Continuous Spatial Authentication

Stavros Papadopoulos[1], Yin Yang[1], Spiridon Bakiras[2], and Dimitris Papadias[1]

[1] Dept. of Computer Science and Engineering,
Hong Kong University of Science and Technology
{stavros, yini, dimitris}@cse.ust.hk
[2] Dept. of Mathematics and Computer Science,
John Jay College, City University of New York
sbakiras@jjay.cuny.edu

Abstract. Recent advances in wireless communications and positioning devices have generated a tremendous amount of interest in the continuous monitoring of spatial queries. However, such applications can incur a heavy burden on the *data owner* (DO), due to very frequent location updates. Database outsourcing is a viable solution, whereby the DO delegates its database functionality to a *service provider* (SP) that has the infrastructure and resources to handle the high workload. In this framework, *authenticated query processing* enables the clients to verify the correctness of the query results that are returned by the SP. In addition to correctness, the dynamic nature of the monitored data requires the provision for *temporal completeness*, i.e., the clients must be able to verify that there are no missing results in between data updates. This paper constitutes the first work that deals with the authentication of continuous spatial queries, focusing on ranges. We first introduce a baseline solution (BSL) that achieves correctness and temporal completeness, but incurs false transmissions; that is, the SP has to notify clients whenever there is a data update, even if it does not affect their results. Then, we propose CSA, a mechanism that minimizes the processing and transmission overhead through an elaborate indexing scheme and a virtual caching mechanism. Finally, we derive analytical models to optimize the performance of our methods, and evaluate their effectiveness through extensive experiments.

1 Introduction

In *database outsourcing* [9], a *data owner* (DO) delegates its DBMS tasks to a *service provider* (SP) that has the necessary resources to perform advanced query processing. The SP is then responsible for processing client queries on behalf of the DO. *Authenticated query processing* allows the SP to prove to the client that (i) the results are *authentic* (i.e., originated from the DO), (ii) *sound* (i.e., no result object is fictitious or modified), and (iii) *complete* (i.e., all objects satisfying the query are present). We refer to these three terms collectively as *correctness*. Figure 1.1 illustrates the general framework, commonly used in the outsourcing literature. Initially, the DO obtains, through a trusted key distribution center, a *private* and a *public* key [20]. The private key is known only to the DO, while the public key is accessible by all the clients. The DO signs the data with its private key, generating one (or more) signatures. Then, it sends the signature(s) and the data to the SP, which constructs an *authenticated data structure* (ADS) for efficient query processing. The ADS is essentially an index that contains additional authentication information (typically, hash digests and signatures). When the SP receives a query from a client, it generates a *verification object* (*VO*) by accessing the ADS. The *VO* contains the result set along with the necessary authentication information. The SP sends the *VO* to the client, which can verify the results by matching the *VO* against the public key of the DO.

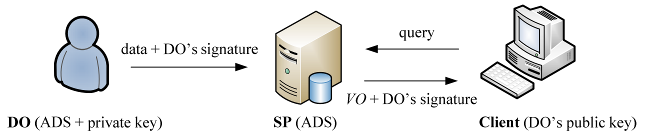


Fig. 1.1: Database outsourcing framework.

While there is extensive literature on authenticated processing in conventional databases, there is very limited work on outsourced data in the presence of frequent updates, especially for spatio-temporal databases. In this paper, we focus on authenticated processing of continuous spatial ranges, motivated by advances in wireless communications and GPS-enabled devices (e.g., RFID chips, sensor networks, navigation systems, etc.). Consider, for instance, a SP that receives locations of shipments around the globe (using RFID technology). A company (i.e., a client) that wishes to track its products through the SP registers long-running queries that monitor certain locations of interest. Whenever an update (arrival or departure) influences a query, the corresponding client is immediately informed. In addition to the timely delivery of query results, it is crucial for the subscribers of such a system to be able to verify their correctness.

The dynamic nature of the data in the above scenario, and the potentially large number of long-running queries, pose several technical challenges. First, a system for continuous authentication on dynamic data must accommodate very fast updates and also support efficient query processing. Second, it must provide effective mechanisms for minimizing the communication cost with the clients, and their verification effort. Third, in addition to correctness, the clients must be able to verify the *temporal completeness* of their results, i.e., confirm that they receive all the updates that affect their queries.

This paper constitutes the first work on continuous authentication of dynamic spatial data. We first introduce a *baseline solution*, called BSL, that achieves correctness and temporal completeness, but incurs false transmissions; that is, the SP has to notify clients whenever there is a data update, even if it does not affect their results. Then, we propose CSA (for *continuous spatial authentication*), a mechanism that minimizes the processing and transmission overhead through an intricate indexing scheme and a virtual caching mechanism. Third, we derive accurate models for estimating the size of the *VO*, which is the most important factor that determines the performance of an outsourcing system. We apply these models to optimize CSA. Finally, we empirically show that CSA outperforms BSL significantly in all aspects and is, therefore, applicable in highly dynamic environments. The remainder of the paper is organized as follows. Section 2 reviews existing ADSs for database outsourcing. Section 3 describes BSL, while Section 4 proposes the CSA technique. Section 5 presents our experimental results and Section 6 concludes the paper.

2 Related Work

The *Merkle Hash Tree* (MH-Tree) [12] is a main-memory binary tree that provides efficient authentication of equality queries on single-dimensional data. It assumes that the database is sorted on the query attribute and, at the leaf level, every node stores the digest of the binary representation of the record. The digests are computed with a *one-way, collision-resistant hash function* (e.g., SHA1 [15]). The tree is built bottom-up and internal nodes store the digest of the concatenation of the digests of their children. After the tree is constructed, the DO signs the digest stored in the root of the tree and sends it, along with the data, to the SP. During query processing, the SP traverses the tree and, apart from the requested record, it inserts into the *VO* the digest stored in the sibling of every visited node. Having the *VO*, the DO's signature and the DO's public key, the client can verify the authenticity of the result by re-constructing the digest of the root. Devanbu et al. [5] modify the query processing mechanism of the MH-Tree for answering one-dimensional range queries, while satisfying soundness and completeness. They also extend their methods to multiple dimensions, combining the MH-Tree with the *Range Search Tree* [2].

The *Verifiable B-Tree* (VB-Tree) [18] is the first *signature-based* approach that augments a standard B^+-Tree with signed digests. However, this method only guarantees the correctness of the results, but not the completeness. To address this problem, Pang et al. [17] and Narasimha and Tsudik [16] introduce a technique called *signature chaining*. They assume that the dataset is sorted on one attribute, and every record is associated with one signature. This signature combines hashed information about the record, and both its immediate successor and predecessor in the sorted order. In addition, the DO inserts two special (boundary) records at the two ends of the sorted order. To assure integrity for a range query, the constructed *VO* contains (i) the result set, (ii) the signature for each record in the result set, and (iii) the boundary records.

The *Merkle B-Tree* (MB-Tree) [10] extends the MH-Tree to external memory (the node fanout depends on the disk page size). Every node has a digest, which is computed by applying the hash function to the concatenation of all its children's digests. The DO then signs the hash of the concatenation of the digests of the entries contained in the root node of the tree. Range query processing is performed by two top-down traversals (one for each boundary record). At each visited node, the digests of the nodes that do not overlap the query range are inserted into the *VO* (along with the result set and the signed root).

In the context of multi-dimensional databases, which is closely related to this work, there have been very few ADSs proposed in the literature. First, Cheng et al. [3] introduce two authenticated structures, namely the *Verifiable KD-Tree* (VKD-Tree) and the *Verifiable R-Tree* (VR-Tree). Both structures are modified versions of the standard KD-Tree and R-Tree, respectively. Specifically, in every node of the tree, the points and/or MBRs (Minimum Bounding Rectangles) contained therein are sorted according to their *x*-coordinate, and a signature chain is generated. Range queries are processed by following the signature chain at every node that overlaps the query range. However, these structures incur large space and query processing overhead for the SP, high initial construction cost for the DO, and considerable verification burden for the clients. Furthermore, they lack algorithms for insertions and deletions (updates are not discussed in [3]), which render them inapplicable to dynamic environments.

Currently, the most efficient ADS for multi-dimensional databases is the *Merkle R-Tree* (MR-Tree) [21], which combines the idea of the MB-Tree with the structure of the R*-tree. In particular, every leaf node is associated with a digest that is computed on the concatenation of the binary representation of all objects in the node. Internal nodes are assigned a digest that summarizes the child nodes' MBRs and digests. Each node digest is stored at the corresponding parent entry. The root digest is signed by the DO. Range queries are handled by a depth-first traversal of the tree. The resulting *VO* contains (i) all the points in every leaf node visited, (ii) the MBRs and digests of all the pruned nodes, and (iii) the DO's signature. Nevertheless, the MR-Tree cannot support very fast updates and is, thus, not suitable for our problem.

Also related to our work are two recent solutions that authenticate continuous range queries on one-dimensional data streams. First, Li et al. [11] deal with authentication in sliding window streams, i.e., a tuple expires *w* time units after its arrival. Their method segments the time into slots, and builds a separate MH-Tree on the tuples that arrived in each slot. Its goal is to reduce the communication cost at the expense of delayed result updates. Papadopoulos et al. [19] introduce CADS, which also deals with streaming environments, but focuses on real-time reporting. CADS combines space (i.e., domain) partitioning with MH-Trees for effective indexing.

Our work extends the general methodology of [19] for continuous spatial ranges. Specifically, we integrate space partitioning, MH-Trees and Hilbert curves for indexing highly dynamic spatial data. In addition to data structures, we develop a comprehensive set of algorithms for the initial computation and the continuous monitoring of the results. Finally, we propose accurate models for determining the best space partitioning granularity, a factor that significantly affects the scheme effectiveness in our setting.

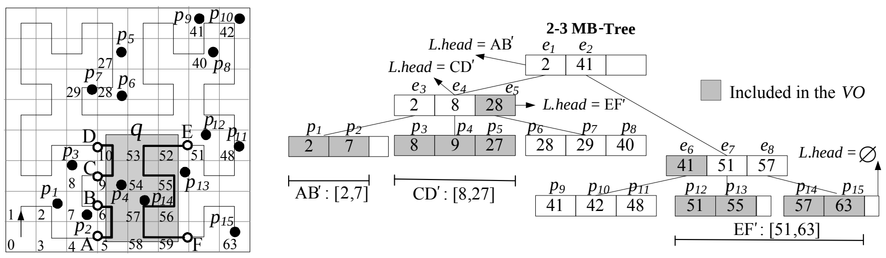
3 Baseline Solution

Since there is no spatial ADS that can handle frequent location updates, in this section we devise a baseline solution (BSL). Each point[1] *p* is represented by a tuple of the form <*p.id*, *p.x*, *p.y*>, where *p.id* is a unique identifier and (*p.x*, *p.y*) are *p*'s coordinates. BSL maps all the 2D points into the 1D domain utilizing a space-filling curve. We employ the Hilbert curve because it preserves spatial locality and leads to low query processing cost [13]. Let D:[L_x, L_y, U_x, U_y] be a square dataspace, where (L_x, L_y) and (U_x, U_y) are the lower left and upper right corners. D is partitioned in $2^{2\cdot o}$ regular cells, where *o* is an integer called the *order* (or *resolution*). Figure 3.1a depicts a Hilbert curve with *o*=3. The cell at the lower left corner has Hilbert value 0, and the

[1] We use terms *point* and *object* interchangeably.

values of the remaining cells follow the Hilbert curve (for simplicity, we only include the values of selected cells). Each point p is associated with the Hilbert value $p.hv$ of the cell that covers it, e.g., $p_1.hv$=2, $p_2.hv$=7, $p_3.hv$=8, etc.

The DO indexes the points with a 2-3 MB-Tree using their Hilbert values as search keys. The 2-3 MB-Tree is similar to a main memory MB-Tree, where each node may have either 2 or 3 entries. An insertion in a full node causes it to split in two nodes, each containing 2 entries. On the other hand, a deletion from a node with 2 entries leads to an underflow. Similarly to B$^+$-Trees, the node first tries to *borrow* an entry from a full sibling node. If this is not possible, the node is *merged* with a sibling. To support multiple updates at the same timestamp, we do not alter any digest, but temporarily mark the visited paths. Then, the marked paths are revisited and the digests are computed bottom-up. In this way, the (expensive) hash computations are performed only once.



(a) Hilbert curve (o=3) (b) 2-3 MB-Tree in BSL

Fig. 3.1: Indexing and query processing in BSL

Each leaf entry p in the 2-3 MB-Tree has the form <$p.id$, $p.x$, $p.y$, $p.hv$>. An intermediate entry e is a triplet ($e.h$, $e.k$, $e.ptr$), where $e.k$ is the Hilbert value of the first point in the subtree of e, $e.ptr$ is a pointer to the corresponding child node and $e.h$ is a digest computed on the concatenation of the digests of the entries in $e.ptr$. Figure 3.1b contains the tree for the points of Figure 3.1a, showing only the Hilbert value of each entry. The DO computes a signature on H_{root}, D and o, i.e., it performs $sign(h(H_{root} \mid L_x \mid U_x \mid L_y \mid U_y \mid o))$, where h is the hash function (in our work we employ SHA1 [15]) and '|' denotes concatenation. As shown later, D and o are necessary during the verification process. Then, it sends the tree, D, o and the signature to the SP. Upon receiving a 2D window query q, the SP finds the parts of the Hilbert curve corresponding to cells that intersect with q. Given the shaded query in Figure 3.1, (i) poly-line AB corresponds to cells 5 and 6, (ii) CD to 9 and 10, and (iii) EF to cells 52-59. The union of points in these cells constitutes the result of q; i.e., the result is $\{\}\cup\{p_4\}\cup\{p_{13}, p_{14}\}$ for points in AB, CD, EF, respectively. Note that the result may contain some false positives, e.g., p_{13}, that fall out of the query window but reside in an intersecting cell. Such points are filtered out by the client.

Each poly-line corresponds to a 1D range in the 2-3 MB-Tree. One solution would be for the SP to process these ranges one by one. This involves an expansion of each range to include boundary records. For instance, AB is extended to AB′:[2,7] so that it covers p_1 and p_2. Similarly, CD and EF are extended to CD′:[8,27] and EF′:[51,63] to include boundary tuples p_3, p_5 and p_{12}, p_{15}, respectively. Finally, the SP should construct a separate *VO* for each of the expanded ranges. However, executing the 1D ranges individually and generating separate *VO*s would be inefficient, because (i) tree nodes may be visited multiple times, and (ii) *VO* components (i.e., digests and/or boundary points) may either appear in several *VO*s, or they may not be necessary as they can be reconstructed from information contained in other *VO*s. To avoid these problems, we integrate the execution of all sub-queries in one traversal that produces a single *VO*. The generated *VO* has no redundancy and can be verified efficiently by a linear scan. The detailed algorithm, called *MultiRangeMB*, is shown in Figure 3.2. Note that the algorithm utilizes special tokens [and] that indicate the scope of a node.

```
MultiRangeMB (MBNode n, List L)
1.  Append [ to the VO
2.  For each entry e in n
3.     If n is an intermediate node
4.        if e intersects L.head // e may contain results
5.           MultiRangeMB(e.ptr, L) // e.ptr points to a child node
6.        Else append e.h to the VO
7.     Else // n is a leaf node and e is a point
8.        Append <e.id, e.x, e.y> to the VO
9.        If e is the last entry of n AND e.hv is ≥ the upper bound of L.head
10.          Evict L.head from L
11. Append ] to the VO
```

Fig. 3.2: Algorithm *MultiRangeMB* of BSL

MultiRangeMB takes as arguments the root of the 2-3 MB-Tree, and a list *L* that stores the 1D (Hilbert) sub-ranges of query *q* sorted in ascending order. The algorithm traverses the tree in a depth-first fashion, and checks all the entries contained in each visited node. For an intermediate node, if an entry *e* overlaps with the head of *L* (*L.head*), the traversal continues in *e*'s subtree (lines 4-5). Otherwise, the digest of *e* is inserted into the *VO* (line 6). Line 6 also captures the case where *L* is empty, so that the digests of all unexamined entries along the path from the last leaf visited up to the root are appended to the *VO*. When the algorithm reaches a leaf node, it first appends all point entries to the *VO* (line 8). If the Hilbert value of the last point contained in the leaf is greater than or equal to the upper bound of *L.head* (i.e., the boundary point entry for *L.head* is already inserted in the *VO*), the latter is evicted from *L*.

We illustrate this multi-range traversal using the example of Figure 3.1. The SP sorts the expanded ranges AB′, CD′ and EF′ in ascending order of their lower boundary value and inserts them in the ordered list *L*. Then, it starts by processing range AB′:[2,7] at the head of *L*. Every point (p_1, p_2) satisfying AB is appended to the *VO* (such entries are shown in grey). After the leaf accommodating the last point (p_2) is visited, AB′ is evicted from *L*. Subsequently, the algorithm continues at entry e_4, where it starts processing CD′:[8,27] and includes p_3, p_4, p_5 in the *VO*. CD′ is evicted from *L*, EF′:[51,63] commences at e_5 and $e_5.h$, $e_6.h$, p_{12}, p_{13}, p_{14}, p_{15} are appended to the *VO*. Note that it is not necessary to include *p.hv* in the *VO* because, along with the *VO* and the signature, the SP sends *D* and *o*. Having this information available, the client can compute the Hilbert values of the points locally.

In general, the *VO* contains a sequence of point entries for each processed 1D interval, and (possibly) digests interleaved between pairs of point sequences. Upon receiving the *VO*, the client first decomposes *q* into the same set of 1D intervals as the SP (before their expansion) using *D* and *o*, sorts them on their lower boundary value and inserts them in a list *L*. Then, it utilizes an algorithm to reconstruct the digest of the root (H_{root}). This algorithm is similar to the evaluation of parenthesized arithmetic expressions, where the tokens play the role of the parentheses. When the algorithm encounters a token], it has all the information (digests or records) to compute the digest of the node that started at the corresponding [. The digests and records are appended to a buffer *B*, which after termination is used to derive $H_{root}=h(B)$. Furthermore, for each encountered point sequence, the algorithm computes the Hilbert values of the points, checks whether the boundary points for *L.head* exist, and evicts the latter from *L*. Also it reports the points that satisfy *q* during this process. At the end of the algorithm, *L* must be empty and the reconstructed H_{root} combined with *D* and *o* must verify the signature.

The proof of soundness is straightforward, since if the SP modifies any *VO* component, the signature will not be verified (due to the collision-resistance of the hash function). Furthermore, recall that the client receives *D* and *o* intact (otherwise the signature verification fails) and, therefore, it can determine the exact 1D queries processed by the SP. It can then ensure completeness by simply checking the existence of the boundary objects for *each* sub-query.

The above discussion captures the initial result computation in BSL; next we describe the continuous monitoring component. Whenever there is a data modification, the DO alters its tree and forwards the update(s) to the SP, according to the *positive-negative* model. An object insertion is denoted as (+<*p.id*, *p.x*, *p.y*, *p.hv*>), and a deletion as (-<*p.id*, *p.hv*>). An object movement is handled by a deletion followed by an insertion. In addition to the actual data, each transmission contains a DO signature and two timestamps: *LT* is the current time and *ST* is the time of the previous transmission. The signature incorporates the new H_{root}, *LT* and *ST*. The two timestamps are necessary so that the clients can detect temporal attacks, i.e., situations where the SP avoids reporting some result updates. Specifically, an authentication scheme satisfies *temporal completeness*, if it is impossible for the SP to omit sending a result change to the client, without the latter detecting it [19]. The DO periodically sends updates to the SP, along with the new signature, *LT* and *ST*. The SP updates the tree structure, the timestamps and the signature accordingly, and generates a new *VO* for *every* monitored query. It then sends the new *VO*, *LT*, *ST* and the signature to the corresponding clients (*D* and *o* do not need to be re-sent).

Proof of temporal completeness (sketch): Suppose that at time *t* the SP avoids sending the *VO* for an update that affects the client's result. At a later time *t'* the SP transmits a new *VO* to the client. Note that multiple omissions may have occurred between *t* and *t'*. The client will detect the attack by noticing that the time of the previous update (included in the new *VO*) is $ST \geq t$, at which it did not receive any *VO*. Note, however, that temporal completeness cannot be guaranteed if the client does not receive any *VO* for a long time, in which case it cannot be sure whether the last results are still up-to-date. This problem can be solved using the concept of *query freshness* [10], according to which the DO revokes old signatures at periodic time intervals. □

The efficient query processing and update operations of the 2-3 MB-Tree render BSL suitable for dynamic environments. However, BSL incurs *false transmissions* of *VO*s for queries whose result is not affected by the latest data updates. This imposes significant CPU cost to the SP (for computing the *VO*s) and to the clients (for verifying them). Furthermore, it leads to excessive network overhead. The next method aleviates these problems by integrating sophisticated indexing schemes and query processing algorithms.

4 Continuous Spatial Authentication – CSA

Section 4.1 describes the indexing scheme of CSA and Section 4.2 explains the query processing algorithms. Section 4.3 presents the analytical models used to optimize the performance of CSA.

4.1 Indexing Scheme

CSA subdivides the dataspace into partitions, in order to reduce the area affected by an update and limit the number of false transmissions. Let D:[L_x, L_y, U_x, U_y] be a square dataspace. We build an $m \cdot m$ regular grid over D, by decomposing each axis into m equal segments. Let l_P be the extent of each partition along the two axes. A point p with co-ordinates ($p.x$, $p.y$) can be located in constant time in partition P_{ij}, where $i=\lfloor p.x \,/\, l_P \rfloor$ and $j=\lfloor p.y \,/\, l_P \rfloor$. In order to capture skewed point distributions, we embed a *Temporal Merkle Hash-Tree* (TMH-Tree) in each partition P. The TMH-Tree is a modified 2-3 MB-Tree that incorporates temporal information used by a virtual caching mechanism. Specifically, every entry e in an intermediate node contains a timestamp $e.t$ that signifies the latest (i) record insertion/deletion/update that occurred in the subtree of e, or (ii) movement of e to another node due to a split/merge operation. Figure 4.1 summarizes the index structures in CSA.

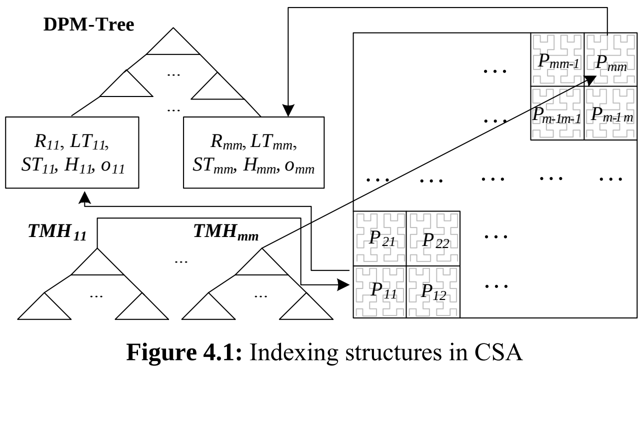


Figure 4.1: Indexing structures in CSA

For each partition, we construct a Hilbert curve of order $P.o$ in P and compute the Hilbert values of the residing points. The TMH-Tree then indexes the points using their Hilbert values as search keys. Each leaf entry p has the form $<p.id, p.x, p.y, p.hv>$ and an intermediate entry e is a quadruplet ($e.h$, $e.k$, $e.ptr$, $e.t$), where the semantics are the same as in BSL, except that $p.hv$ is computed locally within each partition (instead of the entire dataspace). To avoid confusion, the term *cell* is used only for the Hilbert grid. The term *partition* is used for the grid constructed over D. Note that the value of o may be different for each partition. Similar to m, the choice of o may have a significant effect on performance. Section 4.3 contains models for choosing appropriate values of m and o.

All partitions are indexed by a *Domain Partition Merkle-Tree* (DPM-Tree). The DPM-Tree is a binary tree that organizes digests in a way similar to the MH-Tree. A leaf node of the DPM-Tree corresponding to partition P contains a pointer $P.R$ to the root of the TMH-Tree embedded in P, timestamps $P.LT$ and $P.ST$, order $P.o$, and a digest $P.H$. $P.LT$ ($P.ST$) is the timestamp of the last (second last) update that occurred in P ($P.LT \geq P.ST$). $P.H$ is computed on the concatenation of the digests contained in the root of the TMH-Tree along with $P.ST$ and $P.o$. The information in intermediate nodes is inserted bottom-up. An intermediate node N contains value $N.H$, which is the digest of the concatenation of the digests of its children, and timestamp $N.T$ which is the maximum between the timestamps of its children. In order to establish a neighborhood relationship among the nodes of the DPM-Tree, we consider that the root corresponds to the entire dataspace. Its two children are generated by splitting the space *vertically* into two equal subspaces. Subsequently, each child generates two new children by dividing its subspace *horizontally* into two new equal subspaces. This process is repeated recursively (selecting the splitting axis in a round-robin fashion) until the final subspaces are single partitions (leaf level 0).

Let H_{DPM} (T_{DPM}) be the digest (timestamp) in the root of the DPM-Tree. The DO computes $h(H_{DPM} \mid T_{DPM} \mid L_x \mid L_y \mid U_x \mid U_y)$, signs it and sends it to the SP along with the dataset. CSA supports multiple updates at the same timestamp as follows. The TMH-Trees are first modified, as discussed in Section 3, without altering any hash or timestamp value, and the visited paths are marked. When an entry is deleted from a full *intermediate node* (i.e., there is no underflow), it is replaced with a *dummy* value, so that the order of the remaining entries in the node remains the same. Then, the marked paths are revisited and the digests and timestamps are computed bottom-up, only once. Finally, a single depth-first traversal of the DPM-Tree locates the leaf nodes that correspond to the affected partitions and computes the appropriate digests and timestamps bottom-up. However, if at some instant the number of points in a partition P changes significantly, the DO may decide to change $P.o$ in order to improve performance. In this case, it notifies the SP that computes the new Hilbert values for the residing points and re-builds the embedded TMH-Tree.

Finally, the SP maintains some book-keeping structures for query monitoring. Specifically, each partition P is associated with an influence list $P.IL$ that stores the identifiers of the queries that overlap with P. A hash table QT on $q.id$ maintains a tuple $<q.id, q.rg, q.t>$ for every running query q, where $q.id$ is a unique identifier, $q.t$ is the timestamp of q's last VO update., and $q.rg$ is the spatial range [q_{Lx}, q_{Ly}, q_{Ux}, q_{Uy}] of q.

4.2 Query Processing

First we discuss snapshot query processing and verification. Upon receiving a spatial range *q*, the SP starts its execution by traversing the DPM-Tree. At every visited node, the SP obtains the subspace covered by its subtree. This is performed by recursively breaking *D* into two equal spatial subspaces, either horizontally or vertically, in a round-robin fashion. If *q* overlaps with the corresponding subspace of a node, the algorithm continues traversing its subtree. Otherwise the node's digest is appended to the *VO*. Upon reaching the leaf level of the DPM-Tree, if *q* does not overlap with a partition *P*, *P.H* is inserted into the *VO*. Otherwise, the algorithm appends *P.ST* and *P.o* into the *VO*, and decomposes *q* into a set of 1D intervals, by determining its intersections with the embedded Hilbert curve. Then, it expands the intervals to include boundary records, sorts them on their lower boundary and stores them in a list *L*. The expanded sub-queries are issued to the embedded TMH-Tree, using the multi-range algorithm of BSL (Figure 3.2).

Given the *VO* and *D*, the client can verify its correctness, by computing the digest H_{DPM} at the root of the DPM-Tree. The process is similar to the one described in Section 3, except that intervals are used to determine the extents of each partition on-the-fly. After the client computes H_{DPM}, it hashes it with T_{DPM} and *D*, and matches it against the signature of the DO. The actual results are extracted during the verification process.

Proof of soundness (sketch): Soundness is ensured by the hierarchical organization of the hashes in the two trees and the collision-resistance of the hash function. If an adversary alters or deletes a point from the dataset, or inserts a bogus one, the change will propagate until H_{DPM}. Thus, the client will reconstruct an H_{DPM} that does not match the DO's signature. □

Proof of completeness (sketch): Completeness is satisfied for two reasons. (i) The client has *D* and, therefore, while reconstructing H_{DPM}, it can verify that the SP returns a partial *VO* for every partition overlapping the query. (ii) For each such partition it also has *P.o* and, therefore, it can establish completeness in the way that we discussed in Section 3. Finally, note that, for every *P*, *P.o* is incorporated in *P.H* and *D* is in the signature. Consequently, the SP must send them intact in order for the signature to be certified. □

When the SP receives updates from the DO, it alters the indices and determines the affected queries whose range overlaps with partitions where at least one update has occurred. Finally, it generates a new *VO* for each such query. Motivated by the observation that an updated *VO* shares common components with the previous one, we propose a *virtual caching mechanism* (*VCM*) to further reduce the communication cost. The term *virtual* is due to the fact that the SP does not store the *VO* for any query. Instead, each client keeps in its own cache the *VO* that was received last. *VCM* works as follows: whenever a node *N* of the DPM-Tree (or node entry *e* of the TMH-Tree) is visited during processing query *q*, its corresponding timestamp is checked against *q.t*. If *q.t* is equal or larger, then token *Hit* is inserted into the *VO* and the traversal for the *N*'s (*e*'s) subtree stops. This token instructs the client to retrieve the partial *VO* corresponding to *N*'s (*e*'s) subtree from the cached *VO*. Upon receiving a new *VO*, the client merges the components contained in the updated *VO* with the ones in the cache. Eventually, it reconstructs H_{DPM} (as described above) and matches it against the signature.

Proof of temporal completeness (sketch). Suppose that the initial computation of a query q occurs at a time t and the *VO* is sent to the client. The client successfully verifies its correctness and stores it as *cachedVO*. Now assume that at later time t' ($>t$) one (or more) update(s) takes place in some partition P that overlaps with q, but the SP does not send a new *VO* to the client. Subsequently, another update occurs that affects q. This time the SP generates *newVO* and sends it to the client (along with new signature and T_{DPM}). We distinguish two cases: (i) the *newVO* contains a partial *VO* corresponding to P, thus also *P.ST*. The client compares *P.ST* with the cached T_{DPM} ($=t$). Since $P.ST > t$, at least a potential result update (at *P.ST*) was omitted and the client is alarmed. (ii) *newVO* contains a *Hit* token that corresponds to P. Since the actual *P.ST* is different than the one included in *cachedVO*, the client reconstructs a false *P.H* value and soundness is violated. □

4.3 Computing the Grid Granularity

The granularity m of the dataspace partitioning greatly affects the efficiency of CSA. If m is too coarse (i.e., there are very few partitions), the ability of CSA to reduce false transmissions is subdued. On the other hand, a large number of partitions leads to a tall DPM-Tree and numerous TMH-Trees, which also adversely affects performance. Moreover, for skewed distributions, many of the partitions may contain few or no records at all. Since manually tuning m at the DO is both costly and error-prone, in the sequel we first establish cost models and then clarify the selection of m based on these models.

Our analysis focuses on the expected *VO* (*EVO*) size, for two reasons. First, the *VO* must be transmitted from the SP to the client through the network, which is usually the bottleneck of the entire system. This is especially true for mobile clients (e.g., PDAs), where battery consumption is a major concern (wireless transmissions consume significantly more power than offline computations [6]). The second reason is that other performance goals, such as minimizing the computation at the SP and the client, are strongly correlated with *EVO*. Intuitively, the larger the *EVO*, the more nodes are visited during query processing, and subsequently processed by the client to reconstruct the root digest.

Without loss of generality, we normalize the values along each axis of the dataspace to [0, 1]. In order to keep the analysis tractable, we make the following simplifying assumptions: (i) all partitions have the same length. (ii) The updates follow the distribution of the initial dataset, i.e., the cardinality of each partition does not change significantly over time. When this assumption does not hold, the DO and SP can periodically re-compute m and rebuild the structures of CSA accordingly. (iii) Each query q has expected length $l_q \in (0, 1]$ along each axis. Furthermore, its lower boundary (along each axis) is uniformly distributed in $[0, 1- l_q]$. (iv) The virtual caching mechanism is disabled as its effects are not significantly influenced by the partitioning granularity m.

We use symbol $P_{i,j}$ ($1\leq i, j \leq m$) to denote the partition covering the region $[(i-1)/m, i/m] \cdot [(j-1)/m, j/m]$, which contains a known number $|P_{i,j}|$ of points. Query q takes the shape of a square with length l_q along each axis. CSA involves an initial *VO* computation for a query *q*, as well as the construction of a new *VO* whenever *q* is affected by updates. Let $EVO_{init}(q)$ be the expected size of the initial *VO* of *q*, and $EVO_{upd}(q)$ the expected size of the *VO* generated due to an update. For a given random query sample *QS* (e.g., drawn from a past query log) with cardinality $|QS|$, and a number of timestamps *NU* that updates occur, *EVO* is computed by

$$EVO = \frac{\sum_{q \in QS} EVO_{init}(q) + NU \cdot EVO_{upd}(q)}{|QS| \cdot (NU+1)} \tag{4.1}$$

Regarding EVO_{init}, CSA includes in the *VO* five different types of information: (i) the result set of *q*, (ii) two boundary records for proving completeness, (iii) timestamps of each partition overlapping *q*, which collectively prove temporal completeness, (iv) the digests inserted during the traversal of the DPM- and the TMH-Trees that is used by the client to verify correctness and, finally, (v) the signature of the DO. We do not consider the tokens since their sizes are negligible. Let S_r be the length of a record and $|q|$ be the average number of tuples in the query result set. Types (i) and (ii) consume $S_r \cdot (|q|+2)$. Given the query extent l_q, $|q|$ can be calculated using standard selectivity estimation techniques (e.g., sampling [1], histograms [7], probabilistic models [8]). If *qp* is the number of partitions intersecting *q*, and S_t is the size (in bytes) of a timestamp representation, the size of (iii) is $qp \cdot S_t$. Since each partition has extent $1/m$ on each axis, the expected value for *qp* is $\lfloor 2 \cdot m \cdot l_q \rfloor + 1$. Regarding (iv), we use symbols EVO_D and EVO_T to denote the total size of the digests appended to the *VO* when traversing the DPM-Tree and *all* the TMH-Trees, respectively. Finally, (v) equals the size of one signature (let S_s). Summarizing, EVO_{init} is given by:

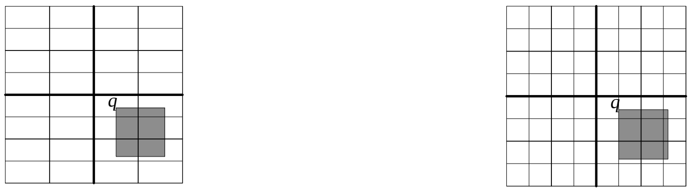
$$EVO_{init}(q) = \left(|q|+2\right) \cdot S_r + qp \cdot S_t + EVO_D + EVO_T(q) + S_s \tag{4.2}$$

We next focus on EVO_D. Note that EVO_D consists of the digests of all *pruned* nodes of the DPM-Tree. A node is pruned, if and only if (i) it does not overlap the query, and (ii) none of its ancestors is pruned (otherwise it is not visited at all). Let $OVN(i)$ be the number of nodes at depth *i* (the root being at depth 0) that overlap with *q*. The number of nodes at depth *i* satisfying condition (i) is $2^i - OVN(i)$. Among these nodes, some are descendents of pruned nodes at higher levels and, thus, violate condition (ii). Given that the DPM-Tree is binary, a node at depth *j* ($j < i$) has 2^{i-j} descendents at depth *i*. Therefore, assuming that $PN(i)$ is the number of pruned nodes at depth *i*, Equation 4.3 gives both $PN(i)$ and EVO_D (S_h is the size of a digest). Note that since *m* is the partitioning granularity for each of the two axes, the height of the DPM-Tree is $\lfloor \lg m^2 \rfloor = 2 \cdot \lfloor \lg m \rfloor$.

$$EVO_D = \sum_{i=0}^{2 \cdot \lfloor \lg m \rfloor} PN(i) \cdot S_h$$
$$\text{where } PN(i) = 2^i - OVN(i) - \sum_{j=0}^{i-1} PN(j) \cdot 2^{i-j} \tag{4.3}$$

We next determine *OVN*. Figure 4.2 depicts the spaces covered by the subtrees of the DPM nodes at an odd and even depth (the root being at depth 0). Specifically, Figure 4.2a (4.2b) shows the 5th (6th) depth. In general, each node at depth i covers $1 / 2^{\lfloor i/2 \rfloor}$ extent on the x-axis and $1 / 2^{\lceil i/2 \rceil}$ on the y-axis. The number of cells overlapping q is thus calculated by:

$$OVN(i) = \left(\left\lfloor 2^{\lfloor i/2 \rfloor} \cdot l_q \right\rfloor + 1\right) \cdot \left(\left\lfloor 2^{\lceil i/2 \rceil} \cdot l_q \right\rfloor + 1\right) \tag{4.4}$$



(a) Odd depth (b) Even depth

Fig. 4.2: DPM-Tree nodes and query q

The derivation of EVO_T depends on the order of the Hilbert curve used in each partition. Let $P.o$ be the order in partition P. An overly coarse (fine) $P.o$ leads to limited indexing effectiveness (many empty cells) and consequently sub-optimal performance. We determine $P.o$ based on the following observation: when records are uniformly distributed in partition P, ideally each Hilbert cell should contain exactly one record. Note that according to our experiments, the optimal m is usually large enough for this local uniform distribution assumption to hold. Therefore, an appropriate value for $P.o$ is $\lg|P|/2$.

For partitions completely contained in q, all data records (and no digests) are inserted into the VO. For partitions that partially overlap with q (i.e., those on the boundary of q as shown in Figure 4.2), the digests of the pruned sibling nodes during the TMH-Tree multi-range traversal are added to the VO (in addition to the records satisfying q). However, the exact analysis of this traversal is very complicated because (i) it involves calculating the number of ranges q is broken into, which itself is a difficult task [14], and (ii) the ranges have different sizes, meaning that the common ancestors are at different levels. Instead, we employ the following approximation for the multi-range traversal: for each TMH-Tree in a partition that partially overlaps q, we count two complete root-to-leaf paths, adding to EVO the digest of the siblings of all visited nodes. This method overestimates the traversal path by counting the root-to-split-node part twice, but on the other hand also underestimates it by not taking into account the small up-and-down paths inside the envelop of the two root-to-leaf paths. These two contradicting factors are expected to partially cancel each other out. Moreover, with reasonably fine partitioning granularity m, (i) the number of partitions on the boundary of q is much smaller than those within q, and (ii) each TMH-Tree is expected to have a small height, both of which render the approximation error insignificant. Summarizing, the formula for EVO_T is (f is the expected fanout of the TMH-Tree):

$$EVO_T(q) = \sum_{P \in PP} 2 \cdot S_h \cdot (f - 1) \cdot \left\lfloor \log_f |P| + 1 \right\rfloor \tag{4.5}$$
$$\text{where } PP = \{P \mid P \text{ partially overlaps } q\}$$

Combining Equations 4.2, 4.3 and 4.5 yields the complete model for EVO_{init}. We next derive EVO_{upd}. Recall that in CSA, the SP sends a new *VO* only when at least one update happens in a partition intersecting with *q*. According to the assumption that the updates follow the same distribution as the initial dataset, the probability that an update falls in any one of QP_1, QP_2, …, QP_{qp} is $\Sigma_i|QP_i|/(\Sigma_j\Sigma_k|P_{j,k}|$, $1\leq i\leq qp$, $1\leq j,k\leq m$. Therefore, for a batch of $|U|$ independent update operations (i.e., insertions or deletions) occurring at a timestamp, the probability $Prob_{VO}(q)$ that the SP transmits a new *VO* (i.e., *q* is affected by any one of these updates) is:

$$Prob_{VO}(q) = 1 - \left(1 - \left(\sum_{i=1}^{qp} |QP_i| \Big/ \sum_{j=1}^{m}\sum_{k=1}^{m} |P_{j,k}|\right)\right)^{|U|} \qquad (4.6)$$

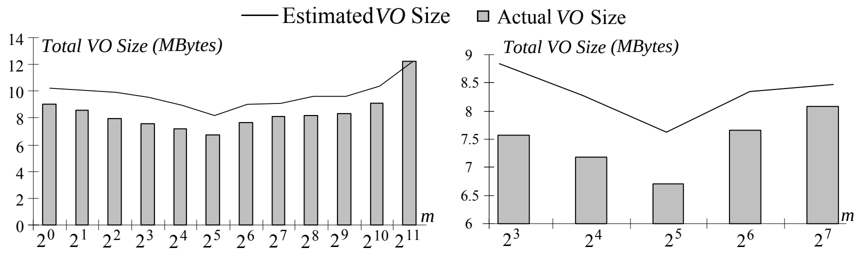
Similar to the case of $EVO_T(q)$, $Prob_{VO}(q)$ is a function of the values of $|QP_i|$ $(1\leq i\leq qp)$, which, in turn, depends on the position of *q*. Let $EVO_{init}(q)$ be the expected *VO* size for a particular query *q*, which is obtained by combining Equations 4.2, 4.3 and 4.5. Then $EVO_{upd}(q) = EVO_{init}(q) \cdot Prob_{VO}(q)$.

Equipped with the above models, we present a simple and effective algorithm *Bestm* to compute an appropriate value for the partitioning granularity *m*. Initially, *Bestm* sets *m* to a maximum value m_{max}. A good choice for m_{max} is half of the first power of 2 that is larger than the data set cardinality. It then scans the dataset once to compute the cardinality for each partition, and utilizes this information to compute *EVO* using the cost models. After that, it decreases *m* to $m_{max}/2$, and computes the corresponding *EVO*. Observe that at this stage it is unnecessary to scan the dataset again to compute the cardinality of the partitions, since these can be obtained by aggregating the corresponding partitions in the previous step. At subsequent steps, *m* is reduced by half each time, and *EVO* is estimated, until $m = 1$. Among all considered values for *m*, the one achieving the minimum *EVO* is chosen as the partitioning granularity for CSA.

5 Experimental Evaluation

We implemented our methods using the Crypto++ library [4], and deployed them on a Core 2 Duo 2.2GHz CPU with 2GBytes of RAM. Each record *r* consumes 100 bytes and contains two search keys *r.x* and *r.y*. The values of *r.x* and *r.y* are obtained from the real dataset CAR (California Roads, available at *www.rtreeportal.org*), and are normalized to [0, 1]. At every timestamp, updates arrive at a rate of *AR*. An update involves a deletion of a random tuple and an insertion of a new one with the same id but different keys. Consequently, the number of update operations $|U| = 2\cdot AR$, and dataset cardinality *DC* is constant at all times. The new key values follow their initial distribution. We monitor *QC* running queries, which are uniformly distributed in the dataspace and cover approximately 0.1% of the data domain.

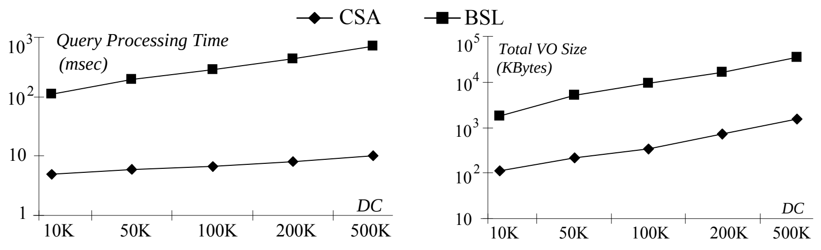
First, we determine the optimal partitioning granularity *m* for CSA, using the models of Section 4.3. We set *DC* = 100K, *QC* = 1K and *AR* = 100, and we disable the *VCM*. Figure 5.1a depicts the estimated total *VO* size generated for all 1K queries with respect to *m*, as well as its actual size computed in our experiments. Figure 5.1b zooms into the part of Figure 5.1a, where $2^3 \leq m \leq 2^7$. The error of our estimation is 5-17%. Our cost models successfully determine the best granularity, which in this case is $m=2^5$. In the sequel, we set m to the best granularity as estimated by our models.



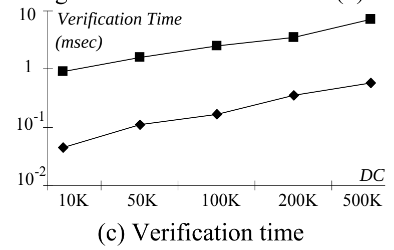
(a) Effect of partitioning granularity (b) Optimal partitioning granularity

Fig. 5.1: Total *VO* size vs. partitioning granularity *m*

Figure 5.2 assesses the effect of the dataset cardinality (*DC*), when *QC* = 1K, *AR* = 100 and the *VCM* is switched on. Figure 5.2a shows the total query processing time per timestamp at the SP. BSL incurs a considerable computational overhead since it re-processes all queries at each timestamp. On the other hand, CSA executes only the queries affected by the updates, as well as a small number of queries that correspond to false transmissions. Figure 5.2b depicts the total *VO* size generated for all queries per timestamp versus *DC*. CSA outperforms BSL, because (i) it executes only the queries affected by the updates, and (ii) the *VCM* enables the SP to omit sending *VO* components corresponding to DPM/TMH subtrees that are not altered by the updates. Figure 5.2c illustrates the verification cost per timestamp at each client. In BSL the client has to verify its query at every timestamp, whereas in CSA it establishes correctness only when its query is affected by an update.



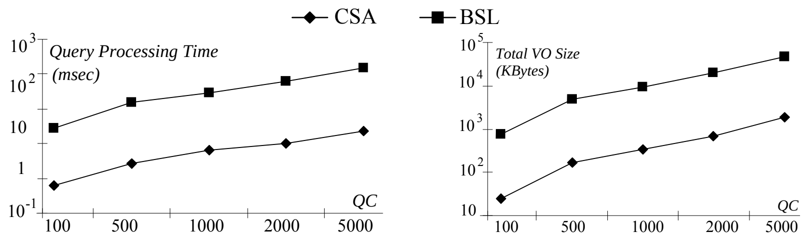
(a) Query processing time (b) Total *VO* size



(c) Verification time

Fig. 5.2: Performance vs. dataset cardinality *DC*

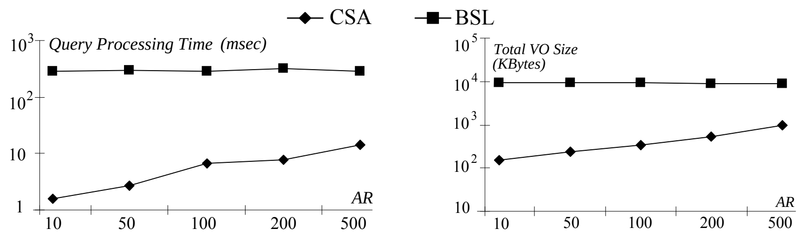
We next investigate the impact of the query cardinality (*QC*), after setting *DC* = 100K and *AR* = 100. Figures 5.3a and 5.3b plot the query processing cost at the SP and the communication cost at every timestamp, respectively. Both costs grow linearly with *QC*. In BSL, each query is evaluated at every timestamp. Therefore, the computational effort at the SP as well as the information communicated between the SP and the client increases with the number of running queries. In CSA, more queries are likely to be affected by the updates (in which case a new *VO* is generated and transmitted to the client) in the presence of a large number of running queries.



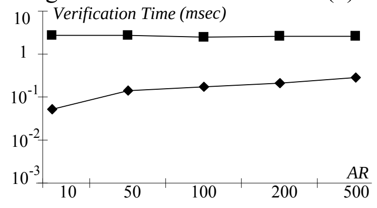
(a) Query processing time (b) Total *VO* size

Fig. 5.3: Performance vs. number of queries *QC*

Figures 5.4a and 5.4b demonstrate the query processing and the communication cost, respectively, versus the arrival rate *AR* (*DC* = 100K, *QC* = 1K). BSL is not influenced by *AR*. The overhead of CSA converges to that of BSL for large values of *AR* because, as more updates occur, more queries are affected and re-evaluated. For these queries, a new *VO* must be produced and transmitted. Furthermore, a high *AR* reduces the effectiveness of *VCM* because the updates alter a large part of the DPM- and TMH-Trees and, consequently, invalidate many *VO* components in the clients' cache.



(a) Query processing time (b) Total *VO* size



(c) Verification time

Fig. 5.4: Performance vs. arrival rate *AR*

Finally, Figure 5.4c shows the verification time at the client at every timestamp versus *AR*. As expected, the verification burden at the client increases for high arrival rates, because its query is affected by an update with higher probability. An interesting observation is that the curve of CSA converges faster to that of BSL, in comparison to the corresponding curves of Figures 5.4a and 5.4b because the *VCM* alleviates the processing and communication costs, but not the verification effort; to establish correctness, the client first has to combine the newly received *VO* with the one in its cache. Therefore, the client eventually verifies a *VO* with size equivalent to the one generated when the *VCM* is disabled.

6 Conclusions

In this paper we address continuous range processing and authentication on highly dynamic spatial databases. We assume a database outsourcing environment, where a service provider (SP) returns to the clients the query results, as well as authentication information necessary to establish their correctness. Due to the dynamic environment, clients must also be able to prove temporal completeness, i.e., that they did not miss any results in-between successive updates. We first propose BSL, a method that achieves these goals at the expense of false transmissions. Next, we introduce CSA, a scheme that utilizes a space partitioning scheme and an efficient caching mechanism to reduce the cost (processing and communication) for both the SP and the client. We optimize the performance of CSA through a detailed analytical study. Finally, we conduct an exhaustive experimental evaluation and show that CSA significantly outperforms BSL for all performance metrics.

Acknowledgments. This work was supported by grant HKUST 6181/08 from Hong Kong RGC.

References

1. Babcock, B., Chaudhuri, S., Das, G. Dynamic Sample Selection for Approximate Query Processing. *SIGMOD*, 2003.
2. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 1997.
3. Cheng, W., Pang, H., Tan, K. -L. Authenticating Multi-dimensional Query Results in Data Publishing. *DBSec*, 2006.
4. Crypto++ Library. Available at www.eskimo.com/~weidai/benchmark.html.
5. Devanbu, P., Gertz, M., Martel, C., Stubblebine, S. Authentic Data Publication Over the Internet. *Journal of Computer Security* 11(3): 291-314, 2003.

6. 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.
7. Guha, S., Shim, K., Woo, J. Rehist: Relative Error Histogram Construction Algorithms. *VLDB*, 2004.
8. Getoor, L., Taskar, B., Koller, D. Selectivity Estimation using Probability Models. *SIGMOD*, 2001.
9. Hacıgümüş, H., Iyer, B., Mehrotra, S. Providing Databases as a Service. *ICDE*, 2002.
10. Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L. Dynamic Authenticated Index Structures for Outsourced Databases. *SIGMOD*, 2006.
11. Li, F., Yi, K., Hadjieleftheriou, M., Kollios, G. Proof-Infused Streams: Enabling Authentication of Sliding Window Queries On Streams. *VLDB*, 2007.
12. Merkle, R. A Certified Digital Signature. *CRYPTO*, 1989.
13. Mokbel, M., Aref, W., Kamel, I. Analysis of Multi-Dimensional Space-Filling Curves. *GeoInformatica* 7(3): 179-209, 2003.
14. Moon, B., Jagadish, H. V., Faloutsos, C., Saltz, J. H. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *TKDE*, 13(1): 124-141, 2001.
15. National Institute of Standards and Technology. FIPS PUB 180-1: Secure Hash Standard. *National Institute of Standards and Technology*, 1995.
16. Narasimha, M., Tsudik, G. Authentication of Outsourced Databases Using Signature Aggregation and Chaining. *DASFAA*, 2006.
17. Pang, H., Jain, A., Ramamritham, K., Tan, K.-L. Verifying Completeness of Relational Query Results in Data Publishing. *SIGMOD*, 2005.
18. Pang, H., Tan, K.-L. Authenticating Query Results in Edge Computing. *ICDE*, 2004.
19. Papadopoulos, S., Yang, Y., Papadias, D. CADS: Continuous Authentication on Data Streams. *VLDB*, 2007.
20. Rivest, R. L., Shamir, A., Adleman, L., A method for Obtaining Digital Signatures and Public-key Cryptosystems. *Communications of the ACM*, 21(2):120-126, 1978.
21. Yang, Y., Papadopoulos, S., Papadias, D., Kollios, G. Spatial Outsourcing for Location-based Services. *ICDE*, 2008.