management of Data, USA, May 13-15, 1997, pp. 369-380

[28] P. Lin, K.S. Candan. Hiding Traversal of Tree Structured Data from Untrusted Data Stores. Proc. of the 2nd International Workshop on Security In Information Systems, WOSIS 2004, Porto, Portugal, April 2004, pp. 314-323
[29] E. Mykletun, M. Narasimha, G. Tsudik. Authentication and Integrity in Outsourced Databases. The 11th Annual Network and Distributed System Security Symposium – NDSS2004, San Diego, California, USA, February 5-6, 2004
[30] S.W. Smith. Secure Coprocessing Applications and Research Issues. Los Alamos Unclassified Release LA-UR-96-2805. Los Alamos National Laboratory, 1996
[31] S.W. Smith, D. Safford. Practical Server Privacy with Secure Coprocessors. IBM Systems Journal, 40(3), 2001
[32] T.K. Sellis, N. Roussopoulos, C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. Proc. of VLDB1987
[33] D.X. Song, D. Wagner, A. Perrig. Practical Techniques for Searches on Encrypted Data. Proc. of IEEE Symposium on Security and Privacy, 2000
[34] A. Umar. Information Security and Auditing in the Digital Age - A Managerial and Practical Perspective. NGE Solutions, December 2003 (e-book version)
[35] D. A. White, R. Jain. Similarity Indexing with the SS-Tree. Proc. of the 20th International Conference on Data Engineering–ICDE1996, USA, February 26-March 1, 1996, pp. 516-523