The VLDB Journal
DOI 10.1007/s00778-008-0113-2

REGULAR PAPER

Authenticated indexing for outsourced spatial databases

Yin Yang · Stavros Papadopoulos · Dimitris Papadias · George Kollios

Received: 1 February 2008 / Revised: 12 May 2008 / Accepted: 23 July 2008

Abstract In spatial database outsourcing, a data owner delegates its data management tasks to a location-based service (LBS), which indexes the data with an authenticated data structure (ADS). The LBS receives queries (ranges, nearest neighbors) originating from several clients/subscribers. Each query initiates the computation of a *verification object* (*VO*) based on the ADS. The *VO* is returned to the client that can verify the result correctness using the public key of the owner. Our first contribution is the MR-tree, a space-efficient ADS that supports fast query processing and verification. Our second contribution is the MR*-tree, a modified version of the MR-tree, which significantly reduces the *VO* size through a novel embedding technique. Finally, whereas most ADSs must be constructed and maintained by the owner, we outsource the MR- and MR*-tree construction and maintenance to the LBS, thus relieving the owner from this computationally intensive task.

Keywords Authenticated index · Database outsourcing · Spatial database · Mobile computing

Y. Yang · S. Papadopoulos · D. Papadias (✉)
Hong Kong University of Science and Technology,
Kowloon, Hong Kong
e-mail: dimitris@cse.ust.hk

Y. Yang
e-mail: yini@cse.ust.hk

S. Papadopoulos
e-mail: stavros@cse.ust.hk

G. Kollios
Boston University, Boston, MA, USA
e-mail: gkollios@cs.bu.edu

1 Introduction

Spatial database outsourcing is motivated by the large availability of spatial data from various sources (e.g., satellite imagery, land surveys, environmental monitoring, and traffic control). Often, agencies collecting such data (e.g., government departments, non-profit organizations) are not able to support advanced query services; outsourcing to a location-based service (LBS) is the only option for utilizing the data. Furthermore, even if an agency possesses the necessary functionality, it may be beneficial in terms of cost, visibility, ease of access etc., to replicate the data in a specialized LBS. Finally, the value of an outsourced dataset may increase if it is combined with the functionality (e.g., driving directions, aerial photos, etc.) of general-purpose online maps.

Figure 1 illustrates a common framework for database outsourcing adopted from the relational literature. The data owner (DO) obtains, through a key distribution center, a *private* and a *public* key. In addition to the initial data, the owner transmits to the LBS a set of signatures required for authentication. Whenever updates occur, the relevant data and signatures are also forwarded to the LBS. The LBS receives and processes spatial queries, (e.g., ranges, k-nearest-neighbors) from clients. The three parties have different computational power: a typical DO possesses a few workstations; the LBS runs a server farm; a client is usually a mobile device (e.g., PDA) on battery power. Therefore, the LBS should perform most of the computation in order to minimize the workload of the DO and, especially, the clients.

Since the LBS is not the real owner of the data, the client must be able to establish the *soundness* and *completeness* of the results. Soundness means that every record in the result set is present in the owner's database and is not modified. Completeness means that no valid result is missing. In order to process authenticated queries efficiently, the LBS indexes

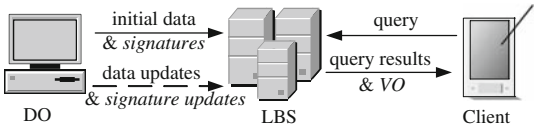


Fig. 1 Database outsourcing framework

the data with an authenticated data structure (ADS). Each incoming query initiates the computation of a *verification object* (*VO*) using the ADS. The *VO* (which includes the query result) is returned to the client that can verify soundness and completeness using the public key of the DO.

Ideally, an ADS must consume little space, support efficient query processing, and lead to small *VO*s that can be easily transferred and verified. In addition, it must be able to handle updates. Most disk-based ADSs focus on 1D ranges. The only work dealing with multi-dimensional ranges is [7], which applies the *signature chain* concept [28] to KD-trees and R-trees. The R-tree based ADS, called VR-tree, is the best between the two options, and is used as the benchmark for the proposed techniques.

Our first contribution is the MR-tree, an index based on the R*-tree, capable of authenticating arbitrary spatial queries. We show, analytically and experimentally, that the MR-tree is considerably faster to build and consumes less space than the VR-tree. At the same time, it is much more efficient for query processing and verification. Furthermore, it supports updates (whereas update algorithms have not been proposed for the VR-tree).

The most important metric in most applications is the *VO* size because it determines the amount of network transmission from the LBS to the client, which is usually the bottleneck of the entire system. This is especially true for mobile clients (e.g., PDAs and smart phones) where battery consumption is a major concern, since wireless transmissions consume significantly more power than offline computations [11]. Our second contribution, the MR*-tree, employs a novel embedding technique to considerably reduce the *VO* size compared to the MR-tree, while imposing marginally more CPU overhead for the LBS and the client.

In the majority of outsourcing systems, the initial construction as well as the updates of the ADSs are not outsourced, but performed locally by the owner and transmitted to the service provider. Consequently, the DO must acquire the software for maintaining the structures, and dedicate the hardware to perform the computations. In the case of spatial outsourcing, the problem is magnified because data indexing and updating require specialized (i.e., non-relational) schemes and incur high cost. Our third contribution solves this problem through a set of secure and efficient protocols for fully outsourcing index maintenance to the LBS. Specifically, the expensive initial construction and update operations are performed entirely by the LBS. The DO only needs to authenticate the index through cheap verification processes. Furthermore, it does not need to store the tree locally.

The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 describes the basic MR-tree structure, discusses query processing, and offers cost models for its performance. Section 4 proposes the MR*-tree and related algorithms. Section 5 focuses on protocols for outsourcing the construction and maintenance of the MR- and MR*-tree. Section 6 contains a comprehensive experimental evaluation, and Sect. 7 concludes the paper.

2 Background

Section 2.1 overviews existing solutions for authenticating one-dimensional range queries. Section 2.2 discusses multi-dimensional query authentication, focusing on the VR-tree. Section 2.3 surveys alternative models for database outsourcing.

2.1 1D range query authentication

The *Merkle Hash Tree* (*MH-tree*) [22] is a main-memory ADS that has influenced several authenticated processing techniques. It is a binary tree that hierarchically organizes hash[1] values (or *digests*). Figure 2 illustrates a MH-tree covering eight data records $d_1 - d_8$, each assigned to a leaf. A node N contains a digest H_N computed as follows: if N is a leaf node, $H_N = hash(d_N)$, and d_N is the assigned record of N, e.g., $H_1 = hash(d_1)$; otherwise (N is an internal node), $H_N = hash(H_{N.lc}|H_{N.rc})$, where $N.lc$ ($N.rc$) is the left (right) child of N, respectively, and "|" concatenates two binary strings, e.g., $H_{1-4} = hash(H_{1-2}|H_{3-4})$. After building the tree, the DO signs the digest H_{root} stored in the root of the MH-tree, using a *public key digital signature scheme* (e.g., RSA [25]).

Devanbu et al. [10] authenticate one-dimensional range queries using a MH-tree on the query attribute. Figure 2 shows an example, where the LBS receives query Q covering records d_4 and d_5. The LBS first determines the *boundary records* of Q, i.e., d_3, and d_6 which bound Q's result. Then, it follows the root-to-leaf path ($root$, N_{1-4}, N_{3-4}, N_3) to the left boundary record d_3. For each node visited, the digest (H_{1-2}) of its left sibling is inserted into the *VO*. Records d_3, d_4, d_5, d_6 are added to the *VO*. Similarly, the digests (H_{7-8}) of all right-siblings on the path from the root to the right boundary d_6 are also appended. The LBS sends the *VO* and the signature of H_{root} to the client. To verify the sequence, the client re-constructs the digest at the root of the MH-tree using d_3, d_4, d_5, d_6, and the digests in the *VO*

[1] Throughout the paper, the term *hash function* implies a *one-way*, *collision-resistant* hash function. In this work we employ SHA1 [25].

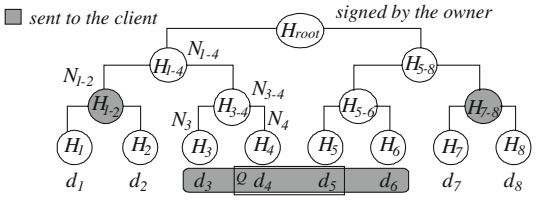


Fig. 2 Example of a MH-tree

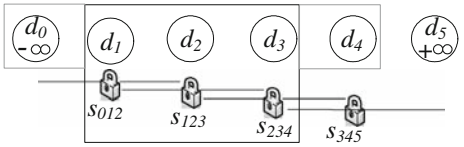


Fig. 3 Example of signature chaining

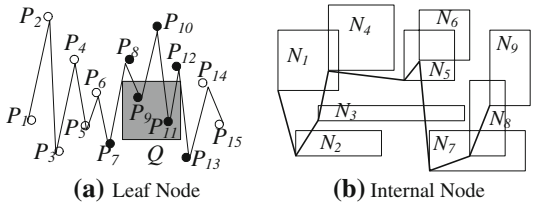


Fig. 4 Signature chains in the VR-tree

$(H_{1-2}, H_{7-8}) : H_{root} = hash(hash\{H_{1-2}|hash[hash(d_3)|(hash(d_4))]\}|hash\{hash[hash(d_5)|(hash(d_6))]|H_{7-8}\})$. If the re-constructed H_{root} matches the owner's signature, the result is sound. The boundary records also guarantee that no records are omitted from the query endpoints (completeness).

The *VB-tree* [31] is a disk-based ADS that establishes the soundness, but not the completeness, of 1D range results. *Signature chaining* [27,28] authenticates both soundness and completeness. Figure 3 illustrates an example, assuming that the database contains four tuples $d_1 - d_4$, sorted on the query attribute. The data owner inserts two special records d_0, d_5 with values $-\infty$ and $+\infty$, and creates four signatures $s_{012}, s_{123}, s_{234}, s_{345}$, one for each triplet of adjacent tuples; s_{012} corresponds to d_1, s_{123} to d_2, and so on. The data and signatures are then transferred to the LBS. Let the result of a range query contain d_1, d_2, and d_3. The LBS inserts into the *VO*: the result (d_1, d_2, d_3), the signature for each tuple in the result $(s_{012}, s_{123}, s_{234})$, and the boundary records d_0 and d_4. Given the *VO*, the client checks that (i) the two boundary records fall outside the query range and (ii) all signatures are valid. The first condition ensures that no results are missing at the range boundaries, i.e., d_1 and d_3 are indeed the first and last records of the result. The second guarantees that all results are correct.

The *Merkle B-tree* (MB-tree) [20] is a disk-based adaptation of the MH-tree. Each internal node stores entries e of the form $(e.p, e.k, e.H)$, where $e.p$ points to a child node N_c, $e.k$ is the search key and $e.H$ is a hash value computed on the concatenation of the digests of the entries in N_c. Leaf nodes store records and their respective digests. The DO signs the hash value of the concatenation of the digests contained in the root of the tree. Compared to signature chaining, the MB-tree incurs less space overhead since digests are smaller than signatures and less verification effort because only the root is signed. Li et al. [20] propose embedded structures inside MB-tree nodes in order to reduce the *VO* size. Adaptations of MH-trees have also been used to handle continuous authenticated processing in streaming environments [21,32]. Finally, Tamassia and Triandopoulos [36] propose a distributed MH-tree for data authentication over peer-to-peer networks.

2.2 Multi-dimensional query authentication

A combination of the MH-tree and the *Range Search tree* [3] is exploited in [10] to authenticate multi-dimensional range queries. Martel et al. [23] extend the MH-tree concept to arbitrary search Directed Acyclic Graphs (DAGs), including dictionaries, tries, and optimized range search trees. Goodrich et al. [14] present ADSs for graph and geometric searching. These techniques, however, focus on main memory and are highly theoretical in nature. For example, the range search tree is rarely used in practice due to its high space requirements: $O(|D| \cdot \log^{d-1} |D|)$, where $|D|$ and d are the size and dimensionality of the data, respectively.

The only multi-dimensional ADSs in the database literature are the VKD-tree and VR-tree [7]. These structures apply the signature chain concept to KD-trees [3] and R-trees [13], respectively. We focus on the VR-tree since, as shown in [7], it outperforms the VKD-tree. All points in a leaf node are sorted according to their x-coordinates. Two fictitious points are added before the first and after the last point of the node. Following [28], the VR-tree creates one signature for each sequence of three points and stores it along with each entry, e.g., in Fig. 4a, the entry for P_8 contains s_{789}. For internal nodes, the minimum bounding rectangles (MBRs) of child nodes are sorted on their left side and a signature chain is formed in a similar way. For instance, in Fig. 4b, the signature of node N_4 is s_{345}.

The processing of range queries is similar to the R-tree, except for the additional *VO* construction. Consider query Q in Fig. 4a, which retrieves P_9 and P_{11}. For each index node visited, *all* MBRs in this node are inserted into the *VO*. The corresponding signatures participate in the incremental construction of an *aggregated*[2] signature s. When a leaf node of the VR-tree is reached, all points whose x-coordinates

[2] *Signature aggregation* [24] condenses multiple signatures into a single one, thus significantly reducing the total size.

fall in the query range ($P_8 - P_{12}$) and the two boundary points (P_7, P_{13}) are inserted into the *VO*. The corresponding signatures are aggregated in s, which is included in the *VO*.

To verify results, the client starts from the root and compares all MBRs against the query. Then, it reads the content of each node whose MBR overlaps the query from the *VO* and recursively checks all its children. Finally, at the leaf level, it can extract the query results. During this procedure, the client incrementally constructs an aggregated digest from the MBRs and points included in the *VO*, which is eventually verified against the aggregated signature. Cheng and Tan [5,6] present an improved version of the VR-tree to authenticate various multi-dimensional query types, including k nearest neighbor and 2D reverse nearest neighbor queries. Furthermore, they apply a collaborative digest computation technique introduced in [28] to avoid disclosing objects that do not belong to the query result (e.g., boundary records).

2.3 Other related work

In certain scenarios, efficient authentication does not require an ADS. Sion [34] assumes a *semi-trusted server* (e.g., LBS in our setting), whose motivation for cheating is to save computation resources. The only client for this server is the DO, which issues a batch of queries together with a *challenge token* that captures authenticated information about the result of a secret query Q from the batch. The SP responds with the result sets of all queries and the id of Q. This method probabilistically establishes that the LBS has indeed performed the necessary computations to correctly answer all queries in the batch. However, a malicious LBS can return incorrect results after performing the proper compuations.

Another method [38] incorporates fake tuples in the dataset. The DO encrypts all records and transmits them to the server. It also provides the clients with the function used to generate the fake tuples. The server cannot distinguish the fake from the real records. The encryption scheme implicitly ensures soundness. The clients can probabilistically verify completeness by checking whether all fake tuples satisfying the query are present in the result set. Note that all clients are considered trusted, because otherwise the server could collude with a client and obtain the fake tuple generator. This scheme cannot be applied to our model, since we do not assume any degree of trust for the LBS and the clients.

Several papers study *privacy preservation* of outsourced data, under a model similar to [34], i.e., the only client is the DO. Hacıgümüş et al. [17,18] assume that the DO transmits encrypted data to the server, along with a set of *crypto-indices*, one for each query attribute. Specifically, for every such attribute, the DO partitions all its values into buckets, and stores in the corresponding crypto-index the bucket ids for each record. To process a range query Q, the DO translates Q into Q' by replacing attribute values with bucket ids; the server answers Q' using the crypto-indices and returns the encrypted tuples. Note that the results of Q' are a superset of that of Q. Decryption and filtering of false hits are performed at the client's site. In the same context, Damiani et al. [12] propose a method where the client executes a sequence of queries that retrieve encrypted index nodes at progressively deeper levels. Agrawal et al. [1] introduce OPES, a scheme where the encrypted data preserve the original order, eliminating the use of additional crypto-indices. Ge and Zdonik [15] study aggregate query processing over encrypted data. De Capitani di Vimercati et al. [8] investigate access control enforcement in outsourced databases. Finally, Wong et al. [37] enable mining of association rules on data encrypted with 1-to-n item mapping transformations. Query authentication methods can be used in combination with the above-mentioned techniques.

3 MR-tree

Section 3.1 presents the structure of the MR-tree, and describes query processing and authentication. Section 3.2 contains cost models for various performance metrics, and compares the MR-tree and the VR-tree analytically.

3.1 Structure and query processing

The MR-tree combines concepts from MB- [20] and R*-trees [4]. Figure 5 illustrates the node structure. Leaf nodes are identical to those of the R*-tree: each entry P_i corresponds to a data object. Note that although our examples use points, the MR-tree is applicable to objects with arbitrary shapes. We hereafter use terms "object" and "point" interchangeably. A digest is computed on the concatenation of the binary representation of all objects in the node. Internal nodes contain entries of the form (p_i, MBR_i, H_i), signifying the pointer, minimum bounding rectangle, and digest of the ith child, respectively. The digest summarizes child nodes' MBRs (MBR_1-MBR_f), in addition to their digests ($H_1 - H_f$). As we discuss shortly, this design is vital to prove *completeness* of spatial query results. The digest of the root node H_{root} is signed by the DO and is stored with the tree.

To process a range query Q, the LBS invokes *RangeQuery* (Q, *root*), shown in Fig. 6, where symbols $e.P$ ($e.p$, $e.MBR$, $e.H$) denote the data object (child pointer, MBR, digest) stored in an entry e of a leaf node (internal node), respectively. The algorithm computes the verification object by following a depth-first traversal of the MR-tree. The *VO* contains three types of data: (i) all objects in each leaf node visited (line 4), (ii) the MBR and digests of *pruned* nodes (line 8), and (iii) special *tokens* [and] that mark the scope of a node (lines 1 and 9). New entries are always *appended* to the end of the *VO*.

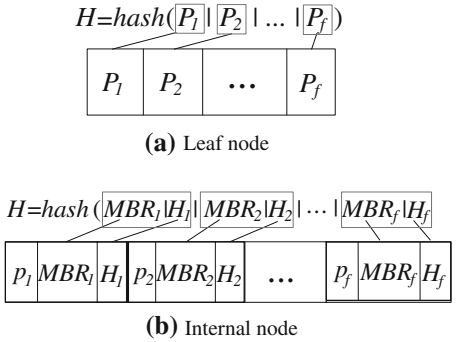


Fig. 5 MR-tree node structure

```
RangeQuery(Query Q, MR_Node N) // LBS
1.  Append [ to VO
2.  For each entry e in N
3.    If N is leaf
4.      Append e.P to VO
5.    Else                          // N is internal node
6.      If e.MBR overlaps Q, RangeQuery(Q, e.p)
7.      Else                        // a pruned child node
8.        Append (e.MBR, e.H) to VO
9.  Append ] to VO
```

Fig. 6 Query processing with the MR-tree

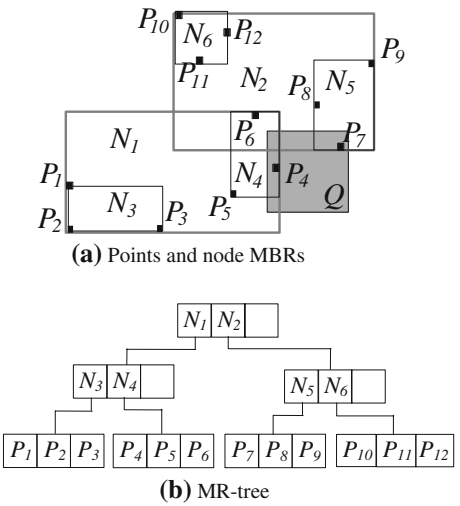


Fig. 7 Example range query

Consider, for instance, query Q in the example tree of Fig. 7. Similar to conventional R-trees, *RangeQuery* starts from the root and recursively visits all entries that overlap the shaded rectangle: N_1, N_2, N_4, N_5. After termination, the verification object is: [[($N_3.MBR$, $N_3.H$), [P_4, P_5, P_6]]], [[P_7, P_8, P_9], (N_6. MBR, $N_6.H$)]]. The tokens signify the contents of a node; for example, component [[($N_3.MBR$, $N_3.H$), [P_4, P_5, P_6]]] corresponds to the first root entry (N_1), and the rest of the *VO* to the second one (N_2). The LBS transmits the *VO* and the root signature s_{root} to the client. Note that the actual result (e.g., P_4, P_7) is part of the *VO*.

```
(MBRValue, HashValue)=RootHash(VerObj VO) // Client
1.  Set str=∅ and MBR=NULL
2.  While VO still has entries
3.    Get next entry e_V from VO
4.    If e_V is a data object P
5.      If P overlaps the query, Add P to the result set
6.      Enlarge MBR to include P
7.      str = str | P
8.    If e_V is a pair of MBR/digest (MBR_c, H_c)
9.      Enlarge MBR to include MBR_c
10.     str = str | MBR_c | H_c
11.   If e_V is [
12.     (MBR_c, H_c) = RootHash(VO)
13.     Enlarge MBR to include MBR_c
14.     str = str | MBR_c | H_c
15.   If e_V is ], Return (MBR, hash(str))
```

Fig. 8 Algorithm for re-computing H_{root}

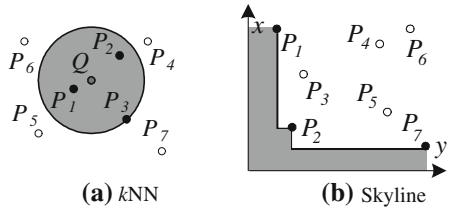
To verify the query results, the client first scans the *VO* to check that: (i) no MBR (of a pruned node) in the *VO* overlaps Q, and (ii) the computed H_{root} from the *VO* agrees with s_{root}. Figure 8 shows the recursive procedure *RootHash* that computes H_{root}. The main idea is to simulate the MR-tree traversal performed by the LBS, and calculate the MBR, and digests bottom-up. Note that the actual query results are contained in the *VO* and are extracted during *RootHash* (line 5).

In the example of Fig. 7, *RootHash* computes the MBR and digest of nodes N_4 (from $P_4 - P_6$), N_1 (from N_3, N_4),N_5 (from P_7 − P_9), N_2 (from N_5, N_6), *root* (from N_1, N_2), in this order. Note that *all* entries in the *VO*, from the [of the root to its], must be used. Furthermore, the algorithm is *online*, meaning that it performs a single sequential scan of the *VO*. The actual results (P_4, P_7) are extracted in line 5. Note that the client also receives some (boundary) objects (P_5, P_6, P_8, P_9) in the *VO*, which are not part of the result, but are necessary for its verification.

Proof of soundness: Assume that an object P in the result set is bogus or modified. Because the hash function is collision-resistant and$P$ must be used by *RootHash*, the re-computed H_{root} cannot be verified against s_{root}, which is detected by the client.

Proof of completeness. Let P be an object satisfying Q. Consider the leaf node N_l containing P. For the re-computed H_{root} to match s_{root}, either N_l's true contents or MBR/hash must be in the *VO*. In the former case, P is in the *VO* and is extracted in line 5 of *RootHash*. In the latter case, N_l's MBR overlaps Q, which alarms the client about potential violation of completeness.

In addition to range search, the MR-tree can authenticate other common spatial queries, including k *nearest neighbors* (kNN) and *skylines*. Given a point Q, a kNN query retrieves the k points from the data set that are closest to Q [19]. In the example of Fig. 9a, the three NNs of Q are P_1, P_2, and P_3, in increasing order of distance from Q. A key observation is that the kNN of Q lie in a circular area C centered at Q that



(a) *k*NN **(b)** Skyline

Fig. 9 Alternative queries

contains exactly k data points. Therefore, the LBS can prove the kNN results by sending to the client the *VO* corresponding to C. Specifically, it first finds the k neighbors, then it computes C, and finally executes *RangeQuery* treating C as the range. The verification process of the client is identical to the one performed for range queries.

A skyline query retrieves all points that are not *dominated* by others in the dataset [30]. A point P_i dominates another P_j, if and only if, the coordinate of P_i on each dimension is no larger than the corresponding co-ordinate of P_j. The skyline in Fig. 9b contains P_1, P_2 and P_7. To prove it, the LBS processes a range query that contains the area of the data space not dominated by any skyline point. This area (shaded in Fig. 9b) can be divided into multiple rectangles. The result contains only the skyline points, and can again be verified according to the methodology of range search.

Finally, our discussion assumes no access control restrictions so that the *VO* includes (boundary) records that do not belong to the query result (e.g., P_5, P_6, P_8, P_9 in the example of Fig. 7). If necessary, the LBS can hide the additional objects, by applying the following modifications based on [5,6,28]. First, the hash value H_N of a leaf node N (containing f points $N.P_1, N.P_2, \ldots, N.P_f$) is redefined as $H_N = hash(hash'(N.P_1)|hash'(N.P_2)|\ldots|hash'(N.P_f))$, where $hash'$ follows the definition of [28]. Second, each point P outside the result that must be present in the *VO* is replaced by a pair $\langle P_{ref}, H'_P \rangle$ [5,6], where (i) P_{ref} is a reference point not necessarily in the dataset, and (ii) H'_P encodes the relationship between P_{ref} and P, such that the client is able to compute $hash'(P)$ with P_{ref} and H'_P, and at the same time verify that P is indeed outside the query Q. An alternative approach to hide the boundary points is based on *zero knowledge proofs* [16].

3.2 Cost models

The important performance metrics for authenticated structures are (i) index construction time, (ii) index size, (iii) query processing cost, (iv) size of the *VO*, and (v) verification time. The first metric affects the party that builds the index, i.e., depending on the system, the DO or the LBS. The second one burdens the LBS and, in some cases, the DO (if it also has to maintain the index). Furthermore, it affects the communication cost between the two. Metric (iii) is important only for the LBS. The size of the *VO* influences the network overhead between the LBS and the client. Finally, the verification time burdens exclusively the client. In the sequel, we analytically compare the MR-tree and the VR-tree on the above metrics. Table 1 summarizes the symbols used in the analysis, as well as their typical values (1 ms $= 10^{-3}$ s, 1 μs $= 10^{-6}$ s). These values were obtained based on the hardware and software settings of our experiments, using the Crypto++ library. Our measurements are similar to those of the library benchmarks [9] and the values suggested in [20].

Table 1 Symbols and values in the analysis

Symbol	Description	Value	
C_s	Cost of *sign* operation	3.4 ms	
C_v	Cost of *verify* operation	160 μs	
C_H	Cost of *hash* operation	28 μs	
C_m	Cost of *multiply* operation	43 μs	
C_{NA}	Cost of a random node access	15 ms	
S_s	Size of a signature	128 bytes	
S_H	Size of a digest	20 bytes	
S_M	Size of an MBR	32 bytes	
S_P	Size of a data point	16 bytes	
$\|D\|$	Data cardinality	2,000,000	
d	Data dimensionality	2	
Q_l	Query extent on one dimension	10% of space	
b	Block size	4096 bytes	
f_l	Fanout of leaf node	VR 19	MR 179
f_n	Fanout of internal node	VR 17	MR 51
g	Height of the tree	VR 5	MR 4

We first establish a simple cost model for the R-tree, based on the fact that in d-dimensional unit space $[0,1]^d$, the probability that two random rectangles r_1, r_2 overlap is

$$P_{\text{overlap}}(r_1, r_2) = \prod_{j=1}^{d} (r_1.l_j + r_2.l_j) \tag{3.1}$$

where $r.l_j$ denotes rectangle r's extent along the jth dimension [29]. For simplicity, we assume that the data set contains points (rectangular data are discussed in [35]) uniformly distributed in the unit space and query Q has equal length Q_l on all dimensions. Let $f_l(f_n)$ be the average fanout of a leaf (internal) node, and $|D|$ be the data cardinality. The number of leaf nodes is $|D|/f_l$, and the height of the R-tree is $g = 1 + \lceil \log_{fn}(|D|/f_l) \rceil$. The number of internal nodes at depth i of the tree (assuming a complete tree where the root has depth 0) is f_n^i, each containing $|D|/f_n^i$ data points in its sub-tree. Because of the uniform distribution, the number of points in a node is proportional to the space covered by this node. Following [35], we assume that all nodes at the

same level are squares with similar sizes. Therefore, a node at depth i covers $1/f_n^i$ space, and has length $\sqrt[d]{1/f_n^i}$ on each dimension. Applying Eq. 3.1, the total cost of processing Q using the VR- or the MR-tree is

$$C_Q = C_{NA}\left(\sum_{i=0}^{g-2} f_n^i\left(\sqrt[d]{1/f_n^i}+Q_l\right)^d + f_l\left(\sqrt[d]{f_l/|D|}+Q_l\right)^d\right) \tag{3.2}$$

where C_{NA} is the cost of a node access. Similarly, the storage overhead of both the VR- and the MR-tree can be estimated by

$$S_{\text{index}} = b\left(\sum_{i=1}^{g-2} f_n^i + |D|/f_l\right) \tag{3.3}$$

where b is the block size. The difference between the two structures regards the authentication information, leading to different fanouts (f_l, f_n). The VR-tree maintains one signature (128 bytes) per entry in every node (leaf or internal). In contrast, the MR-tree adds digests (20 bytes each) only to internal nodes. Assuming a page of 4 KBytes, 70% average storage utilization and double precision, the VR-tree has a fanout of $f_l = 19$ (leaf) and $f_n = 17$ (internal), while for the MR-tree $f_l = 179$ and $f_n = 51$. The lower fanout of the VR-tree increases its height.

Besides R-tree generation, the VR-tree requires a signature for each object and node. The MR-tree only involves cheap computations of digests for nodes (but not objects). If the cost of a *sign*/*verify*/*hash* operation is C_s, C_v, C_H, respectively, the initial construction overhead of the VR-tree (MR-tree) is given by Eq. 3.4 (3.5):

$$C_{init}^{VR} = C_s\left(\sum_{i=1}^{g-1} f_n^i + n\right) \tag{3.4}$$

$$C_{init}^{MR} = C_s + C_H\sum_{i=0}^{g-1} f_n^i \tag{3.5}$$

Let the size of a signature, an MBR, a digest and a data point be S_s, S_M, S_H, and S_P, respectively. Then, the *VO* of the VR-tree with signature aggregation consumes space:

$$S_{VO}^{VR} = S_s + \sum_{i=0}^{g-2} f_n^{i+1}\left(\sqrt[d]{1/f_n^i}+Q_l\right)^d S_M + |D|\left(\sqrt[d]{f_l/|D|}+Q_l\right)^d S_P \tag{3.6}$$

where the last two terms estimate MBRs and points for visited internal and leaf nodes, respectively. Note that with signature aggregation, there is a single signature, thus the *VO* size is relatively small. To prepare this *VO*, however, the LBS must perform modular multiplications, whose cost is

$$C_{VO}^{VR} = C_m\left(\sum_{i=0}^{g-2} f_n^{i+1}\left(\sqrt[d]{1/f_n^i}+Q_l\right)^d + |D|\left(\sqrt[d]{f_l/|D|}+Q_l\right)^d\right) \tag{3.7}$$

Thus, the total query processing overhead for the VR-tree is the sum of the two costs expressed in Eqs. 3.2 and 3.7. The *VO* size of the MR-tree is given by Eq. 3.8. The complicated part is to analyze the total number of *pruned nodes* during query processing. $PN(i)$ estimates the number of pruned nodes at depth i, by computing the number of nodes outside Q, subtracted by descendents of higher pruned nodes.

$$S_{VO}^{MR} = \sum_{i=0}^{g-1} PN(i)\,(S_H+S_M)+|D|\left(\sqrt[d]{f_l/|D|}+Q_l\right)^d S_P$$
$$PN(i) = f_n^i\left(1-\left(\sqrt[d]{1/f_n^i}+Q_l\right)^d\right)-\sum_{j=0}^{i-1} PN(j)\,f_n^{i-j} \tag{3.8}$$

Finally, we estimate the verification time for the client, which is dominated by modular multiplications (VR-tree) or digest computations (MR-tree). The costs of the VR-tree (with signature aggregation) and the MR-tree are given by Eqs. 3.9 and 3.10. The MR-tree has a clear advantage because (i) for each node, the MR-tree invokes the hash function once, whereas the VR-tree performs modular multiplication for each entry, and (ii) $C_H < C_m$.

$$C_{\text{Client}}^{VR} = C_v + C_m\left(\sum_{i=0}^{g-2} f_n^{i+1}\left(\sqrt[d]{1/f_n^i}+Q_l\right)^d + |D|\left(\sqrt[d]{f_l/|D|}+Q_l\right)^d\right) \tag{3.9}$$

$$C_{\text{Client}}^{MR} = \sum_{i=0}^{g-1} f_n^i\left(\sqrt[d]{1/f_n^i}+Q_l\right)^d C_H + C_v \tag{3.10}$$

Table 2 shows the costs calculated by the above-mentioned equations using the typical values of Table 1. The VR-tree incurs about 30 times the overhead of the MR-tree for computing the authentication information (in the entire tree), and is 8 times larger. The MR-tree is also significantly better in terms of query processing and verification cost. The latter is particularly important because the clients are mobile devices with limited computing power. The only aspect where the two structures are similar is *VO* size. In the following section we present an optimized version of the MR-tree, namely the MR*-tree, for reducing the *VO*.

Table 2 Comparison of estimated costs

Costs	MR-tree	VR-tree
Construction overhead	4 s	2 h
Index size	57 MBytes	511 MBytes
Query processing time	2 s	22 s
VO size	390 KBytes	398 KBytes
Verification time	41 ms	991 ms

4 MR*-tree

Recall from Sect. 3.1 that in the MR-tree, each node consists of a simple list of entries. During query processing (Fig. 6), the LBS inserts into the *VO* authentication information (i.e., MBR/digests for internal nodes, points for leaf nodes) for every pruned entry. In practice, a node may contain a large number of entries (51 in an internal node and 179 in a leaf node according to Table 1), and the query often overlaps with only a fraction of these entries, especially at higher levels. Consequently, many of them are pruned and their contents are inserted into the *VO*. As an extreme case, consider a query which only overlaps with one entry per level of the tree and retrieves only one data object. Assuming that the MR-tree has a height of 5, $(51-1)\cdot 4 = 200$ MBR/digest pairs and 179 points are present in the *VO*, while the result set is merely one point.

Motivated by this observation, we propose the *MR*-tree*, an optimized version of the MR-tree that aims at decreasing the *VO* size and improving the communication overhead. The basic idea is to organize the entries of each node using an *embedded* ADS. The embedded ADSs are only *conceptual*, meaning that they do not consume any disk space or decrease the node fanout. Hence the MR*-tree achieves exactly the same query processing cost as the original MR-tree in terms of I/O time. Section 4.1 presents the structure of the MR*-tree, and Sect. 4.2 the query processing algorithms.

4.1 Structure

An embedded ADS in a leaf node is based on the KD-tree [3], whereas that in an internal node on the box-KD-tree [2]. We first clarify the embedding of leaf nodes. Let d be the database dimensionality and N_l an arbitrary MR*-tree leaf. A d-dimensional KD-tree T_{KD}[3] is constructed top-down with the text-book algorithm [3]. Then, in a subsequent step, we inject digests into its nodes bottom-up. Let n_{KD}=(P, lc, rc,h) be a node in T_{KD}, where P is a data object in N_l, and lc, rc are pointers to the left and right children of n_{KD}, respectively;

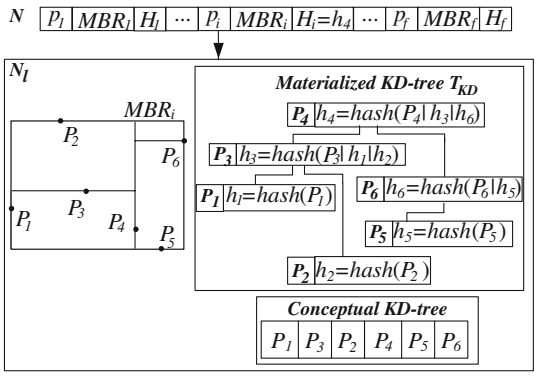


Fig. 10 MR*-tree leaf node structure

h is a hash value defined as follows: when n_{KD} is a leaf node (both lc, rc are empty), h is computed by hashing the binary representation of P. On the other hand, if n_{KD} is an internal node, h=$hash(P|lc.h|rc.h)$, i.e., the hash value of the concatenation of P with the digests of n_{KD}'s children. The root digest of T_{KD} summarizes authentication information about N_l, and is stored in the corresponding entry of N_l's parent. Note that in the MR-tree, the digest of N_l is derived from the direct concatenation of all data objects (Fig. 5a).

Figure 10 illustrates an example MR*-tree leaf node N_l that corresponds to entry (p_i, MBR_i, H_i) stored in parent N; p_i is the pointer to N_l's page and MBR_i is the MBR of the points residing in N_l, namely $P_1 - P_6$. The two-dimensional KD-tree T_{KD} over these points is constructed as follows. We first sort them according to their x-coordinate, yielding ordered list L:[P_1, P_2, P_3, P_4, P_5, P_6]. The median[4] P_4 is stored at the root of T_{KD}. P_4 splits L into two lists; L_1 : [P_1, P_2, P_3] and L_2 : [P_5, P_6]. We then sort L_1 and L_2 based on the y-coordinate, and store their medians (P_3 and P_6) into the left and right children of P_4, respectively. We continue recursively by switching the split axis in a round-robin fashion until all the points reside in exactly one node of T_{KD}. Subsequently, we compute the digests bottom-up. For example, the leaf node that includes P_1 stores $h_1 = h(P_1)$, and the internal node accommodating P_3 contains $h(P_3|h_1|h_2)$. The root digest h_4 is assigned to H_i in N_l's entry in parent N.

Materializing the KD-tree inside the MR*-tree node would decrease its fanout due to the additional digests and pointers. Instead, we only *conceptually* represent T_{KD} by storing the points according to the *in-order* traversal of the tree, discarding the produced digests and pointers. Note that there is always a one-to-one correspondence between a conceptual and a materialized KD-tree. Revisiting the example of Fig. 10, we derive the list [P_1, P_3, P_2, P_4, P_5, P_6] from T_{KD}, store it in N_l, and delete all digests and pointers. In this way,

[3] To avoid confusion between the components (i.e., nodes, entries, etc.) of the MR*-tree and the embedded ADSs, hereafter we append the subscript $_{KD}$ to all symbols denoting those of the latter category.

[4] The median of a list of even cardinality, as well as the switching order of the split axis, are arbitrarily pre-defined by the owner and known to the LBS.

the KD-tree can be built *on-the-fly* upon fetching the node page from the disk, without requiring any sorting operations.

Next, we discuss the embedded ADS of internal nodes in the MR*-tree, based on the box-KD-tree [2] that indexes rectangles. A box-KD-tree can be thought of as a $2d$-dimensional KD-tree, treating each d-dimensional rectangle (defined by $2d$ values) as a $2d$-dimensional point. For instance, a 2D rectangle defined by the two corner points $(x_{\min}, y_{\min})$ and $(x_{\max}, y_{\max})$ is perceived as a 4D point $(x_{\min}, y_{\min}, x_{\max}, y_{\max})$. Similarly, a d-dimensional range query can be converted to a $2d$-dimensional query expressed as the intersection of $2d$ half-planes, e.g., the query $(100 \leq x \leq 200, 500 \leq y \leq 600)$ is transformed to $(x_{\min} \leq 200, x_{\max} \geq 100, y_{\min} \leq 600, y_{\max} \geq 500)$.

Each node n_{BKD} of a box-KD-tree T_{BKD} contains a tuple of the form (e_i, lc, rc, h), where $e_i = (p, MBR, H)$ is an entry stored in the embedding MR*-tree node N_i, and lc, rc are pointers to the left and right children of n_{BKD}, respectively; h is a digest equal to $hash(MBR|H)$ when n_{BKD} is a leaf node, and $hash(MBR|H|lc.h|rc.h)$, otherwise. After building T_{BKD}, we re-arrange N_i's entries following the in-order traversal of T_{BKD}, store its root digest in N_i's parent, and discard T_{BKD}. The owner signs the root digest of the box-KD-tree constructed over the entries of the MR*-tree root.

4.2 Query processing

Upon receiving a range query Q, the LBS calls *RangeQuery** (Q, $root_{MR*}$), shown in Fig. 11. The algorithm fetches an MR*-tree node N (passed as an argument) from the disk, and invokes procedure *KDConstruct* (for leaf nodes) and *BKD-Construct* (for internal nodes), which re-builds an authenticated KD-tree/box-KD-tree over N's entries, depending on whether N is a leaf (line 1) or an internal node (line 4), respectively. We omit further details for these two subroutines since they are already explained in the descriptions. Subsequently, *RangeQuery** accesses the embedded ADSs by invoking algorithm *KDRangeQuery* (when N is leaf) or *BKDRangeQuery* (when N is internal node).

Figure 12 presents *KDRangeQuery*. Parameter a signifies the current split axis of the KD-tree T_{KD}. This recursive algorithm performs a depth-first traversal of T_{KD}. At each visited node n_{KD}, it first appends the data object $n_{KD}.P$ to the *VO*,

```
RangeQuery*(Query Q, MR*_Node N) // LBS
1.  If N is leaf
2.      root_KD=KDConstruct (N)
3.      KDRangeQuery(Q, root_KD, 0)          // Figure 12
4.  Else // N is internal node
5.      root_KD=BKDConstruct (N)
6.      BKDRangeQuery(Q, root_KD, 0)         // Figure 13
```

Fig. 11 Query processing with the MR*-tree

```
KDRangeQuery (Query Q, KD_Node n_KD, Axis a)
1.  Append [ to VO
2.  Append n_KD.P to VO
3.  sv=CalculateSplitValue(n_KD.P, a)
4.  If n_KD is internal node
5.      If sv < Q.a_min
6.          Append n_KD.lc.h to VO
7.      Else, call KDRangeQuery(Q, n_KD.lc, (a+1) mod d)
8.      If sv > Q.a_max
9.          Append n_KD.rc.h to VO
10.     Else, call KDRangeQuery(Q, n_KD.rc, (a+1) mod d)
11. Append ] to VO
```

Fig. 12 Algorithm *KDRangeQuery*

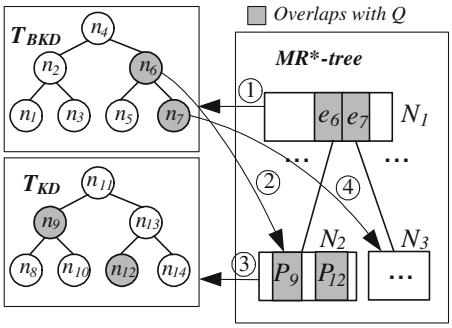
and then calculates the split value using $n_{KD}.P$ and a. When n_{KD} is an internal node, the algorithm determines whether a subtree can be pruned (lines 5 and 8). $Q.a_{\min}$ and $Q.a_{\max}$ denote the lower and upper boundary specified by Q on axis a. If a subtree is pruned, the corresponding digest is appended to the *VO* (lines 6 and 9). Otherwise, traversal continues in the respective subtree, changing the split axis to the next one (lines 7 and 10). Finally, tokens [and] mark the structure of the index to facilitate the verification process.

BKDRangeQuery in Fig. 13 follows a similar idea, but is more involved in two aspects. First, whereas each KD-tree node stores a data object and *KDRangeQuery* simply inserts it to the *VO*, a box-KD-tree node n_{BKD} contains a MR*-tree entry e_i=(p, *MBR*, H), which is included in a MR*-tree internal node N_i. Recall that p is a pointer to a MR*-tree node N_c (which is a child of N_i), and *MBR* and H is the MBR and digest of N_c, respectively. The algorithm first appends to the *VO* the $e_i.MBR$ value of every visited n_{BKD} node (line 3). Observe that this is an important difference between the MR-tree and the MR*-tree. The former only inserts the MBRs of the pruned nodes to the *VO*, since the rest can be re-constructed from information included in lower levels of the tree. On the other hand, the latter must include in the *VO* all the node MBRs, because they cannot be re-constructed from lower levels, due to the use of embedded ADSs. Subsequently, *BKDRangeQuery* checks whether $e_i.MBR$ overlaps Q; if so, query execution temporarily pauses in n_{BKD}, and continues in the MR*-tree traversal by calling *RangeQuery**, which follows pointer p (line 5). On the other hand, if Q does not overlap *MBR*, the algorithm inserts the digest H of the corresponding MR*-tree node into the *VO* (line 6). Later, we further elaborate on the interaction between the global MR*-tree and the embedded box-KD-trees through an example.

The second complication of *BKDRangeQuery* is that the box-KD-tree has dimensionality $2d$ instead of d. In our notation, we assume that the first d dimensions correspond to the lower boundary of the rectangles, and the last d dimensions to the upper boundaries. Therefore, when the current axis a signifies an upper boundary, the algorithm checks whether the corresponding lower boundary of Q is greater than the

BKDRangeQuery (*Query Q*, *BKD_Node* n_{BKD}, *Axis a*)
1. Append [to *VO*
2. Let e_i be the MR*-tree entry stored in n_{BKD}
3. Append $e_i.MBR$ to *VO*
4. If *Q* overlaps with $e_i.MBR$
5. *RangeQuery**(*Q*, $e_i.p$) // Figure 11
6. Else, Append $e_i.H$ to *VO*
7. *sv*=*CalculateSplitValue*($n_{BKD}.MBR$, *a*)
8. If n_{BKD} is *internal node*
9. If ($a \geq d$) AND ($Q.(a\text{-}d)_{min} > sv$)
10. Append $n_{BKD}.lc.h$ to *VO*
11. Else, *BKDRangeQuery*(*Q*, $n_{BKD}.lc$, (a+1) mod 2*d*)
12. If ($a < d$) AND ($Q.a_{max} < sv$)
13. Append $n_{BKD}.rc.h$ to *VO*
14. Else, *BKDRangeQuery*(*Q*, $n_{BKD}.rc$, (a+1) mod 2*d*)
15. Append] to *VO*

Fig. 13 Algorithm *BKDRangeQuery*



VO: [$e_4.MBR$, $e_4.H$, $n_2.h$,[$e_6.MBR$, [P_{11}, [P_9, $n_8.h$, $n_{10}.h$], [P_{13}, [P_{12}], $n_{14}.h$]], $n_5.h$, [$e_7.MBR$, N_3_*VO*]]]

Fig. 14 Query processing example

split value to determine if a sub-tree can be pruned (line 9) and vice versa (line 12).

The example of Fig. 14 provides more insight about the query processing algorithms using an MR*-tree with two levels. *RangeQuery** first fetches the MR*-tree root N_1 from the disk, constructs box-KD-tree T_{BKD} over its entries, and calls function *BKDRangeQuery* to access it. Each box-KD-tree node n_i corresponds to entry N_1. e_i, for $1 \leq i \leq 7$, and each KD-tree node n_j corresponds to data object P_j for $8 \leq j \leq 14$. Entries/nodes overlapping with *Q* are shown in grey color. *BKDRangeQuery* first reaches n_4 in T_{BKD}, obtains e_4 contained in n_4, and appends $e_4.MBR$ to the *VO*. Since *Q* does not overlap with it, the algorithm inserts $e_4.H$ into the *VO*.

Given that *Q* is disjoint with the left sub-tree of n_4, the LBS inserts $n_2.h$ into the *VO* and visits the right child n_6 of n_4. Subsequently, it adds $n_6.MBR$ to the *VO* and, since *Q* overlaps with this MBR, it invokes *RangeQuery** which retrieves MR*-tree node N_2, following the pointer in e_6. Assuming N_2 is a leaf node of the MR*-tree, the LBS builds KD-tree T_{KD} over its points. *RangeQuery** next calls *KDRangeQuery* to access T_{KD}, which appends components [P_{11}, [P_9, $n_8.h$, $n_{10}.h$], [P_{13}, [P_{12}, $n_{14}.h$]] to the *VO*. Note that

HashValue=**RootHash*** (*VerObj VO*) // Client
1. Set *str*=∅
2. While *VO* still has entries
3. Get next entry e_V from *VO*
4. If e_V is a data object *P*
5. If *P* overlaps the query, Add *P* to the result set
6. *str* = *str* | *P*
7. If e_V is a digest or an MBR
8. *str* = *str* | e_V
9. If e_V is [
10. *hash_c* = *RootHash**(*VO*)
11. *str* = *str* | *hash_c*
12. If e_V is], Return *hash*(*str*)

Fig. 15 Verification algorithm

P_{11}, P_9, P_{13} and P_{12} are contained in nodes n_{11}, n_9, n_{13} and n_{12}, respectively. When *KDRangeQuery* on T_{KD} terminates, so does *RangeQuery** on N_2. The processing of *Q* backtracks to *BKDRangeQuery* on N_1, which was visiting node n_6. Continuing the traversal in T_{BKD}, the algorithm inserts $n_5.h$ into the *VO*, then appends $e_7.MBR$ and since *Q* overlaps $e_7.MBR$, N_3 is retrieved and processed similarly to N_2, adding a partial *VO* N_3_*VO*. This concludes the execution of *Q* and completes the *VO*, shown at the bottom of Fig. 14.

The verification process for the MR*-tree is similar to that of the MR-tree. Specifically, the client checks whether (i) no MBR (of a pruned node) in the *VO* overlaps *Q*, (ii) for each pruned KD- or box-KD- tree node, *Q* does not overlap with its corresponding side of the split value on the axis of the level it resides in, and (iii) the computed root digest from the *VO* agrees with the DO's signature. *RootHash**, shown in Fig. 15, has one major difference from that of the MR-tree. When using the MR*-tree, the client does not compute the node MBRs bottom up as in the case of the MR-tree (line 6 and 9 of Fig. 8); instead, the MBRs of all visited nodes are explicitly given in the *VO* (line 3 of Fig. 13). In the example of Fig. 14, the MBRs of nodes N_2 and N_3($e_6.MBR$ *and* $e_7.MBR$, respectively) are included in the *VO*. This is because the MR*-tree is essentially a complex data structure combining heterogeneous components (KD-tree and box-KD-tree as embedded ADSs, R*-tree as the global one), and it is impossible to obtain MBR information of an R-tree entry from its offspring, which are KD- (or box-KD-) nodes. For example, in Fig. 14, since none of the nodes in the KD-tree T_{KD} stores the MBR of its sub-tree, we cannot compute the MBR of N_2 based on this T_{KD}. Finally, similar to the case of the MR-tree, the client extracts from the *VO* the result set of the query during *RootHash** (line 5).

Proof of soundness. Consider a MR*-tree leaf node N_l. Recall that the root digest H_l of the KD-tree constructed over the points in N_l summarizes authentication information about all the points in N_l. H_l is stored in N_l's parent N_i and, in a similar manner, it participates in the generation of the root digest of the box-KD-tree built over the entries of N_i, and so on. In other words, authentication information about *each*

valid point is involved in the digest that the owner eventually signs. Therefore, if the LBS modifies a point, or adds a bogus one, the change will propagate up to the re-constructed root digest (due to the collision resistance of the hash function), which will not be matched against the signature.

Proof of completeness. Consider an arbitrary node n of the KD-tree constructed over the points of MR*-tree leaf N_l, and suppose point $n.P$ satisfies query Q. If the LBS simply omits including $n.P$ in the *VO*, the client will compute a digest that does not match the signature due to the collision resistance of the hash function. On the other hand, the LBS may "hide" $n.P$ by including $n.h$ in the *VO* without changing the root digest; e.g., for n_9 in Fig. 14, the *VO* may contain $n_9.h$ instead of P_9. However, this violates condition (ii), checked by the client during the verification process, i.e., that each node must be correctly pruned. If the LBS indeed hides P_9 under $n_9.h$, assuming the split axis for n_{11} is x, the client finds that Q actually overlaps with the left part of $P_{11}.x$, and detects that the presence of $n_9.h$ is wrong.

The MR*-tree occupies the same disk space as the MR-tree and, therefore, incurs identical construction cost and query processing overhead in terms of disk accesses. Yet, its conceptual indexing of the entries leads to considerably smaller *VO* size due to the effective pruning of entries that do not overlap with the query. The downside of the MR*-tree is that index construction, query processing and verification require more CPU cycles for the hash operations incurred by the embedded ADSs. This tradeoff, however, is usually desirable. From the perspective of the DO and LBS, since I/O cost is the dominating factor, this increase in CPU time deteriorates performance only marginally. For the client, receiving a large *VO* from the network is generally more expensive than local computations, especially for mobile devices. Finally, the extra CPU overhead imposed by the MR*-tree is also insignificant, since the hashing operation depends on the length of the underlying data; for KD- and box-KD-trees, the hash function is applied on the concatenation of merely two nodes. Our experimental evaluation supports these claims.

5 Outsourcing index maintenance

Most query authentication techniques assume that the data owner, rather than the LBS, is responsible for creating and updating the authenticated structures. For the MR-tree and the MR*-tree, however, it is beneficial to delegate maintenance tasks to the LBS, due to several reasons. First, the underlying R*-tree is a rather specialized structure. The DO may not have the software or the expertise to build and maintain it. Second, the update algorithms are expensive, and the DO may not be able to perform them, especially if the data are highly dynamic. The LBS, on the other hand, is assumed to contain the computational power required. Finally, the LBS has knowledge of the query workload and can fine-tune the various parameters (e.g., page size, minimum node utilization), whereas the DO is oblivious to the clients.

In this section, we propose algorithms for outsourcing tree maintenance that are (i) *secure*, i.e., the DO can verify the changes made by the LBS, (ii) *efficient* in terms of processing cost and space overhead for both parties, and (iii) *effective*, meaning that most of the workload is performed by the LBS. Section 5.1 focuses on the initial index construction, and Sect. 5.2 deals with updates.

5.1 Initial construction

Traditionally, in the *initial construction* phase, the DO builds an index and transfers it to the LBS. Clearly the DO performs the entire workload and the LBS passively receives data. In contrast, our solution consists of three interactive steps between the two parties. First, the DO forwards only the data to the LBS. The LBS builds the index, and sends back information about the tree to the DO. Finally, the DO computes and signs the digest of the root node, and transmits it to the LBS.

Specifically, after the LBS has received all data and constructed the tree, it processes a range query Q_{full} covering the entire data space. Then, it transfers $VO(Q_{full})$ to the DO. $VO(Q_{full})$ includes all data points, but it does not contain any MBR or digest of internal nodes, because no node is pruned (all MBRs overlap Q_{full}). To reduce the size of $VO(Q_{full})$, the LBS inserts the ID of each point into the *VO* instead of its concrete value. Therefore, $VO(Q_{full})$ is essentially a list of all point IDs organized by [and] to show the structure of the index. Particularly for the case of the MR*-tree, entries in $VO(Q_{full})$ are also automatically sorted by the query processing algorithm, according to the pre-order traversal of the embedded KD-trees (for leafs) or the box-KD-trees (for internal nodes).

Upon receiving $VO(Q_{full})$, the DO checks that no point ID is missing or repeated, and that all IDs are valid. Furthermore, for the MR*-tree it verifies the proper order of the entries. After that, the DO computes the tree root digest H_{root} using its local copy of the points, signs it to create the signature s_{root}, and returns s_{root} to the LBS. Recall that the LBS sends s_{root} along with a *VO* in response to every client query. We next prove that, using the above construction scheme, it is impossible for the LBS to violate the soundness and completeness of any future query result, without being caught by the clients.

Proof of soundness. Suppose that the LBS modifies an existing (or inserts a bogus) point P in the result of an arbitrary client query Q. According to the proposed scheme, during the construction phase, the DO receives only point IDs from the LBS and, thus, produces the signature s_{root} solely on its genuine data (stored locally). Therefore, s_{root} does not cover any modified (or bogus) point, e.g., P. Due

to the collision resisting property of the hash function, the root hash re-constructed by the client cannot match s_{root}, triggering the alarm.

Proof of completeness. Assume that the LBS omits a point P satisfying an arbitrary client query Q from its result. Recall that, before generating s_{root}, the DO verifies the presence of all point IDs, meaning that all data points, including P, are incorporated into s_{root}. Therefore, in order for the client to establish the correct root digest, the LBS must include in the *VO* a valid digest H that summarizes hash information about P. In other words, the LBS prunes a sub-tree (of the MR-tree or of an embedded tree in the MR*-tree) that contains P during query processing. As discussed in Sects. 3 and 4, the client catches this cheat by checking the MBR (in case of MR-tree) or split value (in case of MR*-tree) accompanying the hash value H. Note that the LBS cannot falsify any MBRs / split values, since the owner computes them itself, based on genuine copies of data records, and incorporates them into s_{root}.

Our approach incurs similar communication overhead with the traditional method, since both entail transmission of the data and the tree. However, we outsource the expensive R*-tree construction (usually several hours) to the LBS, leaving only the cheap (a few seconds) index authentication to the DO. Furthermore, the DO does not need to store the tree locally, since, as we show next, updates are performed exclusively at the LBS.

5.2 Updates

We use the update algorithms of the R*-tree, but other techniques can also fit in our framework. Deletions in the R*-tree are rather simple. Let P be the object to be deleted. First, we locate (through a depth-first traversal) the leaf node N_l containing P, and remove P from N_l. If N_l *underflows* (i.e., the number of its entries is below the minimum utilization threshold), we delete N_l and re-insert all its entries in the tree. On the other hand, insertions are more complex. The insertion of P to a leaf N_l can lead to three cases: (i) N_l has space and P is inserted there; (ii)N_l overflows; a percentage of its entries that are farthest from the center of N_l are deleted and re-inserted to the tree; (iii) N_l still overflows after re-insertion (i.e., all deleted entries are re-inserted in N_l). In this case, N_l is split into two nodes. The effects of insertions and deletions may propagate the tree upward. Since both operations are rather expensive, they should be outsourced to the LBS.

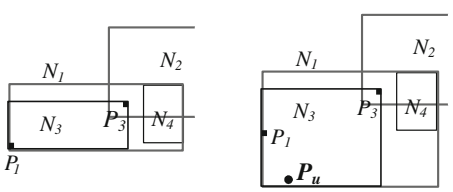
For each insertion or deletion, the DO sends an update request to the LBS, which processes it and returns *a series of VOs* (to be explained shortly). The DO verifies the *VOs* using the old signature, computes a new root signature, and transfers it back to the LBS. The complicated part concerns the generation of *VOs* when the LBS handles the update. In the following we discuss three different cases in increasing complexity order: (i) no re-insertion or split, (ii) re-insertion but not split, (iii) both re-insertion and split.

The simplest case occurs when the LBS inserts (or deletes) a point in a leaf node, without any further complications (i.e., no overflow/underflow). Figure 16 illustrates an example assuming that the capacity is 3 entries per node. A new point P_u is to be inserted to leaf node N_3. Since initially (Fig. 16a) N_3 contains two points, the insertion of P_u does not cause overflow. In this case, only a single *VO* is necessary, constructed as follows: the LBS processes an artificial *update* query Q_u whose range overlaps with all nodes along the root-to-leaf path of N_3 and *only* these nodes (i.e., root, N_1, N_3). The resulting *VO* thus contains three types of information: (i) old root signature s_{root}, (ii) points originally contained in N_3 (e.g., P_1, P_3), and (iii) MBR/digests in each ancestor of N_3 that do not overlap with Q_u, e.g., from the root we obtain MBR/hash of N_2, from N_1 we obtain MBR/hash of N_4.

Note that in the case of the MR*-tree, whenever the LBS generates a new *VO*, it must do so *exactly* in the same manner as in the MR-tree, i.e., it does not build a KD-tree/box-KD-tree on the entries of every visited node, but rather includes *all* the MBR/hash pairs or points of the non-overlapping with Q_u entries. The reason is that KD-trees and box-KD-trees are not incrementally updatable. When the underlying entries change, the embedded ADSs must be rebuilt from scratch. To do so, the DO needs all the entries in the affected node. The MR*-tree query algorithms, which may prune entries, are thus not applicable.

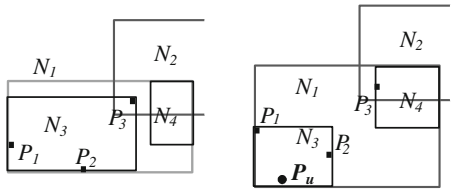
Upon receiving the *VO*, the DO first computes H_{root} using *RootHash* and verifies s_{root}. Next, it computes the new digest for the root bottom-up based on the insertion operation. Continuing the example of Fig. 16, it inserts P_u into N_3, and calculates the new $N_3.MBR$, $N_3.H$, $N_1.MBR$, $N_1.H$, H_{root}, in this order. In the MR*-tree case, during the new H_{root} computation, the DO must also construct the respective KD-tree/box-KD-tree for every affected node itself. Finally, it signs H_{root}, and sends the new signature to the LBS. Note that the insertion of P_u may alter the MBRs of all nodes (i.e., N_3, N_1) in the path to the root.

We now discuss the second case, which involves re-insertion, but no node splitting. Recall that re-insertion can be triggered by both insertions and deletions. Figure 17 shows an insertion example assuming that N_3 is full (it contains



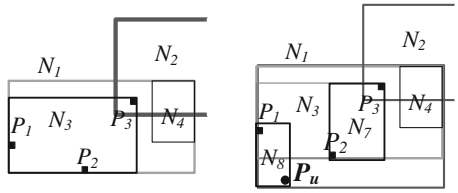
(a) Before insertion **(b)** After inserting P_u

Fig. 16 Insertion without re-insertions and split



(a) Before insertion **(b)** After inserting P_u

Fig. 17 Insertion with re-insertions



(a) Before insertion **(b)** After inserting P_u

Fig. 18 Insertion with split

P_1, P_2, P_3). The addition of P_u to N_3 will cause the deletion of P_3, and its re-insertion to the tree as a fresh point. In general, we may need to re-insert a set E of points from a leaf node N_l. The LBS first sends to the DO the IDs of all the points in E. Then, it deletes these points from N_l and treats each as a new insertion operation. For every insertion, a new *VO* is sent to the DO. Again, the *VO* generation is identical for the MR- and the MR*-trees. In the example of Fig. 17, the LBS sends (i) a *VO* when inserting P_u into N_3, (ii) a message that P_3 will be re-inserted, and (iii) a second *VO* when inserting P_3 into N_4. Each individual *VO* must be built with the most recent information. Only the first *VO* (VO_0) includes s_{root}; subsequent ones do not since the digest of the root has changed. Specifically, $VO_1, \ldots, VO_{|E|}$ must contain the *updated* contents of the node, including MBRs, digests, and data points. In the example of Fig. 17, VO_2 (sent during re-insertion of P_3) contains the *updated* MBR/digest of N_3, which now includes P_1, P_2 and P_u.

The DO, upon receiving the notification that a set of points E will be re-inserted, deletes these points from N_l and re-computes the root hash H^*_{root}. The DO keeps E to check whether the LBS inserts exactly all the points in E. Back to the example of Fig. 17, when VO_1 (for inserting P_3 into N_4) arrives, it has no signature in it and contains N_4 without P_3. The DO computes the root hash from VO_1, and verifies it against H^*_{root}. After that, the DO inserts P_3 into N_4 and re-computes the root hash again, signs it and sends it back to the LBS.

The third case deals with node splits. Figure 18 shows an example where P_u is inserted into N_3 as before, but P_3 is now re-inserted into N_3. The R*-tree then splits N_3 into two nodes N_7 and N_8. The LBS sends to the DO N_7 and N_8, which uses them to re-compute a new digest for the root. Next, the parent of N_3, namely N_1, is changed by removing the entry for N_3 and inserting two entries for N_7 and N_8. Both parties perform this change and update the MBR/hash pairs bottom-up. If N_1 does not overflow, the insertion is over and the DO sends to the LBS the signature of the new H_{root}. If, however, N_1 overflows, we must trigger re-insertion/split to treat it.

Figure 19 summarizes the algorithms for *VO* generation during insertion, re-insertion and split, assuming the underlying structure is the MR-tree. Each call to the function

UpdateQuery (*MR_Node N, MR_Node* N_e)
// same as *RangeQuery* (Figure 6), except for line 6
6. If *e.p* points to an ancestor of N_e *or* N_e *itself*, *UpdateQuery*(*e.p*, N_e)

Insert (*MR_Entry e*)
// *e* can be either a leaf entry (i.e. a data point) or a non-
// leaf entry. *Delete*(*MR_Entry e*) is similar and omitted for
// brevity.
1. Let R*-tree decide a node N_e to put *e* in
2. Call *UpdateQuery*(*root*, N_e)
3. Insert *e* into N_e and update all MBR/hash pairs bottom up // Case 1
4. If N_e overflows
5. If the R*-tree decides to re-insert
6. Call *ReInsert*(N_e) // Case 2
7. Else call *Split*(N_e) // Case 3

ReInsert (*MR_Node N*)
1. Let R*-tree decide the set of entries *E* of *N* that will be re-inserted
2. Send *E* to the DO
3. Delete all entries in *E* from *N* and update all *MBR/hash* pairs bottom-up
4. For each entry *e* in *E*, call *Insert*(*e*)

Split (*MR_Node N*)
1. Let the R*-tree split *N* into two nodes N_1 and N_2
2. Send the contents of N_1 and N_2 to the DO
3. Create two non-leaf entries e_1 (for N_1) and e_2 (for N_2)
4. Delete the entry for *N* in its parent N_p and insert e_1 and e_2; update all *MBR/hash* pairs bottom up
5. If N_p overflows, follow the R*-tree's decision and call *ReInsert*(N_p) or *Split*(N_p)

Fig. 19 Algorithms for update outsourcing

UpdateQuery produces a new *VO*, which is subsequently transmitted to (and verified by) the data owner. For the MR*-tree, besides notational differences (e.g., all *MR_Nodes* are replaced by *MR*_Node*), the only significant change is in the computation of hash values (i.e., line 3 in *Insert*, line 3 in *ReInsert*, and line 4 in *Split*). Specifically, given an MR*-node *N*, the DO first builds its corresponding KD-tree/box-KD-tree in main memory, and generates its root hash accordingly as the hash value for *N*. Finally, we mention that (i) deletions are authenticated in a similar way as insertions, and (ii) a record update is handled by a deletion of the old version, followed by an insertion of the new one.

We next prove that the proposed scheme preserves the soundness and completeness of all future query results.

Proof of soundness. Suppose that the LBS inserts a falsified point P in the result of a query Q. As described in Sect. 5.1, the initial signature involves only genuine information. In the following, we prove that if before an update the root signature does not cover the falsified point P, then P is not involved in the new signature produced after the update. Thus, by induction, no signature contains any information about P. Consequently, due to hash collision resistance, the client rebuilds a root hash with $VO(Q)$ that does not agree with the owner's signature.

Specifically, the DO computes the new root signature based on the a series of VOs $\{VO_0, \ldots, VO_{|E|}\}$ received from the LBS. The correctness of VO_0 is guaranteed by the previous signature, which, according to our induction assumption, does not cover P. Therefore, neither VO_0, nor the new root hash H^0_{root} generated based on VO_0 and the insert operation, involves P. Subsequently, at the ith step ($1 \leq i \leq |E|$), assuming H^{i-1}_{root} does not cover P, since the DO verifies VO_i against H^{i-1}_{root}, VO_i must not contain P in order to pass the checking. Meanwhile, since the DO generates H^i_{root} based on VO_i, it is impossible for H^i_{root} to contain any information of P due to hash collision resistance. By induction, the last root hash $H^{|E|}_{root}$ does not cover P, thus, neither does the resulting s_{root} computed by signing $H^{|E|}_{root}$.

Proof of completeness. A straightforward extension of the above proof of soundness shows that the LBS is unable to fabricate the index structure, including all MBRs (in case of the MR-tree) or split values (for the MR*-tree). Query result completeness is obtained by applying the same reasoning as in the proof presented in Sect. 5.1.

Finally, we discuss two issues related to updates. The first is *query freshness*, i.e., the client must ensure that its results are based on the most recent version of the DO's data. This problem is similar to confirming the validity of digital certificates, which has been extensively studied in the past. Here we describe two popular solutions, both of which are discussed at length in [20]. The first utilizes a certificate revocation scheme, e.g., [26]. Specifically, with such a scheme, the DO publishes a list of revoked (e.g., due to point updates) signatures, and the client verifies that the signature in the VO is not present in that list. The second approach incorporates a time interval into each signature, which indicates its validity period, assuming no critical updates take place during this time span. After the validity period ends, the corresponding signature expires, and the DO re-issues a new one to the LBS. Since our methods involve a single signature, these solutions are rather efficient (as opposed to the VR-tree that involves numerous signatures). The second issue regards the case that the same DO outsources its data to multiple LBSs. In this situation, the DO chooses a random LBS to interact with in order to generate the new signature s_{root} for each update. It then simply transmits the data along with s_{root} to the other LBSs, which modify their local indices accordingly.

6 Experimental evaluation

All experiments were performed with a P4 3 GHz CPU. VR-, MR- and MR*-trees were implemented in C++ using the Crypto++ library [9] and an R*-tree with 4Kbytes page size as the basis for the tree implementations. Each experiment was repeated on two datasets: (i) UNI that contains 2 million uniformly distributed data points, and (ii) CAR that contains 2 million points taken from road segments in California [33]. In order to vary the cardinality, we randomly sampled from these datasets using an appropriate sampling rate. Section 6.1 compares the MR-tree and the VR-tree in terms of initial construction cost, disk space consumption, and query processing/verification overhead. Section 6.2 assesses the relative performance of the MR-tree and the MR*-tree. Section 6.3 measures the maintenance cost of the MR-tree and MR*-tree, at the LBS and the DO.

6.1 Comparison of MR-tree with the VR-tree

Figure 20 illustrates the construction cost for the VR- and the MR-trees as a function of the data cardinality. This cost includes both the time to create the trees and the time to compute the digests (MR-tree) or the signatures (VR-tree). The VR-tree is 1–2 orders of magnitude more expensive to build due to the numerous signatures. Since the VR-tree does not support construction outsourcing, this workload must be performed by the DO. The MR-tree is built by the LBS and is only verified by the owner.

Figure 21 shows the CPU time for computing the necessary authentication information. The MR-tree outperforms the VR-tree by 2–3 orders of magnitude on this metric. Comparing Figs. 20 and 21, the computation of signatures dominates the total construction cost of the VR-tree. On the other hand, the MR-tree involves cheap hashing operations, only for the nodes (and not the data points). Consequently, the overhead of the additional information (with respect to the R*-tree) constitutes a small fraction (less than 1%) of the total construction cost.

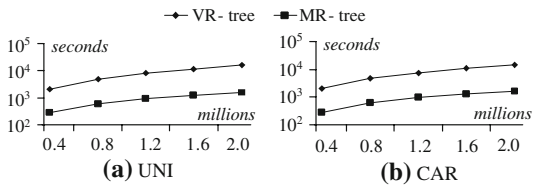


Fig. 20 Construction time vs. data cardinality

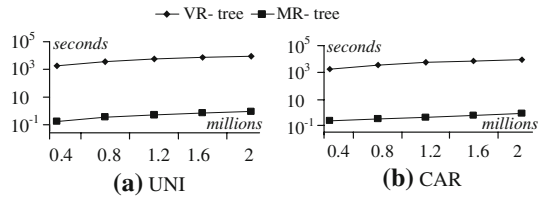


Fig. 21 CPU time for generating authentication information vs. data cardinality

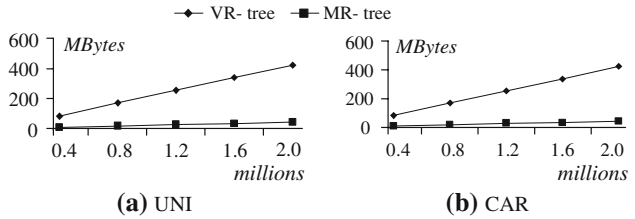


Fig. 22 Index size vs. data cardinality

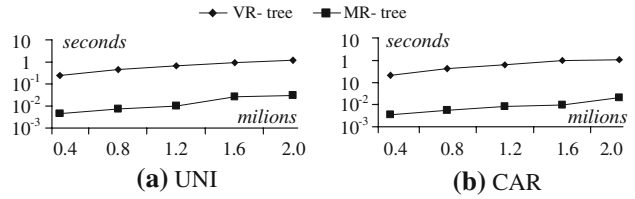


Fig. 23 Query cost vs. data cardinality

Figure 22 illustrates the size of the indices in MBytes. The VR-tree is much larger since it stores one signature (128 bytes) for each data point and node, whereas the MR-tree stores one digest (20 bytes) for every node. Furthermore, in case of the VR-tree, the DO needs to transmit the dataset and *all* the signatures to the LBS, leading to high network cost between them. For the MR-tree, the owner (i) sends to the LBS only the data and a single signature and (ii) receives back the IDs of the data points which are much smaller than the signatures.

We next evaluate the query and verification cost of the two structures. All queries are ranges with extent 10% of the axis length, i.e., each query covers 1% of the entire (2D) space (recall from Sect. 3, that other query types, such as NN, can be converted to ranges). For every experiment, we execute 100 queries at random locations and illustrate the average cost, which burdens the LBS and includes both the result retrieval and the construction of the verification object. Figure 23 illustrates the query cost (in seconds) as a function of the data cardinality. The MR-tree is fast because it creates the *VO* by simply appending MBRs and digests for each pruned node. The VR-tree is about 2 orders of magnitude slower due to the modular multiplications required to create the aggregated signature. Recall that signature aggregation is unavoidable because, otherwise, the *VO* would be extremely large.

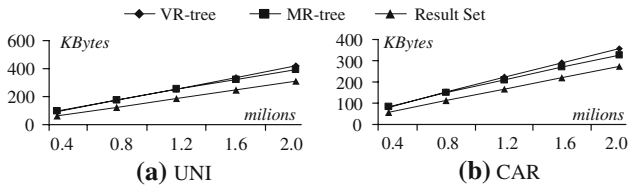


Fig. 24 *VO* size vs. data cardinality

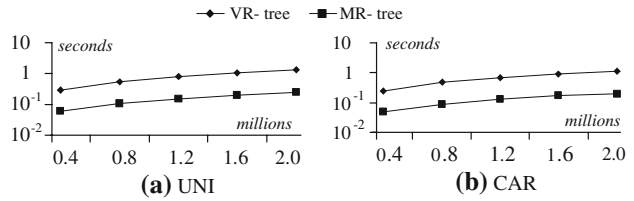


Fig. 25 Verification time vs. data cardinality

Figure 24 depicts the *VO* size versus the data cardinality. For small datasets, the *VO* of MR-trees and VR-trees have similar sizes. However, as the cardinality rises, the *VO* grows faster for the VR-tree because more intermediate MBRs are included in the *VO* (due to the smaller fanout). For comparison, the diagrams also illustrate the result size. The *VO* (of VR-trees and MR-trees) is larger than the corresponding result, because the result is always part of the *VO* (in fact, the result set dominates the total *VO* size).

Finally, Fig. 25 investigates the verification time (at the client) versus the data cardinality. The VR-tree leads to high cost since verification involves a number of modular multiplications, which is proportional to the output size. On the other hand, verification in the MR-tree invokes relatively cheap hash operations. Minimization of verification time is crucial for clients (e.g., PDAs) with limited computational resources.

Summarizing, the MR-tree is considerably faster to build and consumes less space than the VR-tree. At the same time, it is much more efficient for query processing and verification. The only aspect where the MR- and VR-tree perform similarly is the size of the *VO*.

6.2 Comparison between MR-tree and the MR*-tree

Next we compare the MR*-tree with the MR-tree. Recall that the motivation behind the MR*-tree is the reduction of the verification object. Figure 26 measures the *VO* size for the two indices given 100 random queries, each covering 1% of the data space. Since the *VO* is dominated by the result set (see Fig. 24), we only focus on the additional authentication information in the *VO*. Due to the use of embedded ADSs, the MR*-tree reduces the *VO* by more than 50% in most settings, and the performance gap widens with the data cardinality.

As discussed in Sect. 4, the MR*-tree consumes the same space as the MR-tree because the embedded structures are

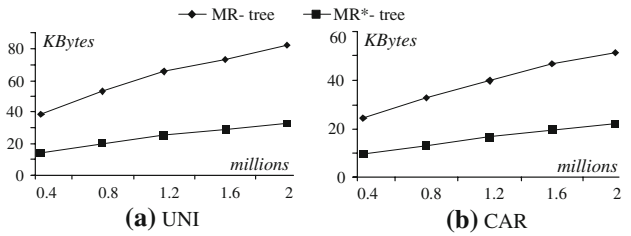


Fig. 26 Size of authentication information in the *VO* vs. data cardinality

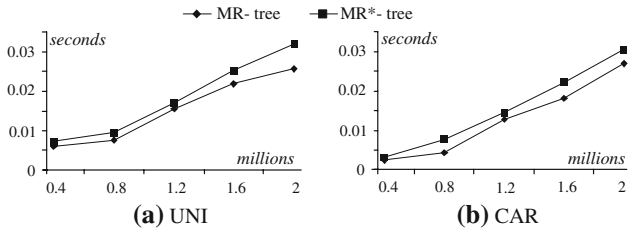


Fig. 28 Query cost vs. data cardinality

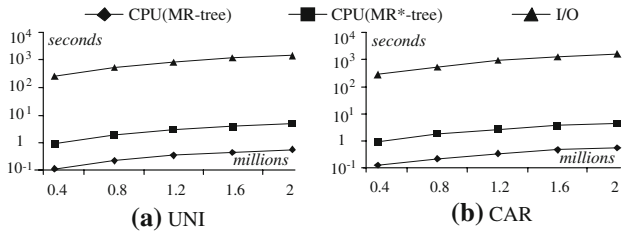


Fig. 27 CPU and I/O cost during index construction vs. data cardinality

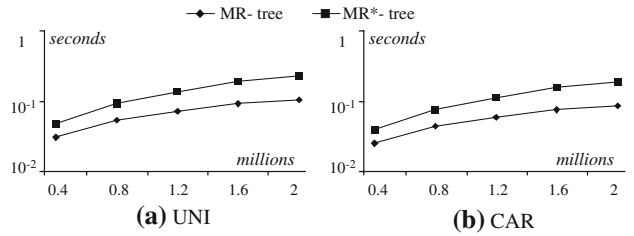


Fig. 29 Verification time vs. data cardinality

conceptual. This also implies identical cost in terms of I/O operations. However, the MR*-tree involves higher CPU cost due to the hash operations incurred by the embedded ADSs. In the next set of experiments, we measure this additional overhead. Figure 27 plots the CPU time for computing authenticated information of the MR- and MR*-tree, as well as the I/O cost for constructing the underlying R*-tree structure. In both datasets, the MR*-tree requires more CPU cycles since it has to build the embedded ADSs. This overhead, however, is negligible with respect to the cost for disk accesses, which is more than two orders of magnitude higher.

Figure 28 compares the two indices in terms of query processing cost. The MR*-tree has a slightly higher overhead, due to the fact that the embedded ADSs are not materialized, and their digests must be computed on-the-fly. The performance gap slowly grows as the data cardinality increases. This happens because larger datasets lead to the retrieval of more nodes and, thus, more embedded ADSs to construct. This effect is less pronounced in the CAR dataset than in UNI, because data objects in the former concentrate on a few dense regions; the queries, on the other hand, are uniformly distributed in space. This means that queries in CAR generally have a higher selectivity and fewer page accesses, which translates to less work for re-building embedded ADSs.

Figure 29 studies the verification overhead as a function of the data cardinality. The MR*-tree incurs higher cost because the client must verify the embedded ADSs in addition to the basic MR-tree structure, which involves hash operations and checks of the structures of the embedded ADSs. Nevertheless, the verification time (as well as the query cost and construction time) of the MR*-tree are well below those of the VR-tree.

In summary, the choice between the MR-tree and the MR*-tree depends on the application. The MR*-tree effectively reduces the *VO* size, and, thus, minimizes network latency; on the other hand, it induces more CPU overhead for query processing and verification. This trade-off is desirable in applications where the network is the main bottleneck (e.g., slow or unreliable wireless networks). Furthermore, the MR*-tree is preferable for battery-powered clients because transmission depletes the battery faster than offline computations [11]. On the other hand, if the network is fast and inexpensive, and the clients have no battery issues, the MR-tree may be a better choice.

6.3 Maintenance of the MR- and the MR*-trees

In this section we measure the tree construction and maintenance cost for the owner and the LBS. Our aim is to show that the burden of the owner is small compared to that of the LBS. Since the VR-tree does not include update algorithms, it is omitted from the diagrams. Figure 30 compares the initial construction cost at the LBS (which is identical to Fig. 20) and the corresponding time spent by the owner to verify it, as a function of the data cardinality. The LBS construction is dominated by disk accesses and the difference between the MR- and MR*- trees is negligible. For the DO, the MR*-tree incurs a marginally higher overhead due to the additional checking of the embedded ADSs. Clearly, the burden of the owner is considerably smaller than that of the LBS for both the MR-tree and the MR*-tree.

Assuming that an MR-tree with 2 million objects has already been built, Fig. 31 measures the maintenance cost as a function of the number of updates ranging between

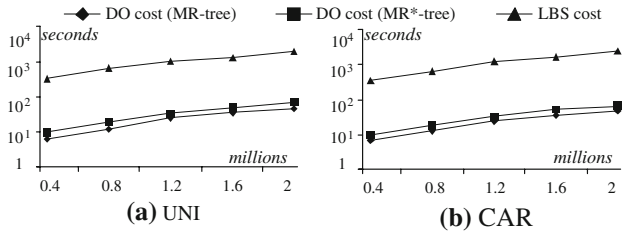


Fig. 30 Construction time vs. data cardinality

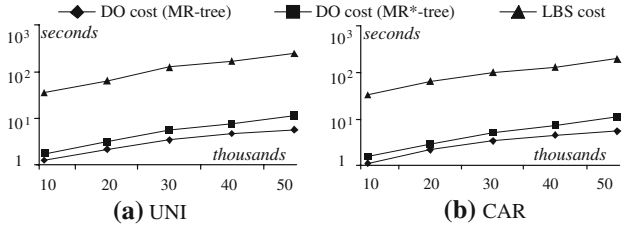


Fig. 31 Maintenance cost vs. # of updates

10K-50K. Specifically, we consider that for each update, the LBS alters the index and sends the new *VO* to the DO, which verifies it. All updates correspond to insertions. The diagram does not include the cost of signing, because this operation must be performed by the DO in any case. Again, for both the MR- and the MR*-trees, the owner's burden is a fraction of that of the LBS.

7 Conclusions

Recent advances in location-based services and sensor networks, as well as the popularity of web-based access to spatial data (e.g., *MapQuest*, *GoogleEarth*, etc.), necessitate query authentication for outsourced and replicated multi-dimensional data. In this paper we propose the MR-tree, an authenticated index based on the Merkle Hash tree and the R*-tree. Our method outperforms the best current solution by several orders of magnitude in many important metrics such as construction cost, index size and verification overhead. Furthermore, we develop the MR*-tree, an alternative to the MR-tree, which significantly reduces the communication overhead between the LBS and the client. In addition, both the MR-tree and the MR*-tree are fully outsourced, in the sense that their maintenance is performed entirely by the LBS, and can be verified with minimal effort by the data owner. We conclude our contributions with an extensive experimental study that validates the effectiveness and efficiency of the proposed structures.

As future work, we plan to extend our solutions to the problem of authenticating other query types that cannot be reduced to ranges, such as spatial joins (e.g., "find all cities that are crossed by a river"), aggregates ("find the number of objects in a given range instead of their IDs"), etc. Another direction concerns more flexible models that let the client decide whether a proof of query results is required from the LBS. A brute-force solution, for example, is to keep an ordinary index for query processing and a separate ADS for authentication. Finally, it would be interesting to investigate the integration of privacy-preserving techniques and query authentication models to reach a solution with both security guarantees.

Acknowledgement This work is supported by grant HKUST 6181/08 from Hong Kong RGC.

References

1. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. SIGMOD (2004)
2. Agrawal, P., de Berg, M., Gudmundsson, J., Hammar, M., Haverkroft, H.: Box-trees and R-trees with near-optimal query time. Discret. Comput. Geom. **28**(3), 291–312 (2002)
3. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry: Algorithms and Applications. Springer, Heidelberg (1997)
4. Beckmann, N., Kriegel, H.-P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. SIGMOD (1990)
5. Cheng, W., Tan, K.-L.: Query assurance verification for outsourced multi-dimensional databases. J. Comput. Secur. (to appear)
6. Cheng, W., Tan, K.-L.: Authenticating *k*NN query results in data publishing. Secure Data Management (2007)
7. Cheng, W., Pang, H., Tan, K.-L.: Authenticating multi-dimensional query results in data publishing. DBSEC (2006)
8. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Over-encryption: management of access control evolution on outsourced data. VLDB (2007)
9. Crypto++ library.: www.eskimo.com/~weidai/benchmark.html
10. Devanbu, P., Gertz, M., Martel, C., Stubblebine, S.: Authentic data publication over the internet. J. Comput. Secur. **11**(3), 291–314 (2003)
11. Datta, V., Vandermeer, D., Celik, A., Kumar, V.: Broadcast protocols to support efficient retrieval from databases by mobile users. ACM TODS. **24**(1), 1–79 (1999)
12. Damiani, E., Vimercati, C., Jajodia, S., Paraboschi, S., Samarati, P.: Balancing confidentiality and efficiency in untrusted relational DBMSs. CCS (2003)
13. Guttman, A.: R-trees: A dynamic index structure for spatial searching. SIGMOD (1984)
14. Goodrich, M., Tamassia, R., Triandopoulos, N., Cohen, R.: Authenticated data structures for graph and geometric searching. CT-RSA (2003)
15. Ge, T., Zdonik, S.: Answering aggregate queries in a secure system model. VLDB (2007)
16. Haber, S., Horne, W., Sander, T., Yao, D.: Privacy-preserving verification of aggregate queries on outsourced databases. HP Labs Tech-Report 2006-128 (2006)
17. Hacıgümüş, H., Iyer, B., Mehrotra, S.: Providing databases as a service. ICDE (2002)
18. Hacıgümüş, H., Iyer, B., Li, C., Mehrotra, S.: Executing SQL over encrypted data in the data-service-provider model. SIGMOD (2002)
19. Hjaltason, G., Samet, H.: Distance browsing in spatial databases. ACM TODS. **24**(2), 265–318 (1999)

20. Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L.: Dynamic authenticated index structures for outsourced databases. SIGMOD (2006)
21. Li, F., Yi, K., Hadjieleftheriou, M., Kollios, G.: Proof-infused streams: enabling authentication of sliding window queries on streams. VLDB (2007)
22. Merkle, R.: A certified digital signature. CRYPTO (1989)
23. Martel, C., Nuckolls, G., Devanbu, P., Gertz, M., Kwong, A., Stubblebine, S.: A general model for authenticated data structures. Algorithmica **39**(1), 21–41 (2004)
24. Mykletun, E., Narasimha, M., Tsudik, G.: Signature bouquets: immutability for aggregated/condensed signatures. ESORICS (2004)
25. Menezes, A., van Oorschot, P., Vanstone, S.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
26. Naor, M., Nissim, K.: Certificate revocation and certificate update. USENIX (1998)
27. Narasimha, M., Tsudik, G.: Authentication of outsourced databases using signature aggregation and chaining. DASFAA (2006)
28. Pang, H., Jain, A., Ramamritham, K., Tan, K.-L.: Verifying completeness of relational query results in data publishing. SIGMOD (2005)
29. Pagel, B., Six, H., Toben, H., Widmayer, P.: Towards an analysis of range query performance in spatial data structures. PODS (1993)
30. Papadias, D., Tao, Y., Fu, G., Seeger, B.: Progressive skyline computation in database systems. ACM TODS. **30**(1), 41–82 (2005)
31. Pang, H., Tan, K.-L.: Authenticating query results in edge computing. ICDE (2004)
32. Papadopoulos, S., Yang, Y., Papadias, D.: CADS: continuous authentication on data streams. VLDB (2007)
33. R-tree portal.: www.rtreeportal.org
34. Sion R.: Query execution assurance for outsourced databases. VLDB (2005)
35. Theodoridis, Y., Sellis, T.: A model for the prediction of R-tree performance. PODS (1999)
36. Tamassia, R., Triandopoulos, N.: Efficient content authentication in Peer-to-Peer networks. ACNS (2007)
37. Wong, W., Cheung, D., Hung, E., Kao, B., Mamoulis, N.: Security in outsourcing of association rule mining. VLDB (2007)
38. Xie, M., Wang, H., Yin, J., Meng, X. Integrity audit of outsourced data. VLDB (2007)