CADS: Continuous Authentication on Data Streams

Stavros Papadopoulos   Yin Yang   Dimitris Papadias

Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{stavros, yini, dimitris}@cse.ust.hk

ABSTRACT

We study processing and authentication of long-running queries on outsourced data streams. In this scenario, a data owner (DO) constantly transmits its data to a service provider (SP), together with additional authentication information. Clients register continuous range queries to the SP. Whenever the data change, the SP must update the results of all affected queries and inform the clients accordingly. The clients can verify the correctness of the results using the authentication information provided by the DO. Compared to conventional databases, stream environments pose new challenges such as the need for fast structure updating, support for continuous query processing and authentication, and provision for *temporal completeness*. Specifically, in addition to the correctness of individual results, the client must be able to verify that there are no missing results in between updates.

We face these challenges through several contributions. Since there is no previous work, we first present a technique, called REF, that achieves correctness and temporal completeness but incurs false transmissions, i.e., the SP has to inform clients whenever there is a data update, even if their results are not affected. Then, we propose CADS, which minimizes the processing and transmission overhead through an elaborate indexing scheme and a virtual caching mechanism. Finally, we extend CADS to the case where multiple owners outsource their data to the same SP. The SP integrates all data in a single authentication process, independently of the number of DOs.

1. INTRODUCTION

Database outsourcing [HIM02] has recently received considerable attention. According to this model, a data owner (DO) outsources its database to one (or more) specialized service providers (SPs) that have the necessary computational power and tools to support advanced query processing. Clients issue their queries directly to the SP. Outsourcing provides several benefits for all parties involved: (i) the DO does not need to acquire or dedicate the resources necessary for running a full-scale DBMS, (ii) the SP can achieve economies of scale by serving multiple owners, and (iii) the clients can obtain the data by a SP that is close in terms of network latency. Furthermore, the system robustness is improved because the DO ceases to be the single point of failure. However, since the SP is not the real owner of the data, it must be able to prove (to the users) the *soundness* and *completeness* of the query results. Soundness ensures that all the records returned originate from the DO and no spurious records exist. Completeness guarantees that all the tuples that satisfy the query are present in the result set. We refer to these two terms collectively as *correctness*.

Existing systems, presented in Section 2, use the general framework of Figure 1.1. The DO obtains, through a (trusted) key distribution center, a *private* and a *public* key. The private key is known only to the DO, whereas the public one is available to the clients. The DO signs the dataset and transmits it along with the signature (created using its private key) to the SP. The SP keeps the data and the signature locally. In order to facilitate query processing, the dataset is indexed by an authenticated data structure (ADS). This is similar to a conventional index, but it contains additional information for proving the correctness of the results. When a client issues a query, the SP generates a *verification object* (*VO*) by accessing the ADS. The *VO* contains the result set along with the authentication information necessary for proving correctness. The SP sends the *VO* and the corresponding signature to the client. The client can verify correctness by matching the received signature against the *VO* and the public key of the owner. Alternative implementations of the framework differ on the choice of signature techniques, ADS, and verification processes. Furthermore, most systems necessitate the maintenance of identical copies of the ADS at the DO.

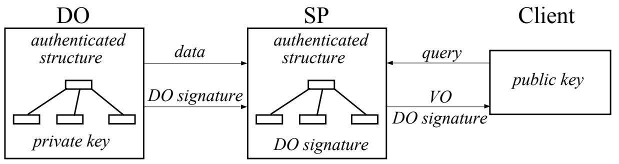


Figure 1.1 Authentication framework for outsourced databases

All previous work in the database literature focuses on disk-resident and relatively static datasets. On the other hand, increasing monitoring of transactions, ecological parameters, homeland security, RFID chips etc., establishes new and highly dynamic environments for data outsourcing. As an example assume a SP that receives current stock values from one or more stock exchanges. Subscribers register long-running queries at the SP. Whenever a stock update 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 establish their correctness. As a second application, consider a web-based SP that collects item prices from different outlets, but is not allowed to publish them (in order to avoid direct competition leading to discounts). A client looking for a bargain registers his/her price range (e.g.,

VLDB '07, September 23-28, 2007, Vienna, Austria.

below \$500) for an item (e.g., Sony PS3) as a query. Each time an outlet posts a price in the desired range, the SP sends the corresponding verification object to the client that is able to automatically verify it.

In the database literature, authenticated query processing has been traditionally linked with outsourcing. However, similar concepts also apply in situations where the DO and the clients communicate directly. Consider, for instance, a sensor network where a *sink* collects temperature readings from various sensors. Clients (e.g., environmental agencies, fire departments) register continuous range queries to the sink (e.g., report sensors with temperature reading in the range [45-50]). Assuming that the network is unreliable, authentication is necessary for establishing the correct transmission of the results before taking action. As an alternative, a server may gather intelligence data from satellites, and authentication is required for detection of attacks on the communication channel. In both cases, the sink/server plays the combined role of the DO *and* the SP in the framework of Figure 1.1. For consistency with previous work, we follow that framework (assuming distinct DO and SP), but the proposed methods can be used for authentication without outsourcing.

The dynamic nature of the data and the potentially large number of long-running queries pose several challenges. First, a system for continuous authentication on data streams must accommodate very fast updates and, at the same time, support efficient query processing. Second, it must include effective mechanisms for minimizing the communication cost with the clients, and their verification effort. Third, the SP may have to integrate data from several stream sources (e.g., stock exchanges, outlets, servers) in a single authentication process. Finally, in addition to correctness, the clients must be able to verify *temporal completeness*, i.e., confirm that they receive all result changes that are relevant to their queries. We aim at solving the above problems with the following contributions:

1. Due to the lack of previous work on authenticated data streams, we first present a technique, called REF, used as a benchmark in our evaluation. REF achieves correctness and temporal completeness but incurs *false transmissions*, i.e., the SP has to inform clients whenever there is a data update, even if their results are not affected.
2. We propose CADS, a technique that minimizes the processing and transmission overhead through an elaborate indexing scheme and a virtual caching mechanism. Both CADS and REF are main memory-based in order to achieve real-time query evaluation and fast structure updating.
3. We extend CADS for situations where a SP hosts stream data from several owners, but each query involves a single verification process.
4. We show through extensive experiments that CADS outperforms REF significantly in all aspects. Furthermore, its efficiency permits its application in highly dynamic environments involving numerous clients and a large volume of data.

The rest of the paper is organized as follows. Section 2 surveys the basics on cryptography and the related bibliography. Section 3 presents REF and Section 4 focuses on CADS. Section 5 deals with multiple owners. Section 6 experimentally evaluates CADS and compares it against REF. Finally, Section 7 concludes the paper with directions to future work.

2. BACKGROUND

A one-*way, collision-resistant hash* function is a computationally efficient mapping h: $\{0,1\}^* \rightarrow \{0,1\}^l$. The output of h is called *digest* and has fixed length l. The function is such that, given a digest $y = h(M)$, it is computationally hard (i) to derive the message M from y, and (ii) to find another message M' such that $y = h(M')$. In this work we employ SHA1 [NIST95], which takes variable-length inputs and produces 20-byte digests. In the sequel, the term hash function (h) implies a one-way, collision-resistant hash function.

A public-key digital signature scheme is used to verify that a message is not falsified (*integrity*), and that it originates from the party that signs it (*authenticity*). Our techniques adopt RSA [RSA78]. The digital signature generated using RSA has a typical size of 128 bytes. A key generator creates a private key a and a public key (b, c). The signer keeps the private key and publishes the public key. To create the digital signature sig of a message M, the signer performs operation $sig = sign(M, a, c) = h(M)^a \bmod c$. Given sig and the signer's public key, the verifier can confirm the authenticity of M by checking if $verify(M, sig, b, c) = sig^b \bmod c$ equals $h(M)$.

Mykletyn et al. [MNT04] devise two schemes that aim at reducing the communication cost and the verification time, when multiple signatures are to be transferred and verified at once. Both schemes allow *aggregating* multiple signatures into a single one that can be verified almost as fast as an individual signature. The first scheme is called Condensed-RSA and uses RSA for aggregating signatures generated by a single signer. The second is called BGLS and is based on the usage of elliptic curves and bilinear mappings to aggregate signatures generated by different signers. BGLS is expensive and elliptic curves are not as widely used as RSA.

The *Merkle Hash Tree* (*MH-Tree*) [M89] is a main-memory binary tree originally proposed for efficient authentication of equality queries in a database sorted on the query attribute. Every record corresponds to a leaf node that stores the hash value of the binary representation of the record. The tree is constructed bottom-up, with each internal node storing the hash value of the concatenation of the hash values of its children. The owner signs the hash value stored in the root of the tree. Consider that a client asks for record r_3 in the MH-Tree of Figure 2.1. The SP accesses the tree to locate the record. During the tree traversal, apart from record r_3, it inserts into the *VO* the hash value stored in the sibling of every visited node (i.e., h_{12} and h_4). Having the *VO*, signature sig and the owner's public key, the client can verify the authenticity of the result by reconstructing the hash value of the root as $h_{1234} = h(h_{12}|h(h(r_3)|h_4))$ and matching it against sig ('|' denotes concatenation).

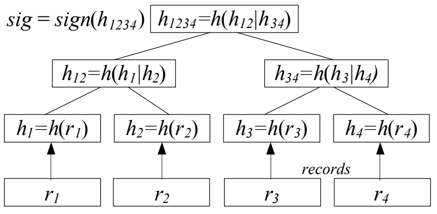


Figure 2.1 Example of the Merkle Hash Tree

Devanbu et al. [DGMS03] utilize the MH-Tree for answering one-dimensional range queries, satisfying soundness and completeness. They also extend their method to multiple dimensions, combining the MH-Tree with the *Range Search Tree* [BKOS97]. Martel et al. [MND+04] develop a generalized framework for creating efficient authenticated versions of a broad class of data structures. Finally, Goodrich et al. [GTTC03] introduce techniques for authenticating data structures that represent graphs and geometric objects.

The first disk-based authenticated structure that provides soundness, but not completeness is the VB-Tree [PT04], which is a B+-Tree augmented with signed digests. *Signature chaining* [PJRT05, NT06] guarantees both soundness and completeness, but has several drawbacks. First the owner must construct a number of signatures equal to the dataset cardinality and transmit them to the SP. The SP must store these signatures, each consuming 128 bytes (comparable to the size of a typical record). The large space overhead also affects the cost of query processing. Finally, the transmission of a result to the client contains a signature for each tuple. The client has to verify all these signatures, which can be a rather expensive process. Signature aggregation can be used to reduce the communication and verification cost.

The current state-of-the-art, disk-based, authenticated structure is the *Merkle B-Tree* (*MB-Tree*) [LHKR06]. The MB-Tree is basically a B+-Tree that hierarchically organizes digests, following the concept of the MH-Tree. Every 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 B+-Tree search key and $E.h$ is a hash value computed on the concatenation of the hash values of the entries contained in N_c. An entry in a leaf node is associated with the hash value of a record. The owner signs the hash of the concatenation of the hash values contained in the root of the tree. The EMB-Tree [LHKR06] embeds a MH-Tree inside each MB-Tree node in order to reduce the *VO* size.

Under the data stream paradigm, tuples generated by various sources are collected at a data stream management system (DSMS), where users register continuous queries. When a new tuple arrives, all relevant queries are re-evaluated. Query processing is usually performed by routing tuples through operator trees, where operators closely resemble their traditional counterparts such as selections or joins. Depending on the application characteristics, DSMSs adopt different models regarding the validity of tuples. A popular model assumes a *sliding window* of a given time frame w, i.e., a tuple s expires w time units after its arrival. In this case, all arrivals in the system that correspond to insertions and deletions are implicit. Another common model assumes *positive-negative* tuples, i.e., the DSMS receives a negative tuple $–s$ that takes the same route through the operator tree as s, and erases all occurrences of its positive counterpart. Surveys of various DSMSs can be found in [BBD+02, GO03]. Nevertheless, to the best of our knowledge, none of the existing DSMSs considers authentication issues.

3. A REFERENCE SOLUTION

This section introduces a competitor, hereafter called REF (for *reference* solution), used as a benchmark in our experimental evaluation. We first assume a SP that collects data from a single DO (multiple owners are discussed in Section 5). For simplicity, we consider that each tuple r has only two attributes: the primary key $r.id$ and the search key $r.k$ (queries are ranges on $r.k$). According to REF, tuples are sorted on the search key and indexed by an authenticated structure called the *DMH-Tree* (for *Dynamic Merkle Hash-Tree*), i.e., a MH-Tree where each node has 2 or 3 entries. Figure 3.1 illustrates an example DMH-Tree. Each leaf node (level 0) contains 2 or 3 records. For intermediate nodes, each entry e is a triplet ($e.h$, $e.k$, $e.p$), where $e.k$ is the search key of the first record in the subtree of e, and $e.p$ is a pointer to the corresponding child node. The value of $e.h$ depends on the level. For level 1, $e.h$ is a hash value on the concatenation of all records in the node pointed by $e.p$; for the upper levels, $e.h$ is computed on the concatenation of the hash values of the entries in $e.p$. The DO and the SP maintain identical trees in main memory. In addition, the DO computes a value H_{root} by hashing the concatenation of the hash values contained in the root of the tree e.g., in the example of Figure 3.1, $H_{root} = h(h_{1,5} | h_{6,12})$. Then it applies its private key to sign H_{root}, using the RSA public key cryptosystem. The SP stores a copy of this signature.

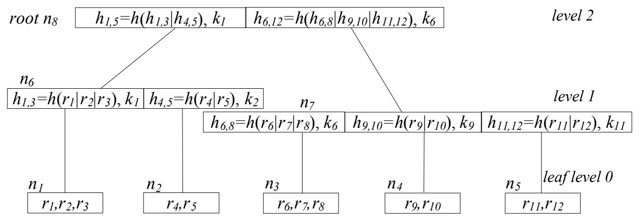

root n_8
$h_{1,5}=h(h_{1,3}|h_{4,5}), k_1$
$h_{6,12}=h(h_{6,8}|h_{9,10}|h_{11,12}), k_6$
level 2
n_6
$h_{1,3}=h(r_1|r_2|r_3), k_1$
$h_{4,5}=h(r_4|r_5), k_2$
n_7
level 1
$h_{6,8}=h(r_6|r_7|r_8), k_6$
$h_{9,10}=h(r_9|r_{10}), k_9$
$h_{11,12}=h(r_{11}|r_{12}), k_{11}$
n_1
n_2
n_3
n_4
n_5
leaf level 0
r_1,r_2,r_3
r_4,r_5
r_6,r_7,r_8
r_9,r_{10}
r_{11},r_{12}

Figure 3.1 An example of a DMH-Tree

The DMH-Tree supports fast (i.e., logarithmic) updates[1], based on the insertion/deletion algorithms of the B+-tree. Specifically, an insertion in a full (i.e., with 3 entries) node causes its split in two nodes, each containing 2 entries. On the other hand, a deletion from a node n with 2 entries leads to an underflow. Similarly to B+-trees, n first tries to *borrow* an entry from a full sibling node. If this is not possible, n is *merged* with a sibling. Since we do not use "right" pointers at the leaf level (as in B+-trees), in our context the term sibling signifies the previous or the next node under the same parent. In addition, the DMH-Tree can support multiple updates at the same timestamp. First, the structure is modified to accommodate all updates, without altering any hash value, but temporarily marking the visited paths. Then, the marked paths are revisited and the hash values are computed bottom-up. In this way, the (expensive) hash computations are performed only once.

The DMH-Tree handles snapshot query processing and verification as follows. Let q:[q_L, q_U] be a range query on $r.k$, where q_L (q_U) is the lower (upper) bound. The SP performs two top-down traversals to locate the tuples r_L and r_U immediately before q_L and after q_U, respectively. These *boundary records* are necessary to enforce completeness, i.e., that the SP does not omit results at the range limits. Then it expands q to [$r_L.k$, $r_U.k$] and applies the *RangeDMH* algorithm of Figure 3.2 to compute the verification object (*VO*), which contains the actual result and additional data so that the client can establish its correctness. Specifically, the *VO* includes: (i) the hash function of every pruned entry, (ii) the tuples in every visited leaf node, (iii) special tokens [and] that indicate the scope of a node. Consider for example a query that retrieves records r_5 to r_8 in Figure 3.1. The

[1] The original MH-Tree requires re-computation of hash values and reconstruction of the tree from scratch for every update.

expanded query covers tuples r_4 to r_9. The application of *RangeDMH* to the expanded query yields the *VO*: [[$h_{1,3}$ [r_4,r_5]][[r_6,r_7,r_8][r_9,r_{10}] $h_{11,12}$]]. Note that the tokens in the *VO* reveal the tree structure, e.g., [$h_{1,3}$ [r_4,r_5]] corresponds to the first root entry and the remainder to the second one. The SP transmits the *VO* and the owner's signature to the client.

```
RangeDMH(DMH_Node n, Expanded query q)
1.  Append [ to the VO
2.  For each entry e in n
3.      If n is an intermediate node
4.          If e intersects the query range // e may contain results
5.              RangeDMH(e.p, q) // e.p points to child node
6.          Else append e.h to the VO
7.      Else // n is a leaf node and e is a record
8.          Append e to the VO
9.  Append ] to the VO
```

Figure 3.2 Range query in the DMH-Tree

The verification process at the client utilizes the tree-structure information, encapsulated in the *VO*, to compute the hash value H_{root} of the root. Figure 3.3 illustrates the pseudo-code of *ReconstructH*$_{root}$. The main concept is similar to 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 (hashes or records) to compute the hash value of the node that started at the corresponding [. The hash values and records are appended to a buffer *B*, which after termination is used to derive $H_{root}=h(B)$. Having H_{root} and the signature of the DO, the client can establish authenticity and correctness using the public key of the DO. *ReconstructH*$_{root}$ is *online*, i.e., it performs a single linear scan of the *VO*. Note that the actual results (i.e., records r_5 to r_8 in the query range) are extracted in line 6. In addition, the client receives some boundary records (r_4, r_9, r_{10}) in the *VO*, which are not part of the result. Pang et al. [PJRT05] propose a solution for avoiding disclosure of boundary records, 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 such records, if necessary.

```
ReconstructH_root (VerificationObject VO)
1.  Initialize an empty string B
2.  While VO still has entries
3.      Remove next entry E from VO
4.      If E is a hash value h OR a record r
5.          Append E to B
6.          If E is a record r that satisfies the query, Report r
7.      If E is [, Append ReconstructH_root(VO) to B
8.      If E is ], Return hash(B)
```

Figure 3.3 Algorithm for reconstructing H_{root}

Proof of soundness: Suppose that a record *r* in leaf node *n* is bogus or modified. Because *h* is collision-resistant, the hash value of *n* (stored in the parent node) is different from that of the owner. The change propagates all the way to the root. Therefore, the reconstructed H_{root} is also different from the original, and the signature verification fails.

Proof of completeness: Given the boundary records, the client can detect that a result is missing, because the reconstructed H_{root} will not match the owner's signature. The only complication occurs when there are no boundary tuples, i.e., when the query contains the first and/or last record in the database. To cover this case, previous schemes [PJRT05, LHKR06] include two fictitious records at the beginning and the end of the dataset. Our approach does not require fictitious tuples. For ease of explanation, we consider that there is no left boundary (the case of right boundary is symmetric). Assume that the SP sends the complete result. Due to the depth-first traversal of *RangeDMH*, the first non-token entry of the *VO* is the first record r_1 satisfying the query. Although r_1 is not a boundary, the client can verify completeness since the re-constructed H_{root} matches the signature. Now consider that (i) the first non-token entry of the *VO* includes a hash value h_1 and (ii) the first record in the *VO* (after h_1) is r_2 (satisfying the query). The existence of h_1 implies that there are records preceding r_2. Therefore, the client detects that boundary records should be included, and their absence raises an alarm about possible violation of completeness.

Next, we extend REF to capture long-running queries on streams. Whenever there is a data modification, the DO alters its tree and forwards the update(s) to the SP in the form of a data stream, according to the *positive-negative* model [2]. The transmission of a new record *r* from the DO to the SP is denoted as (+<*r.id*, *r.k*>), and the deletion of an existing record as (-*r.id*). An update on *r* corresponds to a deletion (-*r.id*) followed by the insertion of the new values. 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, we say that 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.

Upon receiving an update from the DO, the SP modifies its own copy of the DMH-Tree accordingly. Then, it generates a new *VO* for *every* running query (by processing the query using *RangeDMH*) and sends it to the corresponding client. The client can reconstruct the signed root of the updated DMH-Tree and verify it using the DO's public key. Furthermore, using *LT* and *ST*, it can confirm that the results are current and there is no missing update. Note that temporal completeness in REF necessitates *VO* generation even for queries whose results are not affected by the update. We illustrate this through an example. Assume that at time $\tau=1$, a client *C* obtains a result. At $\tau=2$, the SP receives a new record r_1, but it does not inform *C*. At $\tau=3$, r_1 is deleted and a new tuple r_2 becomes part of the result. The SP sends to *C* a new *VO* including r_2, *LT*=3 and *ST*=2. *C* detects that there was an update at time 2, but it cannot determine if its query was affected or not. The only way that clients can be sure about the temporal completeness of their results, is if the SP transmits a new *VO* and signature to *every client* for *every timestamp that there is an update*.

Proof of temporal completeness: Suppose that at time τ the SP omits sending the *VO* for an update affecting the client's result.

[2] The proposed methods can also be used with sliding windows. We apply the positive-negative model since it is more general.

At a later time τ' the client receives a new *VO* from the SP. The client will detect the omission by noticing that the time of the previous update (included in the new *VO*) is $ST > \tau$. The only potential vulnerability regards the situation where 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* [LHKR06], according to which the DO revokes old signatures at periodic time intervals.

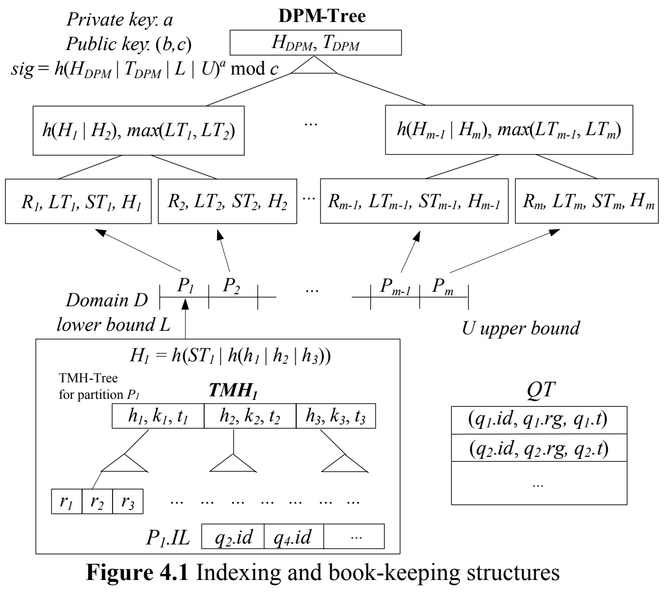
Although REF guarantees correctness and temporal completeness, it incurs *false transmissions* of *VO*s for queries whose result is not affected by the latest data updates. This imposes significant CPU cost to the SP (for computing the *VO*s) and to the clients (for verifying them). Furthermore, it leads to excessive network overhead. The proposed CADS method avoids these problems by integrating sophisticated indexing schemes and query processing algorithms.

4. Continuous Authentication on Data Streams

Section 4.1 summarizes the index structures utilized by CADS. Section 4.2 describes the initial result computation. Section 4.3 presents the monitoring algorithm for continuously updating the query results.

4.1 Indexing Scheme

Let *D* be the domain of the query attribute *r.k*. We decompose *D* into *m* disjoint partitions. Without loss of generality, we select *m* to be a power of two. Records are distributed into the partitions according to their search key (*k*) values. CADS includes two types of structures: (i) tuples in each partition are indexed by a *TMH-Tree* (*Temporal Merkle Hash-Tree*); (ii) all partitions are indexed by a *DPM-Tree* (*Domain Partition Merkle-Tree*). Figure 4.1 illustrates the indexing scheme. The intuition behind the framework is that fixed partitions are necessary to avoid false transmissions by localizing the effect of data updates. The embedded TMH-Tree in each partition alleviates the effects of skewness in the data. For instance, in the extreme case that all data fall in a single partition, CADS behaves similarly to REF.


Private key: a
Public key: (b,c)
$sig = h(H_{DPM} \mid T_{DPM} \mid L \mid U)^a \bmod c$
DPM-Tree
H_{DPM}, T_{DPM}
$h(H_1 \mid H_2), max(LT_1, LT_2)$
...
$h(H_{m-1} \mid H_m), max(LT_{m-1}, LT_m)$
R_1, LT_1, ST_1, H_1
R_2, LT_2, ST_2, H_2
$R_{m-1}, LT_{m-1}, ST_{m-1}, H_{m-1}$
R_m, LT_m, ST_m, H_m
Domain D
P_1 P_2 P_{m-1} P_m
lower bound L
U upper bound
$H_1 = h(ST_1 \mid h(h_1 \mid h_2 \mid h_3))$
TMH-Tree for partition P_1
TMH_1
h_1, k_1, t_1
h_2, k_2, t_2
h_3, k_3, t_3
r_1 r_2 r_3
$P_1.IL$
$q_2.id$ $q_4.id$
QT
$(q_1.id, q_1.rg, q_1.t)$
$(q_2.id, q_2.rg, q_2.t)$
Figure 4.1 Indexing and book-keeping structures

The TMH-Tree is a modified DMH-Tree that incorporates temporal information used by a virtual caching mechanism (to be discussed in Section 4.3). Specifically, every entry *e* in an intermediate node is a tuple (*e.h*, *e.k*, *e.p*, *e.t*), where *e.h*, *e.k*, *e.p*, have the same meaning as in the DMH-Tree (see Section 3), and *e.t* is a timestamp 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. Each partition *P* is associated with a tuple (*P.R*, *P.LT*, *P.ST*, *P.H*), where: *P.R* is a pointer to the root of the corresponding TMH-Tree indexing the tuples of *P*; *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 a hash value computed on the concatenation of *P.ST* with the hash value (H_{root}) of *P.R*.

The DPM-Tree is a binary tree that organizes hash values in a way similar to the MH-Tree. It is constructed bottom-up as follows. Each leaf node corresponds to a partition tuple (*P.R*, *P.LT*, *P.ST, P.H*). An adjacent pair P_i, P_{i+1} of leaves generates an internal node *N* at the next level that stores (*N.H*, *N.T*), where $N.H = h(P_i.H \mid P_{i+1}.H)$ and $N.T = max(P_i.LT, P_{i+1}.LT)$. The tree construction continues recursively in the same manner until the root. Intuitively, every internal node contains hashed information about the records in the partitions covered by its subtree, and the latest timestamp signifying updates in these partitions. Both the SP and the DO maintain the aforementioned authentication structures. Let H_{DPM} (T_{DPM}) be the hash value (timestamp) in the root of the DPM-Tree, and *L* (*U*) the lower (upper) bound of domain *D*. The owner computes $h(H_{DPM}, T_{DPM}, L, U)$, signs it (using its private key), and sends it to the SP, which keeps it locally (together with the above structures).

The indexing scheme can support 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 hash values 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 hashes and timestamps bottom-up.

CADS also maintains some book-keeping structures regarding the queries. In particular, the SP stores every running query *q* in a table *QT* as a record of the form (*q.id*, *q.rg*, *q.t*), where (i) *q.id* is a unique identifier, (ii) *q.rg* is the query range, and (iii) *q.t* is the timestamp of *q*'s last *VO* update. Each partition *P* is associated with an *influence list P.IL*, which stores the identifiers of the running queries that overlap with *P*. *QT* is organized as a hash table on *q.id* in order to support fast search for queries. Table 4.1 summarizes the notation, grouping symbols by category.

4.2 Initial Result Computation

The initial result computation corresponds to a snapshot authenticated query, i.e., the user can establish correctness, but does not need to verify temporal completeness. Given a new query *q*, the SP calls *RangeDPM*(*root*, *q*, *D*) shown in Figure 4.2, which performs a depth-first traversal of the DPM-Tree. Every node *N conceptually* corresponds to an interval *N.I*, which is the union of the partitions covered by the node's subtree (for the *root*

General symbols	
r: record	***r.id***: primary key of *r*
r.k: search key of *r*	***D***: domain of search key
***L*,(*U*)**: lower (upper) bound of *D*	***m***: number of partitions
P: partition	***P.IL***: influence list of *P*
P.R: root of TMH-Tree of *P*	***P.H***: hash value on *P.R* and *P.ST*
P.LT: time of last update in *P*	***P.ST***: time of second last update
TMH-Tree symbols	
n: TMH-Tree node	***e***: node entry
e.h: hash value in *e*	***e.k***: search key value in *e*
e.p: pointer to child node of *e*	***e.t***: time of last modification in *e*
DPM-Tree symbols	
sig: signature on H_{DPM}, T_{DPM}, *D*	***N***: DPM-Tree node
N.H: hash value in *N*	***N.T***: timestamp in *N*
$\boldsymbol{H_{DPM}}$: hash value in the root	$\boldsymbol{T_{DPM}}$: timestamp in the root
Query symbols	
q: query	***q.id***: unique identifier of *q*
q.rg: range of *q*	***q.t***: time of last *VO* creation for *q*

Table 4.1 Summary of symbols

N.I =[*L*,*U*]). If *q* does not overlap with *N.I*, the hash value *N.H* is inserted into the *VO*. Otherwise, *computeIntervals* (line 3) splits *N.I* into two equal intervals I_1 and I_2, corresponding to the two sub-trees of *N*, and the traversal continues recursively. When reaching a leaf node N_l, if *q* does not overlap with $N_l.I$, $N_l.H$ is included into the *VO*. Otherwise, $N_l.ST$ is inserted into the *VO* and *RangeTMH* is invoked, after expanding *q* (line 13) to include the boundary records, as discussed in Section 3. *RangeTMH* is similar to *RangeDMH* (in Figure 3.2) except that it adds to the *VO* a *dummy* value for each empty intermediate entry found during the traversal (the functionality of *dummy* values will become clear in Section 4.3). The tokens *begin_TMH* and *end_TMH* are appended to the *VO* to signify the *VO* components needed for reconstructing $N_l.H$. After the *VO* is generated, the SP inserts a new entry for *q* in *QT*, with *q.t* set to T_{DPM}. Finally, *q.id* is added to the influence lists (*IL*) of all partitions that overlap *q*.

```
RangeDPM(DPM_Node N, Query q, Interval I)
1.  If N is an intermediate node // in the DPM-Tree
2.      If q overlaps with I // i.e., N.I
3.          ($I_1$, $I_2$) = computeIntervals(I)
4.          Append [ to the VO
5.          RangeDPM(N.left_child, q, $I_1$)
6.          RangeDPM(N.right_child, q, $I_2$)
7.          Append ] to the VO
8.      Else append N.H to the VO
9.  Else // N is a leaf node that corresponds to a partition P
10.     Append begin_TMH to the VO
11.     If q overlaps with I
12.         Append N.ST to the VO // N.ST = P.ST
13.         q′ = ExpandQuery(q, N.R) // N.R is the root P.R of the
14.         Call RangeTMH(N.R, q′)   // TMH-Tree for partition P
15.     Else append N.H to the VO // q does not overlap with I
16.     Append end_TMH to the VO
```

Figure 4.2 Range query in the DPM-Tree

Figure 4.3 illustrates an initial *VO* generation for a query *q* with range [50, 75], assuming that *D*=[1, 80] and *m*=4. The SP starts by traversing the DPM-Tree. Since *q* does not overlap with $N_5.I$ (=[1, 40]), H_5 (i.e., $N_5.H$) is appended to the *VO*. The traversal continues with N_6 and reaches leaf N_3, corresponding to partition P_3. Since *q* overlaps with P_3, $N_3.ST$ (=$P_3.ST$) is appended to the *VO*. Then, the TMH-Tree of P_3 (TMH_3) is traversed to locate the left boundary record r_4 ($r_4.k$=48). Because *q* covers the right endpoint of P_3, it is not necessary to find its right boundary; hence, *q* is expanded to *q*′:[48, 75]. *RangeTMH* is called for TMH_3 with *q*′ as an argument. The entries in the root of TMH_3 are checked sequentially. Since the first entry does not overlap *q*′, h_1 is appended to the *VO*. On the contrary, node n_2 must be visited and its records (r_4, r_5) are inserted into the *VO*. A *dummy* value is appended in place of the third (empty) entry of n_3. Finally, the leaf N_4 of the DPM-Tree is visited and a partial *VO* is generated in a similar way, after appending $N_4.ST$ to the *VO* and expanding *q* to *q*′:[50, 76]. The complete *VO* is shown at the top of Figure 4.3. The SP sends the *VO* to the client, with *D*, T_{DPM} and *sig*.

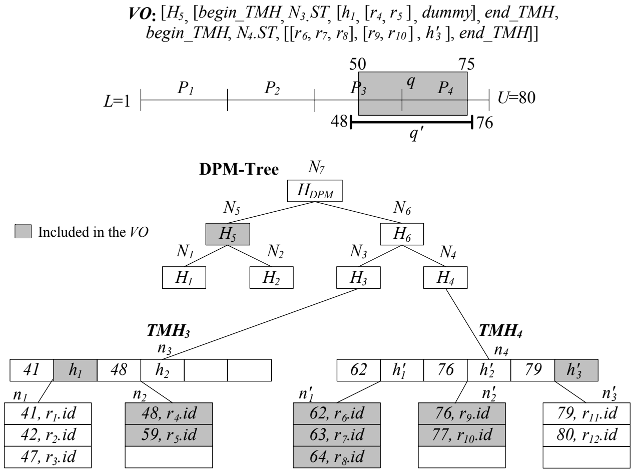


Figure 4.3 Example of initial result computation

Given the *VO* and *D*, the client verifies its correctness, by computing the hash value H_{DPM} at the root of the DPM-Tree using *Reconstruct*H_{DPM}(*VO*, *D*), shown in Figure 4.4. The functionality of the algorithm is similar to that of *Reconstruct*H_{root} (Figure 3.3), except that that *Reconstruct*H_{DPM} uses intervals to determine the extents of each partition on-the-fly. After the SP computes H_{DPM}, it hashes it with T_{DPM} and *D*, and matches it against the signature of the owner. The actual results are extracted during the verification process.

Proof of soundness: Suppose that a record is bogus or modified in partition *P*. Because the hash function is collision-resistant, the *P.H* value computed by the client is different than that of the owner. Therefore, the reconstructed H_{DPM} is also different from the original, and the signature verification fails.

Proof of completeness: Let *P* be a partition that overlaps with query *q*. If the partial *VO* corresponding to the TMH-Tree associated with *P* is included in the *VO*, the client can verify the completeness of the results residing in *P*, as shown in the proof of completeness for REF. The complication is how the client can determine that the *VO* actually contains components of *P's* sub-tree. For instance, a malicious SP can include only *P.H* in the *VO*, hiding potential results in *P* without affecting correctness. The client obtains the authenticated bounds (*D*:[*L*,*U*]) of the domain along with the *VO*. With this information, *Reconstruct*H_{DPM} computes the interval (*P.I*) covered by *P*, when the *begin_TMH* token that corresponds to *P* is encountered (lines 16-19). If *P.H* follows the token, then completeness is violated and the client is alarmed (lines 10-11).

```
$ReconstructH_{DPM}$ (VerificationObject VO, Interval I)
1.  Initialize an empty string buffer B
2.  Remove next entry E from VO
3.  If E is begin_TMH
4.      Remove next entry E from VO
5.      If E is an ST value
6.          Append E to B
7.          Append $ReconstructH_{root}(VO)$ to B
8.          Remove next entry E from VO // E is end_TMH
9.          Return hash(B)
10.     Else // E is a P.H value
11.         If the query overlaps with I, completeness is violated
12.         Else
13.             Remove next entry E' from VO // E' is end_TMH
14.             return E // i.e., P.H
15. If E is a hash value H, return H
16. If E is [
17.     $(I_1, I_2)$ = computeIntervals(I)
18.     Append $ReconstructH_{DPM}(VO, I_1)$ to B
19.     Append $ReconstructH_{DPM}(VO, I_2)$ to B
20. If E is ], return hash(B)
```

Figure 4.4 The algorithm for reconstructing H_{DPM}

The above discussion focuses on a single query. If there are several running queries in the system, the SP could process them independently, by calling *RangeDPM* for each query. This however, would lead to high processing cost due to multiple tree traversals. Instead, CADS applies *RangeDPM* only once, and checks each visited node against all running queries.

4.3 The Query Monitoring Algorithm

Considering that the initial result has been computed, we describe its continuous monitoring in the presence of data updates. Recall from Section 3 that, in order to achieve temporal completeness, REF performs false transmissions that lead to large communication overhead, high processing cost at the SP, and redundant verification effort at the clients. In the sequel, we present a solution that minimizes the false transmissions. Moreover, motivated by the observation that an updated *VO* shares common components with the previous one, we propose a *virtual caching mechanism* (*VCM*) that further reduces the communication cost. The term *virtual* is due to the fact that the SP does not store the *VO* for any query, which could lead to excessive memory consumption (proportional to the number of queries). Each client keeps in its own cache only a single *VO*.

When the SP receives a list of updates from the DO, it first determines the set of *affected partitions* in which at least one update occurs. Let *AQ* be the set of *affected queries* stored in the influence lists of these partitions. The SP will create new *VO*s only for the queries in *AQ* (as opposed to all queries for REF). Note that, depending on the granularity of the partitioning, false transmissions may still occur for queries that intersect an affected partition, without being influenced by the update(s). *VO* generation is performed by a modified version of *RangeDPM*. Specifically, when a node *N* is visited, its timestamp (*N.T*) is checked against *q*'s timestamp (*q.t*). Recall that (i) *N.T* is the time of the last update in any partition under *N*, and (ii) *q.t* is the time of the last update in the *VO* of *q*. If $q.t \geq N.T$, then all updates in *N* have been sent to the client during a previous transmission. Therefore, the *VO* components needed for reconstructing *N.H* are already present in the client's cache and up-to-date. A special token *Hit* is appended to the *VO* to signify that the client must retrieve these components from its own cache. Otherwise ($q.t < N.T$), the process is identical to the one used for the initial computation. Similar modifications apply to *RangeTMH*.

The SP sends the updated *VO* to the client along with a new signature and T_{DPM}. The client executes *CombineVO* (Figure 4.5) in order to merge the components contained in the updated *VO* (*newVO*) with the ones in the cache (*cachedVO*). The resulting *VO* is then stored in the client's cache (i.e., it becomes the new *cachedVO*). *CombineVO* scans the two *VO*s in parallel, retrieving an entry E_n (E_c) from *newVO* (*cachedVO*) at each step. An important invariant is that E_n and E_c must always correspond to the same item. The algorithm distinguishes four cases. If E_n and E_c have the same type (i.e., they are both hash values, records, dummies or tokens), E_n is appended to the new *VO* (lines 4-5). In the second case (lines 6-8), E_n is a non-token value and E_c is [. This implies that *newVO* contains updated information about the sub-tree starting at [. Therefore, E_n is added to *VO*, and all entries of *cachedVO* up to the matching] (signifying the end of the sub-tree) are deleted in order to retain synchronization between E_n and E_c. Lines 9-11 capture the reverse case, where a non-token value in *cachedVO* is replaced by a sub-tree in *newVO*. All entries between [and] in *newVO* that correspond to this sub-tree are inserted into *VO*. Finally (lines 12-16), if E_n is *Hit*, the matching value or sub-tree of *cachedVO* is appended to *VO*. With the new *VO*, the client recomputes H_{DPM} and verifies it against the new signature.

```
CombineVO (newVO, cachedVO)
1. Initialize VO to empty
2.  While newVO still has entries // also for cachedVO
3.      Remove next entry $E_n$ from newVO and $E_c$ from cachedVO
4.      If $E_n$ and $E_c$ have the same type
5.          Append $E_n$ to VO
6.      Else if $E_n$ is a hash or record or dummy value and $E_c$ is [
7.          Append $E_n$ to VO
8.          Remove all entries from cachedVO until matching ]
9.      Else if $E_n$ is [ and $E_c$ is a hash or record or dummy value
10.         Append $E_n$ to VO
11.         Remove all entries from newVO until matching ] and
            append them to VO
12.     Else if $E_n$ is Hit
13.         If $E_c$ is a hash value, append $E_c$ to VO
14.         Else if $E_c$ is [ or begin_TMH
15.             Append $E_c$ to VO
16.             Remove all entries from cachedVO until matching
                ] or end_TMH and append them to VO
17. Return VO
```

Figure 4.5 The *CombineVO* algorihtm

Figure 4.6 illustrates the concepts of monitoring and *VCM* by continuing the example of Figure 4.3, assuming that the initial result computation occurred at time τ =1 ($q.t$=1). The diagram also includes the timestamps inside the nodes and the entries. At τ =2 there is at least one change in P_2 ($N_2.T$=2), but since P_2 does not overlap with the query range, the SP does not perform *VO* generation and transmission. At τ =3, there are 3 deletions (of r_6, r_7 and r_8) and one update ($r_{10}.k$ changes from 77 to 74) in P_4, and one insertion of a new record r_n in P_1. Because P_4 intersects with the query, a new *VO* is generated. *RangeDPM* first visits the root of the DPM-Tree and then node N_5 ($N_5.T$ =3), whose interval does not overlap q. Since $N_5.T > q.t$, $N_5.H$ is different (due to the insertion of r_n) from the cached value and is appended to *newVO*. The traversal continues with N_6 and reaches leaf N_3. Because $N_3.LT = 1 = q.t$, all the components needed to reconstruct $N_3.H$ are already in *cachedVO* and a *Hit* token is added to *newVO*. Then, *RangeDPM* proceeds to N_4, where $N_4.LT$=3 > $q.t$. Thus, the corresponding TMH-Tree (TMH_4) must be traversed, after expanding q to q':[50,76].

Because in TMH_4 the three records (r_6, r_7, r_8) originally stored in leaf n'_1 are deleted, a merge operation has been performed between n'_1 and n'_2. This has reduced the number of entries in the parent node n_4, and a *dummy* value replaces the (deleted) first entry, which is appended to the *VO*. The timestamp of the second entry (3) is larger than $q.t$, which signifies that at least one update has occurred in n'_2 after $q.t$. Therefore, all its records are added to the *VO*. Finally, a *Hit* token is inserted for the third entry, because h'_3 has not been altered since τ=1. Note that *dummy* values are important for synchronization between the *newVO* and *cachedVO* during the execution of *CombineVO*.

$\tau = 1$ ***cachedVO*:** [H_5, [*begin_TMH*, $N_3.ST$, [h_1, [r_4, r_5], *dummy*], *end_TMH*, *begin_TMH*, $N_4.ST$, [[r_6, r_7, r_8], [r_9, r_{10}], h_3'], *end_TMH*]]

$\tau = 3$ ***newVO*:** [H_5, [*Hit*, *begin_TMH*, $N_4.ST$, [*dummy*, [r_{10}, r_9],*Hit*], *end_TMH*]]

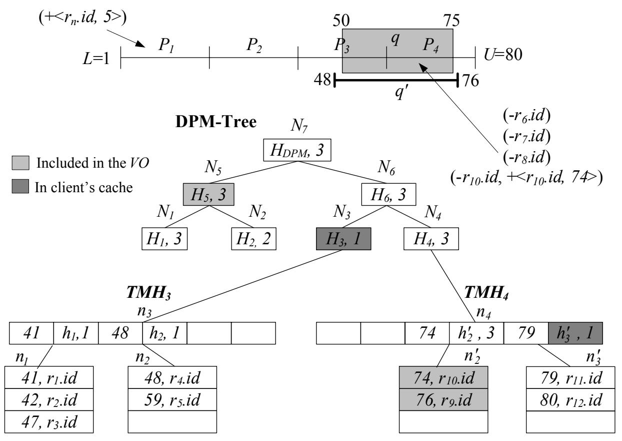


Figure 4.6 Query monitoring example

Proof of temporal completeness. Suppose that the initial computation of a query q occurs at a time τ and the *VO* is sent to the client. The client successfully verifies its correctness and stores it as *cachedVO*. Now assume that at τ' (>τ) 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 *sig* 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} (=τ). Since $P.ST > \tau$, 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.

5. MANAGING MULTIPLE OWNERS

All existing techniques in the database outsourcing literature consider the existence of a single owner. However, there is a variety of applications (recall the examples of Section 1) in which a SP collects multiple related datasets, each outsourced by a different DO. Let K be the number of DOs. A straightforward extension of CADS in such applications would be to maintain independent indexing schemes and generate separate *VO*s for each owner. We refer to this solution as CADS_K. The drawback of CADS_K is that the query processing cost at the SP and the network overhead for transferring the *VO*s are linearly correlated with K. In order to avoid this problem, we propose a version of CADS, called CADS_1, that integrates the data of all owners in a single authentication process with one *VO*, independently of the number of DOs. Section 5.1 describes CADS_1, and Section 5.2 compares it analytically with CADS_K.

5.1 CADS_1

The domain D and the number of partitions m are common for all owners and the SP. Each DO_j maintains its own data using an indexing scheme similar to the one of Figure 4.1. Let $P_{i,j}$ be the i-th partition at owner DO_j. The data in $P_{i,j}$ are indexed by a TMH-Tree $TMH_{i,j}$. All partitions are indexed by a *DPM+-Tree*, which is a modification of the DPM-Tree. Specifically, a leaf node corresponds to a partition tuple ($P_{i,j}.R$, $P_{i,j}.LT$, $P_{i,j}.ST$, $P_{i,j}.H$), where all attributes have the same meaning as in the DPM-tree. An adjacent pair $P_{i,j}$, $P_{i+1,j}$ of leaves generates an internal node N at the next level that stores ($N.H$, $N.T$), where $N.H$ = ($P_{i,j}.H \cdot P_{i+1,j}.H$) mod c, and $N.T = max(P_{i,j}.LT, P_{i+1,j}.LT)$. That is, the only difference between the two structures regards the hash values of intermediate nodes (in the DPM-Tree, $N.H = h(P_i.H \mid P_{i+1}.H)$). The modular product stored in the root of the DPM+-Tree of owner DO_j is $H_{DPMj}= \prod_{i=1}^{m} P_{i,j}.H$ (mod c).

Every owner DO_j signs H_{DPMj} and sends its signature sig_j to the SP along with its records. Note that sig_j does not entail any knowledge about the data of the other owners. However, the signing process involves some modifications with respect to CADS. Specifically, a trustworthy key distribution center assigns to each owner DO_j a private key a_j and a public key (b_j, c), i.e., all public keys have the same[3] component c. Given H_{DPMj}, DO_j creates $sig_j = h(H_{DPMj})^{a_j}$ mod c. A client can confirm the authenticity of H_{DPMj} by checking if sig^{b_j} mod c equals $h(H_{DPMj})$.

The SP maintains, for each partition P_i, the TMH-Trees of all owners separately. The hash value of P_i is $P_i.H = \prod_{j=1}^{\text{K}} P_{i,j}.H$ (mod c), where K is the number of owners. A single DPM+-Tree indexes all partitions. Specifically, the value $N.H$ in an internal node N is the modular product of the hash values in the child nodes. The product in the root of the DPM+-Tree is H_{DPM}= $\prod_{i=1}^{m} P_i.H$. Note that, due to the commutative and associative properties of modular multiplication, H_{DPM} can be written as: $\prod_{j=1}^{\text{K}}\prod_{i=1}^{m} P_{i,j}.H$ (mod c)= $\prod_{j=1}^{\text{K}} H_{DPMj}$ (mod c). This fact is exploited by the verification process at the client.

We describe the *VO* generation and verification algorithms of CADS_1 using Figure 5.1, where m=4 and there are two owners, DO_1 and DO_2. The query q overlaps partitions P_1 and P_2. For simplicity, we omit all temporal information from the example (and the structures) as its use is identical to the case of a single owner. Furthermore, we focus on a snapshot query, since the

[3] Note that using a single component c for all owners does not compromise the private key of any DO according to RSA.

monitoring mechanism is the same as CADS. The SP calls *RangeDPM*, which performs a pre-order traversal of the DPM+-Tree, reaching leaf node N_1. The two TMH-Trees that are associated with partition P_1 (one per owner) are visited and the two partial *VO*s ($VO\_TMH_{1,1}$ and $VO\_TMH_{1,2}$) are appended to the *VO*. Next, *RangeDPM* visits node N_2 and, similarly, adds $VO\_TMH_{2,1}$ and $VO\_TMH_{2,2}$ to the *VO*. Finally, because q does not overlap with N_6, H_6 is inserted into the *VO*. The SP transmits to the client the *VO*, D, as well as the signatures sig_1, sig_2 in a pre-determined order, which is known to the client.

VO: [[*begin_TMH*, $VO\_TMH_{1,1}$, $VO\_TMH_{1,2}$, *end_TMH*, *begin_TMH*, $VO\_TMH_{2,1}$, $VO\_TMH_{2,2}$, *end_TMH*], H_6]

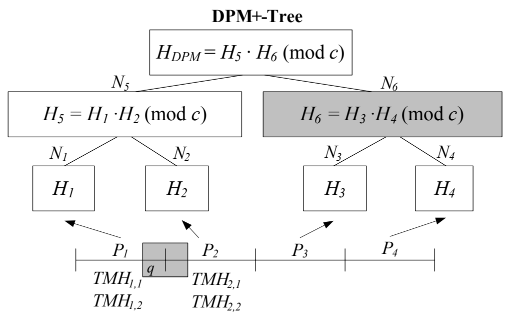


Figure 5.1 *VO* generation for multiple owners

The client first computes $P_{1,1}.H$ and $P_{1,2}.H$ from $VO\_TMH_{1,1}$ and $VO\_TMH_{1,2}$, respectively, using $ReconstructH_{root}$. Then, it performs their modular multiplication and produces $P_1.H$ (i.e., H_1). In a similar way, it computes H_2. H_5 is derived by multiplying H_1 with H_2. Finally, it reconstructs the product of the root as: $H_{DPM}= H_5 \cdot H_6$ (H_6 was received as part of the *VO*). The client also obtains (from the SP) $sig_1 = H_{DPM1}^{a_1} \bmod c$ and $sig_2 = H_{DPM2}^{a_2} \bmod c$, where a_1 and a_2 are the private keys of DO_1 and DO_2. Since $H_{DPM} = H_{DPM1} \cdot H_{DPM2} \bmod c$, in order to verify authenticity, it suffices to prove that the reconstructed H_{DPM} is equal to $sig_1^{b_1} \cdot sig_2^{b_2} \bmod c$, where (b_1,c) and (b_2,c) are the public keys of DO_1 and DO_2. In general for K owners, the results are sound, iff the re-constructed H_{DPM} equals $sig_1^{b_1} \cdot \ldots \cdot sig_K^{b_K} \bmod c$.

Proof of soundness: Let $S_{i,j}$ be the set of records in P_i owned by DO_j. A malicious SP can attack CADS_1 if it can find $m\cdot$K sets of records $S'_{i,j}$ such that: (i) the modular product of their respective $P'_{i,j}.H$ values is identical to H_{DPM} and (ii) there is at least one pair (i, j) satisfying $S'_{i,j} \neq S_{i,j}$. This, however, is impossible, due to collision-resilience of the hash function in the signature scheme. Now assume that the SP can determine $m\cdot$K values $P'_{i,j}.H$ (with at least one $P'_{i,j}.H \neq P_{i,j}.H$) such that their modular product yields H_{DPM}. Because the hash function is one-way, it is computationally infeasible for the SP to find $m\cdot$K sets of records $S'_{i,j}$ yielding these hash values. The proofs of completeness and temporal completeness are identical to the case of a single owner and omitted.

5.2 Analytical Comparison of CADS_1 and CADS_K

Similar to the single owner case, CADS_1 combines all queries in a single traversal of the DPM+-Tree that checks each visited node against every query. Therefore, it is faster than CADS_K in terms of query processing. Next, we compare the two methods analytically on space consumption, *VO* size and verification cost. Table 5.1 summarizes the symbols used in the analysis, as well as their typical values. 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]. S_h=20 bytes is the digest size of SHA1 [NIST95] (one-way, collision-resistant) hash function. S_m is the size of the modular product, which equals the signature size. We use RSA with 128 bytes signatures, which, currently is the minimum size that guarantees security.

Symbol	Meaning	Typical Value
C_v	CPU cost of *verify signature* operation	115 μsec
C_h	CPU cost of *hash* operation	1.1 μsec
C_m	CPU cost of *multiply* operation	44 μsec
S_h	size of a hash value	20 bytes
S_m	size of a modular product	128 bytes

Table 5.1 Symbols and values in the analysis

Let N be a node of the DPM+-Tree whose subtree covers an interval I of domain D. In CADS_K, the SP maintains K DPM-Trees and, therefore, there are K nodes that correspond to the same I, each storing a hash value of size S_h. In CADS_1, the SP maintains a single DPM+-Tree and stores one modular product of size S_m in N. In order for CADS_1 to start outperforming CADS_K in terms of memory consumption, the number of owners should be lower bounded by the following formula:

$$\text{K} \cdot S_h \geq S_m \Leftrightarrow \text{K} \geq \left\lceil \frac{S_m}{S_h} \right\rceil \tag{5.1}$$

Given the typical values S_h=20 bytes, S_m=128 bytes from table 5.1, CADS_1 incurs less memory consumption than CADS_K for K ≥ 7. The same lower bound applies to the *VO* size, because during the *VO* generation (i) in CADS_K the SP includes K hash values into the *VO* for every pruned node of the DPM-Tree, whereas (ii) in CADS_1, it includes one modular product for every pruned node of the DPM+-Tree.

Next we analyze the verification cost at the client, excluding the reconstruction of $P.H$ values, which is common in CADS_K and CADS_1. Let E_{over} be the number of partitions overlapping with the query, and $E_{h/m}$ the number of hash values (modular products) that are needed for reconstructing the root of the DPM-Tree (DPM+-Tree). In CADS_K, the client has to reconstruct K DPM-Tree root hashes by performing $E_{h/m}$ hash operations (each costing C_h = 1.1 μsec) per tree, and to verify all trees. Thus, the cost of verification in CADS_K is:

$$VC_\text{K} = \text{K} \cdot E_{h/m} \cdot C_h + \text{K} \cdot C_v \tag{5.2}$$

In CADS_1, the client has to reconstruct a single DPM+-Tree root with cost $E_{h/m} \cdot C_m$ (C_m = 44 μsec). Also for every partition P_i that overlaps with the query, it has to compute the hash value of the corresponding DPM+-Tree leaf by multiplying K $P_i.H$ values (one per owner) at a cost of $E_{over} \cdot$ (K-1) $\cdot C_m$. While verifying the reconstructed H_{DPM}, the client first raises each signature to the power of the respective owner's public key component b_j. The overhead of each exponentiation is almost equal to that of verifying a single signature. Therefore, the total cost of exponentiation is K $\cdot C_v$. Finally, the client multiplies the K signatures together at a cost of (K-1) $\cdot C_m$. Summing up the aforementioned factors yields the following equation for the total verification cost of CADS_1:

$$VC_1 = E_{h/m} \cdot C_m + E_{over} \cdot (\text{K}-1) \cdot C_m + \text{K} \cdot C_v + (\text{K}-1) \cdot C_m \tag{5.3}$$

Combining equations (5.2) and (5.3), we derive equation (5.4) that lower bounds the value of K, after which CADS_1 starts to outperform CADS$_K$ in terms of verification cost. Note that this equation does not take into account implementation issues. Specifically, CADS$_K$ executes functions *ReconstructH$_{DPM}$* and *CombineVO* K times, which involve numerous recursive calls. As we show in the experimental evaluation, CADS$_1$ outperforms CADS$_K$ even for small values of K.

$$K \geq \left\lceil \frac{C_m(E_{h/m} - E_{over} - 1))}{C_h E_{h/m} - C_m(E_{over} + 1)} \right\rceil \qquad (5.4)$$

6. Experimental Evaluation

We deployed REF and CADS on a P4 3GHz CPU with 2GBytes of RAM, using the Crypto++ library [Crypto]. Each record consumes 100 bytes and its search key ranges from 0 to 10^6. For generality, we repeat each experiment on two datasets: (i) in UNI the initial search key distribution is uniform; (i) in SKD the search keys follow the Zipfian distribution (with the skew parameter set to 0.8, so that 77% of the records fall in 20% of the data space). Each experiment is a simulation of 100 timestamps. At every timestamp, updates arrive at a rate *AR*. An update involves a deletion of a random tuple and an insertion of a new one with the same id but with a different search key. To produce the new value, a random number is generated in range [-10^3, $+10^3$] and added to the old one. Consequently, the dataset cardinality *DC* is constant at all times. We monitor *QC* running queries, which are uniformly distributed in the dataspace and their result set size is approximately 0.1% of *DC*. Finally, after a fine tuning step we set the number of partitions *m* of CADS to 8192 for both datasets. Table 6.1 summarizes the system parameters under investigation, along with their ranges and default values (per timestamp). Section 6.1 compares CADS against REF, and Section 6.2 evaluates CADS$_1$ and CADS$_K$ in the presence of multiple owners.

Parameter	Default	Range
Data cardinality (*DC*)	100K	10K, 50K, 100K, 200K, 500K
Query cardinality (*QC*)	1K	100, 500, 1K, 2K, 5K
Update rate (*AR*)	100	10, 50, 100, 200, 500

Table 6.1 System parameters

6.1 Single Owner

First, we assess the effect of the data cardinality *DC*, after setting the other parameters to their default values (*QC* = 1K, *AR* = 100). Figure 6.1 illustrates the total query processing time (milli-seconds) per timestamp at the SP. The overhead of REF is significantly higher because it has to process all the running queries. On the other hand, CADS re-evaluates only the queries whose result changes, plus a small number of queries that overlap affected partitions (although their results do not change). The cost of REF increases logarithmically with *DC* due to the DMH-Tree. CADS is not very sensitive to *DC* because of the virtual caching mechanism (VCM). Specifically, when a visited node has not been altered with respect to the previous transmission, the traversal of its sub-tree is entirely skipped.

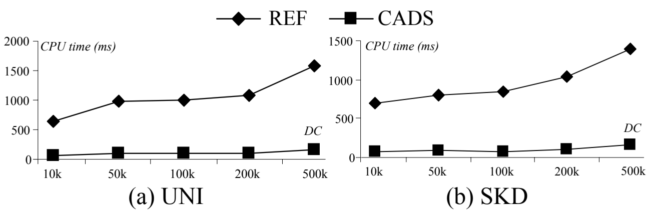


Figure 6.1 Query processing time vs. *DC*

Figure 6.2 shows the total *VO* size (Mbytes) for all queries, transmitted by the SP per timestamp as a function of *DC*. The communication overhead of REF increases linearly because, since the selectivity is fixed, the number of records in the result is linear to the cardinality. All these records are transferred to the client at each timestamp. In CADS, the growth of the result is absorbed by the VCM. Specifically, since *QC* and *AR* are fixed and independent of *DC*, the size of the result matters mainly for the first transmission. For SKD, the *VO* size of CADS increases slightly faster because of the unbalanced nature of the partitions. In particular, an update in a dense partition may invalidate a large part of the client's cache.

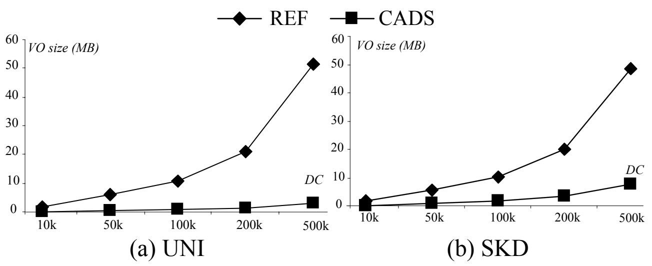


Figure 6.2 *VO* size vs. *DC*

Figure 6.3 depicts the verification time (milli-seconds) per timestamp at each client. REF imposes a heavy burden on the clients because, due to the false transmissions, the client must verify its query at every timestamp. Note that the performance gap between CADS and REF does not increase as fast as in the previous diagrams, because even if a partial *VO* is in the cache, the client still needs to combine it with the new *VO* components and match it against the signature.

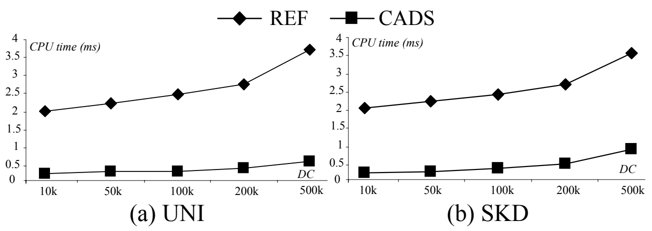


Figure 6.3 Verification time vs. *DC*

Figure 6.4 compares the two methods on memory consumption. REF and CADS consume about the same space, which is dominated by the records. Specifically, 75%-80% of the space overhead is due to the stored tuples and most of the rest due to the hash values. The additional information of CADS (query book keeping, influence lists, etc.) is negligible.

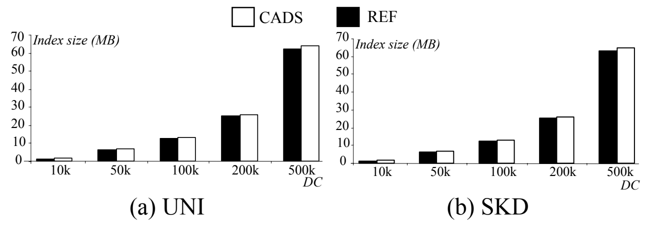
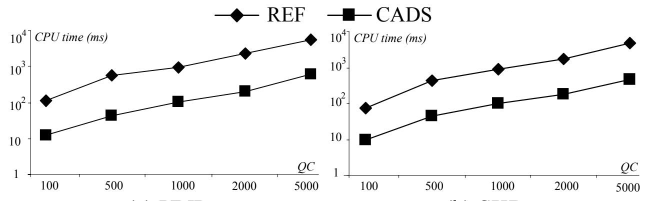


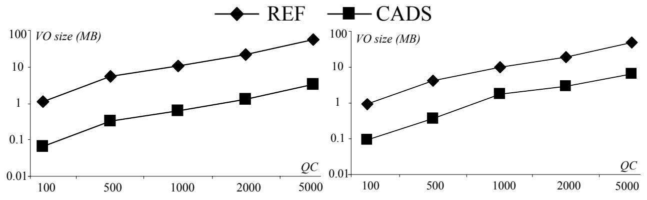
Figure 6.4 Index size vs. *DC*

The second set of experiments evaluates the effect of the query cardinality (*QC*) for *DC* = 100K, and *AR* = 100. The query processing cost of both methods increases due to different reasons. In REF, each query is evaluated at each timestamp. In CADS, the number of queries affected by an update (and therefore have to be re-evaluated) is proportional to *QC*. The *VO* size in Figure 6.6 follows similar trends for the same reasons.



(a) UNI (b) SKD

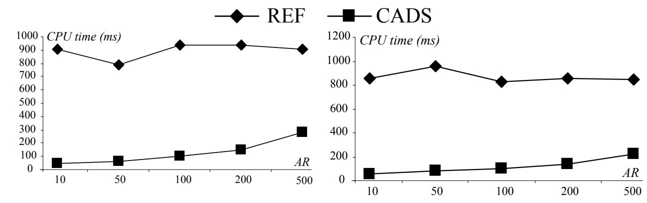
Figure 6.5 Query processing time vs. *QC*



(a) UNI (b) SKD

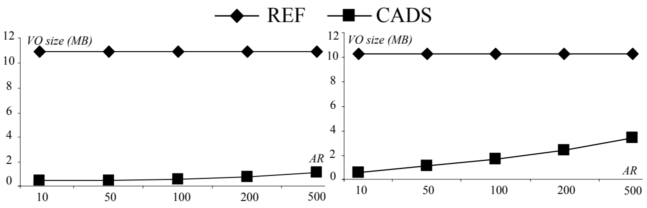
Figure 6.6 *VO* size vs. *QC*

The last set of experiments in this section assesses the effect of the update rate (*AR*), after fixing *DC* to 100K and *QC* to 1K. Figure 6.7 shows the total query processing cost per timestamp at the SP. REF is insensitive to the update rate because, in order to achieve temporal completeness, the SP has to generate and transmit a new *VO* for all the queries, independently of the number of updates. The fluctuations in its performance are caused by changes in the data distribution. On the other hand, the cost of CADS increases because the number of affected queries is proportional to *AR*. This is also reflected in the *VO* size of CADS, shown in Figure 6.8. As expected, *AR* does not affect the *VO* size of REF. Note that the cost of update handling at the SP is negligible compared to that of query processing. Specifically, both CADS and REF consume around 15ms for *AR* = 500, whereas, as shown in Figure 6.7, the corresponding query processing costs are about 300ms (for CADS) and 800ms (for REF).



(a) UNI (b) SKD

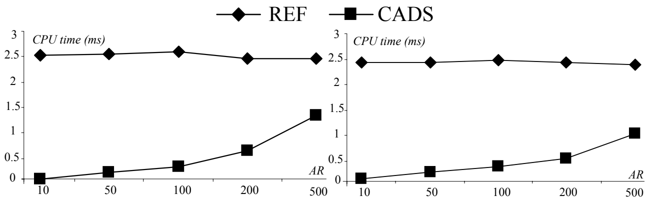
Figure 6.7 Query processing time vs. *AR*



(a) UNI (b) SKD

Figure 6.8 *VO* size vs. *AR*

Figure 6.9 depicts the verification time at the client. The diagrams are similar to those in Figures 6.7 and 6.8, except that the curves converge faster. This is due to the absence of the caching effect. Specifically, although the processing cost and the *VO* size are reduced by VCM, the client still has to verify the affected queries.



(a) UNI (b) SKD

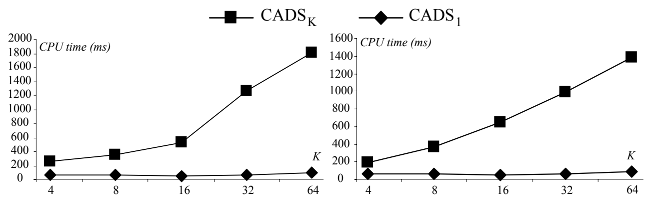
Figure 6.9 Verification time vs. *AR*

Summarizing this section, CADS exhibits considerably lower query processing time than REF, enabling the SP to serve numerous running queries without compromising the quality of service. It also incurs a significant reduction of the communication overhead, a fact that makes it suitable for wireless networks, and, in general, environments where transmission is expensive. Finally, CADS minimizes the verification burden, which is important for clients (i.e., PDAs) with limited resources.

6.2 Multiple Owners

In this section we compare CADS_1 with CADS_K varying the number of owners (K), after setting the other parameters to the default values (*DC* = 100K, *QC* = 1000, *AR* = 100). For each experiment, we divide the records equally among owners, so that each owner maintains approximately *DC*/K alive tuples per timestamp. Updates are split accordingly, i.e., each update originates from every DO with the same probability.

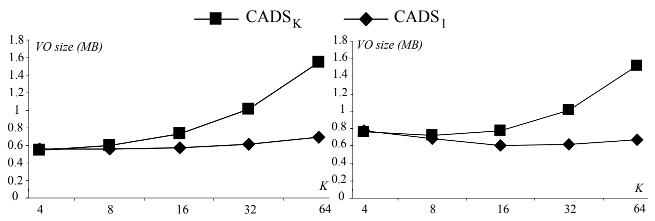
Figure 6.10 illustrates the total query processing time at the SP per timestamp. CADS_1 is better than CADS_K in all cases because it requires one traversal of a single DPM+-Tree. Recall that during this traversal, every visited node is checked against all running queries. On the other hand, CADS_K necessitates the traversal of K distinct DPM-Trees, one for each owner.



(a) UNI (b) SKD

Figure 6.10 Query processing time vs. K

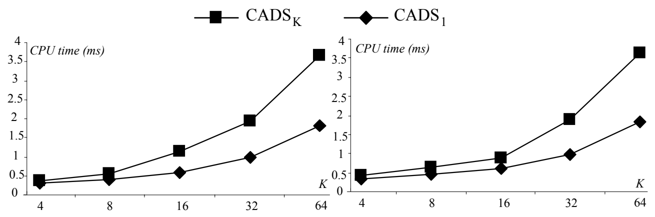
Figure 6.11 shows the total *VO* size transmitted per timestamp. For CADS_K the *VO* size is proportional to K due to the hash values of the distinct DPM-Trees (although the number of results is independent of K). CADS_1 is rather insensitive because K affects the *VO* overhead only for the TMH-Trees, but not for the DPM+-Tree. Recall that CADS_1 includes in the *VO* a modular product of size 128 bytes for each pruned DPM+-Tree node. Although this modular product is larger than a hash value (20 bytes), it is independent of K. Therefore, as predicted by the cost models of Section 5.2, CADS_1 starts outperforming CADS_K when K exceeds 7.



(a) UNI (b) SKD

Figure 6.11 *VO* size vs. K

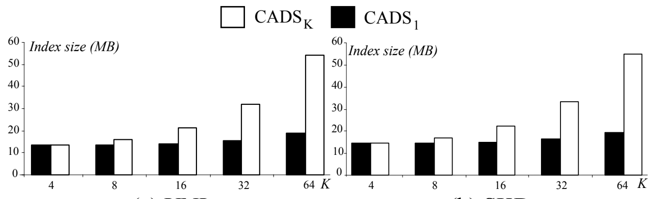
Figure 6.12 depicts the verification time per client per timestamp. The performance gain of CADS$_1$ with respect to CADS$_K$ increases with K, because in CADS$_1$ a client performs a single verification process independently of the value of K. The cost of this process increases with K, but not as fast as that of individual verifications.



(a) UNI (b) SKD

Figure 6.12 Verification time vs. K

Finally, Figure 6.13 depicts the index size at the SP. For 4 DOs, CADS$_1$ consumes slightly more space because the modular product is between 6 and 7 times larger than a hash value. However, the space overhead of CADS$_K$ increases with K because of the distinct DPM-Trees that must be maintained for each DO. Recall from Figure 6.4 that, although the authentication information is relatively small (20%-25%) compared to the total index size, when multiplied by a large value of K, it constitutes a considerable fraction of the total space consumption.



(a) UNI (b) SKD

Figure 6.13 Index size vs. K

Concluding this section, CADS$_1$ outperforms CADS$_K$ on all aspects and the performance gain increases with the number of DOs. It is worth pointing out that CADS$_1$ scales very well with K meaning that, in practice, it could be used in applications involving numerous DOs.

7. CONCLUSION

This paper constitutes the first work addressing continuous query processing and authentication on data streams. We assume a SP that collects information from one or more data owners and at the same time processes queries originating from numerous clients. The SP returns to the clients the query results, as well as verification information necessary to establish their correctness. In addition, the clients must be able to prove temporal completeness, i.e., that there is no result omission in-between subsequent updates. We first propose REF, a method that achieves these goals at the expense of false transmissions. To solve this problem, we introduce CADS, which reduces (i) the processing cost at the SP, (ii) the communication overhead between the SP and the clients, and (iii) the verification effort at the client. Finally, we extend CADS to multiple owners and show that substantial gains can be achieved by integrating the data of different owners in one index and a single verification process.

In the future, we plan to adapt our techniques to spatio-temporal data streams. In this setting, the SP (e.g., a location based service) collects the positions of continuously moving users from one or more owners (e.g., mobile phone operators). Clients (e.g., local businesses) issue spatial (e.g., range, nearest neighbor, etc.) queries to the SP and must be able to verify the results before contacting (e.g., sending offers, e-coupons, etc.) users in their vicinity. CADS could be applied after replacing the single-dimensional partitions with a multidimensional indexing scheme.

ACKNOWLEDGEMENTS

This work was supported by grant HKUST 6184/05E, from Hong Kong RGC.

REFERENCES

[BBD+02] Babock, B., Babu, S., Datar, M., Motwani, R., Widom, J. Models and Issues in Data Stream Systems, *PODS*, 2002.

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

[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.

[GO03] Golab, L., Öszu, T.M. Issues in Data Stream Management. *SIGMOD Record*, 32(2): 5–14, 2003.

[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.

[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.

[NIST95] National Institute of Standards and Technology. FIPS PUB 180-1: Secure Hash Standard. National Institute of Standards and Technology, 1995.

[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.

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

[RSA78] 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.