Spatial Outsourcing for Location-based Services

Yin Yang [1] Stavros Papadopoulos [1] Dimitris Papadias[1] George Kollios[2]
[1]Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{yini, stavros, dimitris}@cse.ust.hk

[2]Department of Computer Science
Boston University
Boston, MA, 02215
gkollios@cs.bu.edu

Abstract

The embedding of positioning capabilities in mobile devices and the emergence of location-based applications have created novel opportunities for utilizing several types of multi-dimensional data through spatial outsourcing. In this setting, a data owner (DO) delegates its data management tasks to a location-based service (LBS) that processes queries originating from several clients/ subscribers. Because the LBS is not the real owner of the data, it must prove (to each client) the correctness of query output using an authenticated structure signed by the DO. Currently there is very narrow selection of multi-dimensional authenticated structures, among which the VR-tree is the best choice. Our first contribution is the MR-tree, a novel index suitable for spatial outsourcing. We show, analytically and experimentally, that the MR-tree outperforms the VR-tree, usually by orders of magnitude, on all performance metrics, including construction cost, index size, query and verification overhead. Motivated by the fact that successive queries by the same mobile client exhibit locality, we also propose a synchronized caching technique that utilizes the results of previous queries to reduce the size of the additional information sent to the client for verification purposes.

1. Introduction

The embedding of positioning capabilities (e.g., GPS) in mobile devices has triggered several types of location-based services. Such services provide fresh opportunities for data sharing and utilization. Consider a data owner (DO) that possesses a proprietary spatial dataset, such as a specialized map overlay or a set of points of interest (e.g., local businesses). The DO can profit by allowing access to the dataset. However, the cost of setting up the infrastructure, hiring qualified personnel and advertising an online service may be prohibitive. Moreover, the value of the dataset will increase if it is combined with the functionality (e.g., driving directions, aerial photos, etc.) of a general-purpose online map. These reasons provide strong motivation for *outsourcing* the dataset to a specialized location-based service (LBS), which achieves economy of scale by servicing multiple owners.

To appear in ICDE 2008

Outsourcing of relational databases was first proposed in [HIM02]. In this paper, we focus on *spatial outsourcing*, motivated by the large availability of spatial data from various sources (e.g., satellite imagery, land surveys, environmental monitoring, traffic control). Often, agencies collecting such data (e.g., government departments, nonprofit organizations) are not able to support advanced query services; outsourcing to a LBS is the only option for utilizing the data. Furthermore, even if a DO possesses the necessary functionality, it may be beneficial in terms of cost, visibility, ease of access etc., to replicate the data in a LBS. The importance of spatial outsourcing is expected to soar with the increasing appearance of data sources and the emergence of novel mobile computing applications.

Our solutions follow the framework of Figure 1.1, adopted from relational database outsourcing. The 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. Since the LBS is not the real owner of the data, the client must be able to verify the *soundness* and *completeness* of the results. Soundness means that every record in the result set is present in the owner's database and not modified. Completeness means that no valid result is missing.

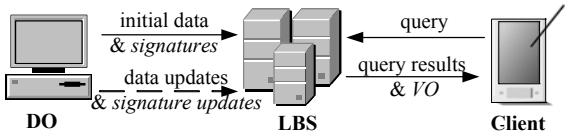


Figure 1.1 Database outsourcing framework

In order to process authenticated queries efficiently, the LBS indexes 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 establish soundness and completeness using the public key of the DO. A crucial part in this framework concerns the ADS. Specifically, the 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 [CPT06], which applies the *signature chain* concept [PJRT05] to KD-trees and R-trees. Although the R-tree based ADS, called VR-tree, is the best between the two options, it still has some serious drawbacks: large space and query processing overhead for the LBS, high initial construction cost for the data owner, and considerable verification burden for the clients. Motivated by these problems, we propose the MR-tree, an index based on the R*-tree [BKSS90], capable of authenticating arbitrary spatial queries. We show, analytically and experimentally, that the MR-tree outperforms the VR-tree significantly on all performance metrics.

Typically, successive queries from the same client focus on a small part of the data space (e.g., a moving client asking about its surroundings). Thus, the *VO*s of these queries have significant overlap. Our second contribution is a *synchronized caching* technique that utilizes this overlap in order to reduce the size of the *VO*. Elegant algorithms continuously update the cache contents of the LBS and the client, so that they are always identical and up-to-date, without requiring any additional communication overhead. Furthermore, the space overhead for the service provider is relatively small, so that a LBS with a realistic amount of main memory (1-2 Gbytes) can support synchronized caching for millions of clients.

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 focuses on the synchronized cache and its maintenance. Section 5 contains a comprehensive experimental evaluation, and Section 6 concludes the paper.

2. Related Work

Query authentication was first studied in the Cryptography literature. The *Merkle Hash Tree* (*MH-tree*) [M89] is a main-memory binary tree that hierarchically organizes hash[1] values. Figure 2.1 illustrates a MH-tree covering 8 data records d_1-d_8, each assigned to a leaf. A node N contains a hash value h_N computed as follows: if N is a leaf node, $h_N = H(d_N)$, and d_N is the assigned record of N, e.g., $h_1 = H(d_1)$; otherwise (N is an internal node), $h_N = H(h_{N.lc} \mid h_{N.rc})$, where $N.lc$ ($N.rc$) is the left (right) child of N respectively, and "|" concatenates two binary strings, e.g., $h_{1\text{-}4} = H(h_{1\text{-}2} \mid h_{3\text{-}4})$. After building the tree, the data owner signs the hash value h_{Root}, stored in the root of the MH-tree, using a *public key digital signature scheme* (e.g., RSA [MOV96]).

[1] Throughout the paper, the term *hash function* (*H*) implies a one-way, collision-resistant hash function. In this work we employ SHA1 [MOV96].

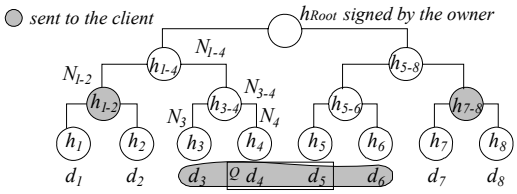


Figure 2.1 Example of Merkle Hash Tree

To authenticate one-dimensional range queries, Devanbu et al. [DGMS03] sort the database records on the query attribute and index them by a MH-tree. Figure 2.1 shows an example, where the DSP 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*, $N_{1\text{-}4}$, $N_{3\text{-}4}$, N_3) to the left boundary record d_3. For each node visited, the hash value ($h_{1\text{-}2}$) 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 hash values ($h_{7\text{-}8}$) 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 hash value at the root of the MHT using d_3, d_4, d_5, d_6 and the hash values in the *VO* ($h_{1\text{-}2}$, $h_{7\text{-}8}$): $h_{Root} = H(H\{h_{1\text{-}2} \mid H[H(d_3) \mid (H(d_4)]\} \mid H\{H[H(d_5)\mid (H(d_6)] \mid h_{7\text{-}8}\})$. If the reconstructed 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).

A combination of the MH-tree and the range search tree [BKOS97] is exploited in [DGMS03] to authenticate multi-dimensional range queries. Martel et al. [MND+04] extend the MH-tree concept to arbitrary search DAGs (Directed Acyclic Graphs), including dictionaries, tries, and optimized range search trees. Goodrich et al. [GTTC03] 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: $O(n\log^{d-1}n)$, where n and d are the size and dimensionality of the data respectively.

The first disk-based ADS in the Database literature is the *VB-tree* [PT04], which authenticates the soundness, but not the completeness, of 1D range results. A subsequent *signature chaining* approach [PJRT05, NT06] authenticates both soundness and completeness. Figure 2.2 illustrates an example, assuming that the database contains four tuples d_1-d_4, sorted on the search 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 service provider.

Let the result of a range query contain d_1, d_2 and d_3. The service provider 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 boundary records can be hidden through an encryption scheme [PJRT05].

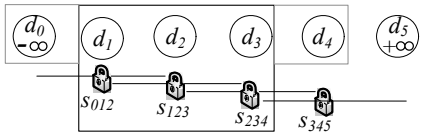


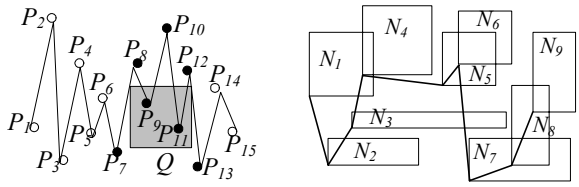
Figure 2.2 Example of signature chaining

The *Merkle B-tree* (MB-tree) [LHKR06] 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 hash values of the entries in N_c. Leaf nodes store records and their respective hash values. The DO signs the hash of the concatenation of the hashes contained in the root of the tree. Compared to signature chaining, the MB-tree incurs less space overhead since hash values are smaller than signatures and less verification effort because only the root is signed.

The only multi-dimensional ADSs in the database literature are the VKD-tree and VR-tree [CPT06]. These structures apply the signature chain concept to KD-trees [BKOS97] and R-trees [G84], respectively. We focus on the VR-tree since, as shown in [CPT06], 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 [PJRT05], the VR-tree creates one signature for each sequence of three points and stores it along with each entry, e.g., in Figure 2.3a, 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 Figure 2.3b, the signature of 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 Figure 2.3a, 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 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*.

[2] *Signature aggregation* [MNT04] condenses multiple signatures into a single one, thus significantly reducing the total size.



(a) Leaf Node (b) Internal Node

Figure 2.3 Signature chains in the VR-tree

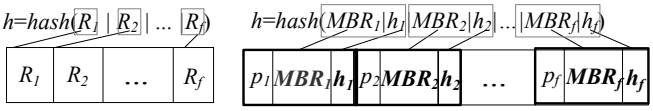
To verify results, the client starts from the root and compares all MBRs against the query. Then, it reads the content of each node whose MBR overlaps the query from the *VO* and recursively checks all its children. Finally, at the leaf level, it can extract the query results. During this procedure, the client incrementally constructs an aggregated digest from the MBRs and points included in the *VO*, which is eventually verified against the aggregated signature. As we show, analytically and experimentally, the VR-tree has some serious shortcomings: large space and query processing overhead, high initial construction cost, and considerable verification burden for the clients. The MR-tree, discussed next, aims at solving these problems.

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- [LHKR06] and R*-trees [BKSS90]. Figure 3.1 illustrates the node structure. Leaf nodes are identical to those of the R*-tree: each entry R_i corresponds to a data object. Note that although our examples use points, the MR-tree is applicable to objects with arbitrary shapes. A hash value 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 hash value of the ith child, respectively. The hash value summarizes child nodes' MBRs (MBR_1-MBR_f), in addition to their hash values (h_1-h_f). The hash value of the root node h_{root} is signed by the data owner and stored with the tree. The MR-tree supports updates based on the corresponding algorithms of the R*-tree. When a node changes (due to an insertion or deletion), the corresponding hash value in the parent entry is updated recursively, until reaching the root. The owner then signs the new root and transmits the changes to the LBS.



(a) Leaf Node (b) Internal Node

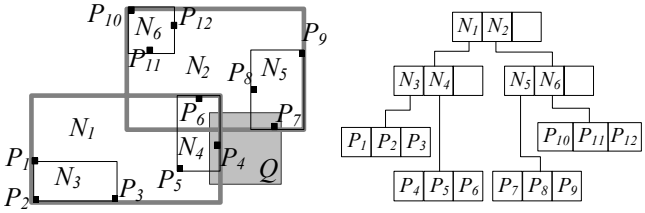
Figure 3.1 MR-tree node structure

To process a range query Q, the LBS invokes *RangeQuery*(*root*, Q), shown in Figure 3.2. 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 hash values of *pruned* nodes (Line 7), and (iii) special *tokens* [and] that mark the scope of a node (Lines 1 and 8). New entries are always *appended* to the end of the *VO*.

```
RangeQuery (Query Q, MR_Node N) // LBS
1. Append [ to VO
2. For each entry e in N // entries must be enumerated in original order
3.    If N is leaf
4.       Append e.data to VO
5.    Else // N is internal node
6.       If e.MBR overlaps Q,   RangeQuery(Q, e.pointer)
7.       Else append e.MBR, e.hash to VO // a pruned child node
8. Append ] to VO
```

Figure 3.2 Range query processing with the MR-tree

Consider, for instance, query Q in the example tree of Figure 3.3. Similar to conventional R-trees, *RangeQuery* starts from the root and visits recursively all entries that overlap the shaded rectangle: N_1, N_4, N_2, N_5. After termination, the verification object is: [[(MBR_N_3, hash_N_3), [P_4, P_5, P_6]]], [[P_7, P_8, P_9], (MBR_N_6, hash_N_6)]]. The tokens signify the contents of a node; for instance, the component [[(MBR_N_3, hash_N_3), [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*.



(a) Points and Node MBRs (b) MR-tree

Figure 3.3 Example range query

To verify the query results, the client first scans the *VO* to check that: (i) each data point in the *VO* is either outside Q, or included in the result set, (ii) no MBR (of a pruned node) in the *VO* overlaps Q, and (iii) the computed h_{root} from the *VO* agrees with s_{root}. Figure 3.4 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 hash values bottom-up. In the example of Figure 3.3, *RootHash* computes the MBR and hash value 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*. During the verification, the actual results (P_4, P_7) are extracted in Line 6. In addition, the client receives some objects (P_5, P_6, P_8, P_9) in the *VO*, which are not part of the result. Pang et. al. [PJRT05] propose a solution for avoiding disclosure of such objects, when the outsourced database must comply with certain access control policies. In this work, we consider that clients can issue queries freely without constraints. Nevertheless, the solution of [PJRT05] can be applied in conjunction with the proposed methods to hide the additional objects, if necessary.

```
(MBRValue, HashValue) RootHash(VO) // Client
1.  Initialize str , MBR to empty string and MBR value respectively
2.  While VO still has entries
3.     Get next entry e_V from VO
4.     If e_V is ], go to Line 13 // break the while-loop
5.     If e_V is a data object R
6.        If R overlaps the query, Add R to the result set
7.        MBR_c = the MBR of R
8.        str_c = the binary representation of R
9.     If e_V is [, (MBR_c, hash_c) = RootHash(VO)
10.    If e_V is a pair of MBR/hash value (MBR_e_V, hash_e_V)
11.       MBR_c, str_c = (MBR_e_V, hash_e_V)
12.    Enlarge MBR to include MBR_c
13.    Concatenate str with str_c
14. Return (MBR, hash (str))
```

Figure 3.4 Algorithm for re-computing h_{root}

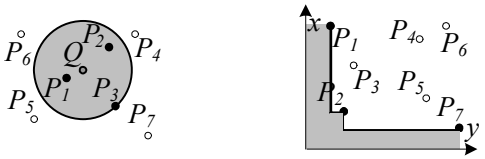
Proof of soundness: Assume that an object P in the result set is bogus or modified. Because the hash function is collision-resistant and P must be used by *RootHash*, the re-computed h_{root} can not be verified against s_{root}, which is detected by the client. □

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 extracted in Line 6 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* (*k*NN) and *skylines*. Given a point Q, a *k*NN query retrieves the k points from the data set that are closest to Q [HS99]. In the example of Figure 3.5a, 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 *k*NN of Q lie in a circular area C centered at Q that contains exactly k data points. Therefore, the LBS can prove the *k*NN 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 [PTFS05]. A point P_i dominates another P_j, if and only if, the co-ordinate of P_i

on each dimension is no larger than the corresponding co-ordinate of P_j. The skyline in Figure 3.5b 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 Figure 3.5b) 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.



(a) *k*NN (b) Skyline

Figure 3.5 Alternative queries

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 compare analytically the MR-tree and the VR-tree on the above metrics. Table 3.1 summarizes the symbols used in the analysis, as well as their typical values (1*msec* = 10^{-3} seconds, 1μ*sec* = 10^{-6} seconds). 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 [Crypto] and the values suggested in [LHKR06].

Symbol	Meaning	Typical Value	
C_s	CPU cost of *sign* operation	3.4 *m*sec	
C_v	CPU cost of *verify* operation	160 μsec	
C_h	CPU cost of *hash* operation	28 μsec	
C_m	CPU cost of *multiply* operation	43 μsec	
C_{NA}	CPU cost of a random node access	15 *m*sec	
S_s	size of a signature	128 bytes	
S_h	size of a hash value	20 bytes	
S_M	size of an MBR	32 bytes	
S_p	size of a data point	16 bytes	
n	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
h	height of the tree	VR 5	MR 4

Table 3.1 Symbols and values in the analysis

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:

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

where $r.l_j$ denotes rectangle r's extent along the jth dimension [PSTW03]. For simplicity, we assume that the data set contains points (rectangular data are discussed in [TS96]) 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 n be the data cardinality. The number of leaf nodes is n/f_l, and the height of the R-tree is $h = 1 + \lceil \log_{f_n}(n/f_l) \rceil$. 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 n/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 [TS96], 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 Equation 3.1, the total cost of processing Q using the VR- or the MR-tree is:

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

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:

$$S_{index} = b\left(\sum_{i=1}^{h-2} f_n^i + n/f_l\right) \qquad (3.3)$$

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 hash values (20 bytes each) only to internal nodes. Assuming a page of 4KBytes, 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 hash values 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 equation 3.4 (3.5):

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

$$C_{init}^{MR} = C_s + C_h \sum_{i=0}^{h-1} f_n^i \qquad (3.5)$$

Let the size of a signature, an MBR, a hash value 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:

$$S_{VO}^{VR} = S_s + \sum_{i=0}^{h-2} f_n^{i+1}\left(\sqrt[d]{1/f_n^i} + Q_l\right)^d S_M + n\left(\sqrt[d]{f_l/n} + Q_l\right)^d S_p \qquad (3.6)$$

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:

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

Thus, the total query processing overhead for the VR-tree is the sum of the two costs expressed in Equations 3.2 and 3.7. The *VO* size of the MR-tree is given by Equation 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.

$$S_{VO}^{MR} = \sum_{i=0}^{h-1} PN(i)\left(S_h + S_M\right) + n\left(\sqrt[d]{f_l/n} + Q_l\right)^d S_p$$
$$PN(i) = f_n^i\left(1 - \left(\sqrt[d]{1/f_n^i} + Q_l\right)^d\right) - \sum_{i=0}^{i-1} PN(j) f_n^{i-j} \qquad (3.8)$$

Finally we estimate the verification time for the client, which is dominated by modular multiplications (VR-tree) or computing hash values (MR-tree). The costs of the VR-tree (with signature aggregation) and the MR-tree are given by Equations 3.9-3.10. 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) $C_h < C_m$.

$$C_{Client}^{VR} = C_v + C_m\left(\sum_{i=0}^{h-2} f_n^{i+1}\left(\sqrt[d]{1/f_n^i} + Q_l\right)^d + n\left(\sqrt[d]{f_l/n} + Q_l\right)^d\right) \qquad (3.9)$$

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

Table 3.2 shows the costs calculated by the above equations using the typical values of Table 3.1. 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. Next, we present an optimization for reducing the *VO*.

Costs	MR-tree	VR-tree
Time for Computing Authentication Data	4 sec	2 hours
Index Size	57 MBytes	511 MBytes
Query Processing Time	2 sec	22 sec
VO size (bytes)	390 KBytes	398 KBytes
Verification (CPU time)	41 *ms*	991 *ms*

Table 3.2 Comparison of estimated costs

4. Synchronized Caching for the MR-tree

Each client is expected to issue numerous queries at different times. The *VO* of these queries always share common entries, specifically, s_{root} and the MBR/hash values of the root nodes (since the root is always accessed). In practice, the overlap is significantly larger because most queries focus on a small part of the data space. For instance, a moving client is likely to ask about its surroundings at successive locations that are close to each other. Assume that the client maintains the *VO* of previous queries in a cache. When the LBS processes a new query, it needs to send only the part of the *VO* that is not already in the cache. However, in order for this optimization to become possible, the LBS must have complete knowledge of the client's cache. We propose a *synchronized cache* (*SC*) scheme, where the LBS maintains, for each client, an *abstract* copy of its cache. The term abstract means that concrete hash values and records are replaced with *placeholders* (to be discussed shortly). As shown in the experimental evaluation, a LBS with a reasonable amount of main memory can support synchronized caching for millions of clients.

Figure 4.1 summarizes the proposed framework for synchronized caching. Given a query from the client, the LBS computes the (uncompressed) VO_{raw}. Then, it applies an algorithm (*ReduceVO*) that utilizes the contents of the *SC* to derive a compressed $VO_{reduced}$, i.e., the part of VO_{raw} that is not in *SC*. $VO_{reduced}$, which is usually much smaller than VO_{raw}, is sent to the client. The client restores VO_{raw} (using the reverse process of *ReduceVO*) and uses it to verify the query result. Both the LBS and the client incorporate the content of $VO_{reduced}$ to the *SC* through *MergeVO* algorithm. The addition of new content may increase the size of the *SC* beyond a predefined limit. In this case, *PurgeSC* frees space by expunging "old" data. It is easy to verify that if the LBS and the client start with an empty *SC* and have the same space limit, then their cache contents are identical at all times. Thus, there is no additional communication overhead for cache synchronization. On the other hand, this optimization minimizes the *VO* size and the associated transmission cost.

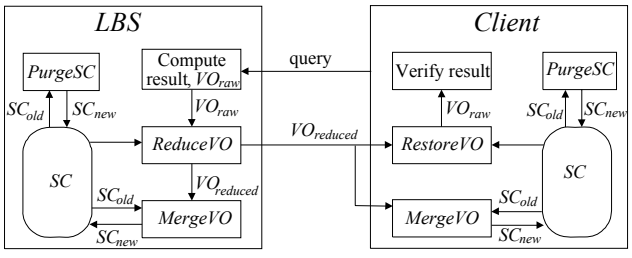


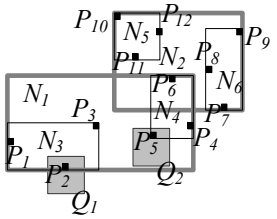
Figure 4.1 Framework of synchronized caching

Section 4.1 describes the *VO* minimization process (i.e., *ReduceVO*), while Section 4.2 focuses on the *SC* maintenance (i.e., *MergeVO* and *PurgeSC*). In our

discussion, we distinguish between *value* and *token* entries in the *VO*. A value entry is a data point, a pair of MBR/hash values, or the signature s_{root}. A token is [or].

4.1 Minimizing the Size of the *VO*

Similar to the *VO*, the *SC* is a linked list of value and token entries except that the copy maintained by the LBS uses a *placeholder* (i) for each (MBR, hash value) pair and (ii) for all records in a leaf node. Moreover, each token [is associated with a timestamp to be discussed later. We use the running example of Figure 4.2, where a client asks two queries Q_1, Q_2. The *SC* is initialized to be empty. When the first query is processed, its *VO* is copied to the caches of both the LBS and the client. After this step, the *SC* equals $VO(Q_1)$ = [[[P_1, P_2, P_3], (MBR_N_4, hash_N_4)], (MBR_N_2, hash_N_2)], s_{root}. When later the LBS processes Q_2, it compares $VO(Q_2)$ = [[(MBR_N_3, hash_N_3), [P_5, P_6]], (MBR_N_2, hash_N_2)], s_{root} with the *SC*. (MBR_N_2, hash_N_2) and s_{root} are in the *SC* and replaced with a token *SC_hit*, reducing the *VO* size from 5 to 3 value entries. Moreover, the entire sub-tree of N_3 (P_1-P_3) is in the *SC*, meaning that the client is able to compute MBR_N_3 and hash_N_3. Therefore, the LBS substitutes the entry (MBR_N_3, hash_N_3) with a token *SC_compute*, leading to a $VO(Q_2)$ with only 2 value entries P_5 and P_6.



Query	*VO*
Q_1	[[[P_1, P_2, P_3], (MBR_N_4, hash_N_4)], **(MBR_N_2, hash_N_2)], s_{root}**
Q_2	[[(MBR_N_3, hash_N_3), [P_5, P_6]], **(MBR_N_2, hash_N_2)], s_{root}**

Reduced $VO(Q_2)$: [[*SC_compute*, [P_5, P_6]], *SC_hit*], *SC_hit*

(*a*) Queries (*b*) *VO*s

Figure 4.2 Queries with overlapping *VO*s

Figure 4.3 shows *ReduceVO*, which utilizes the *SC* to minimize the verification object. Let VO_{raw} ($VO_{reduced}$) be the *VO* before (after) the shrinking process. *ReduceVO* scans the *SC* and VO_{raw} in parallel, computing $VO_{reduced}$. Each step retrieves an entry e_V (e_S) from VO_{raw} (*SC*). An important invariant is that e_V and e_S must always correspond to the same node (or data record) in the MR-tree. We illustrate the algorithm using the example of Figure 4.2 and assuming $SC = VO(Q_1)$ and $VO_{raw} = VO(Q_2)$. In the first two steps, e_S and e_V are both [(Case 4), and the LBS simply appends two [into $VO_{reduced}$. Then, e_S becomes [and e_V = (MBR_N_3, hash_N_3) (Case 2). Both e_S and e_V refer to the same node N_3: the *SC* contains *details* about N_3, whereas the *VO* only contains *aggregates* (i.e., MBR and hash). Therefore, it is possible to compute e_V with *SC* entries starting from e_S until its corresponding], i.e., [P_1, P_2, P_3]. Thus the LBS appends an *SC_compute* token to $VO_{reduced}$. Note that we must adjust the current entry of *SC* accordingly (Line 8-10) to ensure the invariant stated above. Next, e_V becomes [(before P_5) and e_S is (MBR_N_4, hash_N_4) (Case 3). Conversely to Case 2, now the VO_{raw} contains details (i.e., [P_5, P_6]) while the *SC* contains aggregates. Starting from this [, the LBS copies everything from VO_{raw} to $VO_{reduced}$, until the corresponding] is reached. Then, both e_V and e_S become successively [(Case 4), and (MBR_N_2, hash_N_2) (Case 1). The token *SC_hit* is appended to $VO_{reduced}$. Finally, for s_{root}, *SC_hit* is appended to $VO_{reduced}$. $VO_{reduced}$ is sent to the client, which *restores* the original VO_{raw} (following the reverse process of *ReduceVO*) and uses it to verify the query results.

```
VO_reduced ReduceVO (SC, VO_raw) // LBS
1.  Initialize VO_reduced to empty
2.  While VO_raw still has entries
3.    Get next entry e_V of VO_raw and e_S of SC
4.    If e_V and e_S are the same value entry                  // Case 1
5.      Append SC_hit to VO_reduced
6.    If e_V is a MBR/hash value pair and e_S is [             // Case 2
7.      Append SC_compute to VO_reduced
8.      Let e_begin = e_S
9.      While e_S is not the matching ] of e_begin
10.       Get next entry from SC as the new value for e_S
11.   If e_V is [ and e_S is a MBR/hash value pair             // Case 3
12.     Append e_V to VO_reduced
13.     Let e_begin = e_V
14.     While e_V is not the matching ] of e_begin
15.       Get next entry from VO as the new value for e_V
16.       Append e_V to VO_reduced
17.   If e_S and e_V are the same token entry                  // Case 4
18.     Append e_V to VO_reduced
```

Figure 4.3 *ReduceVO* algorithm

4.2 Updating the *SC*

Every new $VO_{reduced}$ updates the *SC* at the LBS and the client. Specifically, both LBS/client integrate $VO_{reduced}$ into the *SC* using *MergeVO*, shown in Figure 4.4. The *SC* before (after) this operation is called SC_{old} (SC_{new}). Initially, SC_{new} is empty. Each step of *MergeVO* retrieves pairs of entries $e_V \in VO_{reduced}$ and $e_S \in SC_{old}$ in parallel. Depending on the type of these entries, we have 4 cases, similar to *ReduceVO*. Case 1 occurs when e_V is a hit for e_S; e_S is added to SC_{new} and its [receives a timestamp equal to the current time. As we discuss shortly, timestamps are used to expunge old entries according to an LRU policy. Case 2 happens when e_V can be computed by e_S. *MergeVO* inserts to SC_{new} all entries between the [and] tokens of e_S. The recency of these entries is not updated, because *SC_compute* implies that only the aggregates, but not the actual contents, of e_S are required for the query. Case 3 incorporates new information from the *VO* into SC_{new}. Specifically, when the *VO* contains details of an MR-tree node while SC_{old} has only aggregates, we append these details into SC_{new}. In the example of Figure 4.2, if the client has $SC_{old} = VO(Q_1)$ and receives the reduced $VO(Q_2)$, *MergeVO* updates the timestamps and replaces the

MBR/hash value of N_4 with [P_5, P_6]. Conceptually, SC_{new} becomes the *VO* for query $Q = (Q_1 \text{ or } Q_2)$. Case 4 simply appends token entries.

```
SC_new MergeVO (SC_old, VO) // VO is already reduced
1.  Initialize SC_new to empty
2.  While VO has entries
3.    Get next entry e_V of VO and e_S of SC
4.    If e_V is SC_hit                                  // Case 1
5.      Append e_S to SC_new
6.      Set the timestamp of the [ of e_S to now
7.    If e_V is SC_compute                              // Case 2
8.      Append e_S to SC_new
9.      Let e_begin = e_S
10.     While e_S is not the matching ] of e_begin
11.       Get next entry from SC_old as the new value for e_S
12.       Append e_S to SC_new
13.   If e_V is [ and e_S is an MBR/hash value pair     // Case 3
14.     Let e_begin = e_V
15.     While e_V is not the matching ] of e_begin
16.       Get next entry from SC_old as the new value for e_S
17.   If e_S and e_V are the same token entry           // Case 4
18.     Append e_S to SC_new
```

Figure 4.4 *MergeVO* algorithm

In our implementation, we assign a limit L to the size of the *SC* at the client side. L may depend on the memory of the client, or it may be decided by the LBS. In either case, the LBS and the client agree on the value of L, which may be different for each client (e.g., the LBS may charge clients according to their cache size). If the *SC* exceeds L (after an application of *MergeVO*), *PurgeSC* (Figure 4.5) removes the oldest entries to free space. Specifically, *PurgeSC* performs the opposite of operation *MergeVO*, i.e., it replaces the details of an MR-node (a sequence of entries bounded by [and]) with a single entry that contains the MBR and hash value of the node. The process is applied repeatedly until the size of the *SC* drops below L. At each step, the node to be replaced is chosen according to an LRU policy based on the timestamp stored with each [. Recall that these timestamps are maintained by *MergeVO*.

```
PurgeSC (SC)
1. While the size of the SC exceeds the limit L
2.    Scan SC to find the oldest [ that is not enclosed by other tokens.
      Let e_begin be this [
3.    Let e_end be the corresponding ] of e_begin
4.    Compute the MBR and hash of all SC entries from e_begin to e_end
5.    Replace all SC entries from e_begin to e_end with a single entry
      (MBR, hash)
```

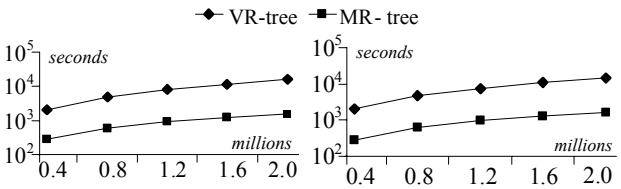
Figure 4.5 *PurgeSC* algorithm

5. Experimental Evaluation

We implemented the MR-tree and the VR-tree in C++, using the Crypto++ library [Crypto] and executed all experiments on a P4 3GHz CPU. Both MR-tree and VR-tree implementations are based on R*-trees using 4Kbytes page size. Each experiment is 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 [R-portal]. In cases where we want to set a specific cardinality, we randomly sample from these datasets using an appropriate sampling rate. Section 5.1 compares the initial construction cost and size of MR-trees and VR-trees. Section 5.2 evaluates the query processing and verification overhead of the two structures. Section 5.3 assesses the benefits of synchronized caching.

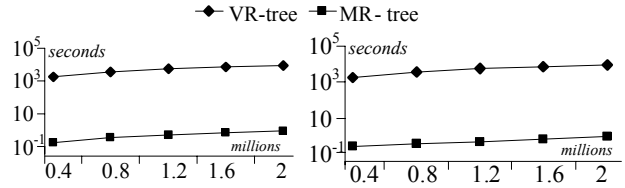
5.1 Initial Construction

Figure 5.1 illustrates the construction cost for VR- and 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 hash values (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. Figure 5.2 shows the CPU time for computing the necessary authentication information. The MR-tree outperforms the VR-tree by 3-4 orders of magnitude on this metric. Comparing Figures 5.2 and 5.1, 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. Figure 5.3 illustrates the size of the indexes in MBytes. The VR-tree is much larger since it stores one signature (128 bytes) for each data point and node, where the MR-tree stores one digest (20 bytes) for every node.



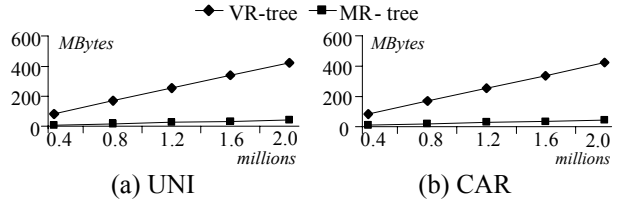
(a) UNI (b) CAR

Figure 5.1 Total construction time vs. data cardinality



(a) UNI (b) CAR

Figure 5.2 CPU time for authentication data vs. cardinality



(a) UNI (b) CAR

Figure 5.3 Index size vs. data cardinality

5.2 Query Processing and Verification

This section evaluates the query and verification cost of the two structures. All queries are ranges (recall from Section 3, that other query types, such as NN, can be converted to ranges), covering 1% of the entire (2D) space. For every experiment, we execute 100 ranges at random locations and illustrate the average cost. This cost burdens the LBS and includes both the result retrieval and the construction of the verification object. The data cardinality (N) varies between $0.4 \cdot 10^6$ and $2 \cdot 10^6$. We do not include synchronized caching since it is evaluated separately in Section 5.3.

Figure 5.4 illustrates the query cost (in seconds) as a function of the data cardinality (Q_l = 10%). The MR-tree is fast because it creates the *VO* by simply appending MBRs and hash values 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.

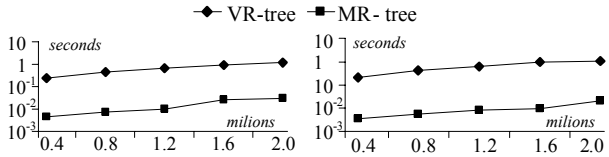


Figure 5.4 Query cost vs. data cardinality

Next we measure the verification object. Figure 5.5 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 verification object (of VR-trees and MR-trees) is larger than the corresponding result, because the result is always part of the *VO*.

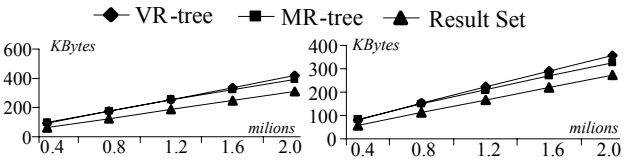


Figure 5.5 *VO* size vs. data cardinality

Finally, Figure 5.6 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.

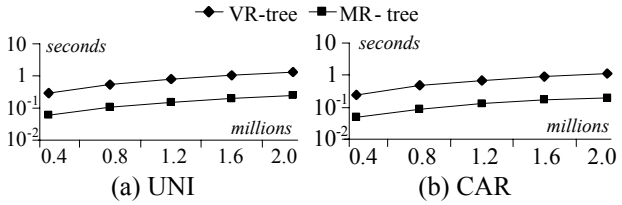


Figure 5.6 Verification time vs. data cardinality

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 have similar performance is the size of the *VO*. Next, we evaluate the impact of synchronized caching on the verification object.

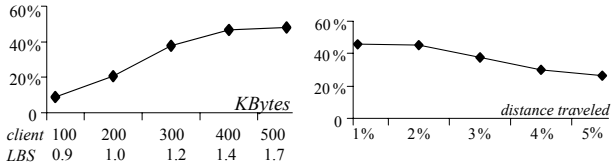
5.3 Synchronized Caching

Recall that synchronized caching entails two caches at the LBS and the client. The *VO* of each processed query is incorporated in the caches and utilized to reduce the *VO* of subsequent queries. This optimization is expected to have considerable benefits in cases where successive queries exhibit locality. In our experiments we simulate a moving client that enquires about its surroundings. Specifically, the first query is a range centered at a random location. The user chooses a direction, moves a certain distance and issues another range search (with a fixed extent). The process is repeated 100 times. The *VO*s of the first 50 queries are used to warm up the cache. For the remaining ones we measure the *average reduction* achieved per *VO*. The average reduction is defined as $(|VO_{raw}|-|VO_{reduced}|)/|VO_{raw}|$, where $|VO_{reduced}|$ ($|VO_{raw}|$) is the size of the *VO* with (without) synchronized caching.

We investigate the effect of the cache size, and the distance traveled between two consecutive queries using the CAR dataset. The results for UNI are similar and omitted due to the lack of space. The data cardinality is set to $2 \cdot 10^6$, and the query extent to 10% per axis. Figure 5.7a illustrates the average *VO* reduction as a function of the cache size (at the client), after fixing the distance between two consecutive queries to 3% of the axis length. Even 100 KBytes of cache result in a reduction of about 10%. The reduction increases with the cache size and stabilizes at around 500 KBytes. After this point, more cache does not have a significant impact on performance, because the new parts of the tree have to be included in the *VO* anyway.

The LBS stores, for each client, placeholders for the corresponding hash values and records. Therefore, the cache copy at the LBS consumes much less space than that of the client. The memory consumption per client at the LBS is shown at the bottom of the x-axis. Assuming a cache of 500Kbytes per client (i.e., 1.7Kbytes at the LBS), a LBS with 1 Gbyte of main memory can support up to

$558 \cdot 10^3$ clients. If the cache size is 300 Kbytes per client, the LBS can support $833 \cdot 10^3$ clients.



(a) vs. cache size (b) vs. distance traveled

Figure 5.7 Average *VO* reduction (CAR)

Figure 5.7b illustrates the average reduction as a function of the distance between successive queries, after setting the cache size to 300 KBytes. As expected, the effect of the cache diminishes with the increasing distance, since the stored *VO* becomes irrelevant faster. Nevertheless, the distances that we use in these experiments are rather large compared to the locality exhibited in most practical applications.

In conclusion, synchronized caching achieves significant reduction of the *VO* size, even for small caches and relatively infrequent (or distant) queries. Most potential clients of spatial outsourcing systems (e.g., PDAs) already include flash memory that reaches several Mbytes and could devote part of this memory for caching purposes. The minimization of the verification object, on the other hand, leads to savings in the communication cost, which is very important for wireless networks. Finally, recall that the update algorithms of Section 4 eliminate the need to transfer cache information between the LBS and the client.

6. Conclusion

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 multidimensional 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 orders of magnitude in many important metrics such as construction cost, index size and verification overhead. Furthermore, we develop a novel synchronized caching protocol, which significantly reduces the communication overhead of the verification step. We conclude our contributions with an extensive experimental study that validates the effectiveness and efficiency of the proposed structure.

References

[BKOS97] de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O. Computational Geometry: Algorithms and Applications. Springer-Verlag, 1997.

[BKSS90] Beckmann, N., Kriegel, H.-P., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD*, 1990.

[CPT06] Cheng, W., Pang, H., Tan, K.-L. Authenticating Multi-Dimensional Query Results in Data Publishing. *DBSec*, 2006.

[Crypto] www.eskimo.com/~weidai/benchmark.html

[DGMS03] Devanbu, P., Gertz, M., Martel, C., Stubblebine, S. Authentic Data Publication Over the Internet. *Journal of Computer Security* 11(3): 291-314, 2003.

[G84] Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD*, 1984.

[GTTC03] Goodrich M., Tamassia R., Triandopoulos N., Cohen R. Authenticated Data Structures for Graph and Geometric Searching. *CT-RSA*, 2003.

[HIM02] Hacıgümüş, H., Iyer, B., Mehrotra, S. Providing Databases as a Service. *ICDE*, 2002.

[HS99] Hjaltason, G., Samet, H. Distance Browsing in Spatial Databases. *ACM TODS*, 24(2):265-318, 1999.

[LHKR06] Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L. Dynamic Authenticated Index Structures for Outsourced Databases. *SIGMOD*, 2006.

[M89] Merkle, R. A Certified Digital Signature. *CRYPTO*, 1989.

[MND+04] 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.

[MNT04] Mykletun, E., Narasimha, M., Tsudik, G. Signature Bouquets: Immutability for Aggregated/Condensed Signatures. *ESORICS*, 2004.

[MOV96] Menezes, A., van Oorschot, P., Vanstone, S. Handbook of Applied Cryptography. CRC Press, 1996.

[NT06] Narasimha M., Tsudik G. Authentication of Outsourced Databases Using Signature Aggregation and Chaining. *DASFAA*, 2006.

[PJRT05] Pang, H., Jain, A., Ramamritham, K., Tan, K.-L. Verifying Completeness of Relational Query Results in Data Publishing. *SIGMOD*, 2005.

[PSTW93] Pagel, B., Six, H., Toben, H., Widmayer, P. Towards an Analysis of Range Query Performance in Spatial Data Structures. *PODS*, 1993.

[PTFS05] Papadias, D., Tao, Y., Fu, G., Seeger, B. Progressive Skyline Computation in Database Systems. *TODS* 30(1), 41-82, 2005.

[PT04] Pang, H., Tan, K.-L. Authenticating Query Results in Edge Computing. *ICDE*, 2004.

[R-portal] www.rtreeportal.org

[TS96] Theodoridis, Y., Sellis, T. A Model for the Prediction of R-tree Performance. *PODS*, 1996.