The VLDB Journal (2010) 19:161–180
DOI 10.1007/s00778-009-0145-2

REGULAR PAPER

Continuous authentication on relational streams

Stavros Papadopoulos · Yin Yang · Dimitris Papadias

Received: 17 September 2008 / Revised: 12 March 2009 / Accepted: 8 May 2009 / Published online: 6 June 2009

Abstract According to the *database outsourcing model*, a data owner delegates database functionality to a third-party service provider, which answers queries received from clients. *Authenticated query processing* enables the clients to verify the correctness of query results. Despite the abundance of methods for authenticated processing in conventional databases, there is limited work on outsourced data streams. 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 data updates. This paper presents a comprehensive set of methods covering relational streams. We first describe REF, a technique that achieves correctness and temporal completeness but incurs false transmissions, i.e., the provider has to inform the 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. In addition, we present an analytical study to determine the optimal indexing granularity, and extend CADS for the case that the data distribution changes over time. Finally, we evaluate the effectiveness of our techniques through extensive experiments.

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

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

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

Keywords Database outsourcing · Authentication · Continuous monitoring · Data streams

1 Introduction

In the traditional data management model, organizations have the entire responsibility for administering their databases. This entails purchasing hardware, deploying a dedicated DBMS, acquiring network bandwidth, and hiring professional personnel to run the system. *Database outsourcing* is a new paradigm that alleviates the above problems. It involves three types of entities: data owners (DOs), service providers (SPs), and clients. A DO delegates its database functionality to one (or more) SP with 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.

Authenticated query processing techniques guarantee the *soundness* and *completeness* of query results in outsourced systems. Soundness ensures that all the records returned to the client 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 outsourcing systems for conventional databases use the general framework of Fig. 1. The DO signs the data set (employing a public-key, digital signature cryptosystem) and transmits it along with the signature to the SP. The SP keeps the data and the

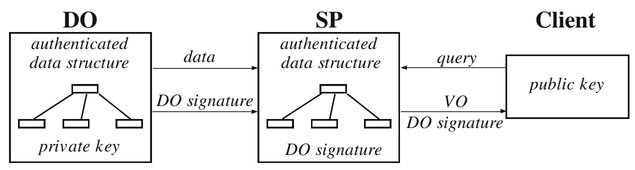

Fig. 1 Framework for outsourced databases

signature locally. In order to facilitate query processing, the data set 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 authentication information. The SP sends the *VO* and the corresponding signature to the client. The client verifies correctness by matching the signature against the *VO* and the public key of the DO.

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. However, the existing work in the database literature focuses on disk-resident and relatively static data sets. 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 verify their correctness.

The dynamic nature of the data and the potentially large number of long-running queries in stream environments 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 of the clients, and their verification effort. Third, 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:

- 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 the clients whenever there is a data update, even if their results are not affected.
- We propose CADS, which minimizes the processing and transmission overhead through an elaborate indexing scheme and a virtual caching mechanism. CADS and REF are main memory-based in order to achieve real-time query evaluation and fast structure updating.
- CADS utilizes a fixed partitioning of the data space. We analyze the effect of the partitioning granularity, and devise models for minimizing the size of the generated *VO*, which is the most important factor when measuring the performance of an outsourcing system.
- We develop an *adaptive* version of CADS (A-CADS) that dynamically updates the partitioning scheme to follow distribution changes in the data stream.
- We show through extensive experiments that CADS outperforms REF significantly in all aspects. Additionally, we confirm the accuracy of the analytical models, and demonstrate the gains of A-CADS in cases of continuously changing distributions.

The rest of the paper is organized as follows. Section 2 describes existing systems for database outsourcing. Section 3 presents REF. Section 4 focuses on CADS, whereas Sect. 5 analyzes its optimal partitioning granularity. Section 6 proposes A-CADS, and Sect. 7 experimentally evaluates all methods. Finally, Sect. 8 concludes the paper.

2 Related work

The *Merkle Hash Tree* (*MH-Tree*) [18] 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 digest of the binary representation of the record. The tree is constructed bottom-up, with each intermediate node storing the digest of the concatenation of the digests of its children. The digests are computed with a *one-way, collision-resistant hash function*, such as SHA-1 [21]. The DO signs the digest stored in the root of the tree, using a public-key, digital signature cryptosystem (e.g., RSA [25]).

Consider that a client asks for record r_3 in the MH-Tree of Fig. 2. The SP accesses the tree to locate the record. During the tree traversal, apart from record r_3, it inserts into the *VO* the digest stored in the sibling of every visited node (i.e., h_{12} and h_4). Having the *VO*, signature *sig* and the DO's public key, the client can verify the authenticity of the result by re-constructing the digest of the root as $h_{1234} = h(h_{12}|h(h(r_3)|h_4))$[1] and matching it against *sig*.

Devanbu et al. [8] utilize the MH-Tree for answering one-dimensional range queries, satisfying soundness and

[1] Henceforth, $h(\cdot)$ denotes a one-way, collision-resistant hash function, and '|' stands for concatenation.

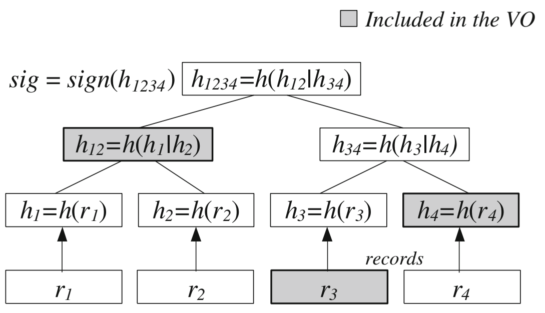

Fig. 2 Example of the Merkle Hash Tree

completeness. They also extend their method to multiple dimensions, combining the MH-Tree with the *Range Search Tree* [4]. Martel et al. [17] develop a generalized framework for creating efficient authenticated versions of a broad class of data structures. Goodrich et al. [10] introduce techniques for authenticating data structures that represent graphs and geometric objects. Tamassia and Triandopoulos [27] implement a distributed MH-Tree in order to provide data authentication over peer-to-peer networks. The authors in [23] build MH-Trees over *inverted indexes* [33] in order to guarantee correctness in keyword-based document retrieval.

The VB-Tree [24] is a B$^+$-Tree augmented with signed digests that ensures soundness, but not completeness. According to the *signature chaining* technique [20,22], the DO produces one signature for every triplet of adjacent tuples (sorted on their query attribute values). It then transmits its data set and signatures to the SP. Upon receiving a client query, the SP includes in the *VO* the result tuples (retrieved using a traditional B$^+$-Tree), the two *boundary* records that enclose them, and the corresponding signatures. Soundness is ensured by the signatures, and completeness by the existence of the boundary records.

Cheng et al. [5] apply signature chaining to devise authenticated versions of the KD-Tree and the R-Tree (called the *VKD-Tree* and the *VR-Tree*, respectively). *Signature aggregation* [19] can be used to reduce the communication and verification cost. This technique condenses multiple signatures into a single one, which can be verified almost as fast as an individual signature. Pang et al. [22] propose a solution for avoiding disclosure of boundary records, when the outsourced database must comply with certain access control policies.

The current state-of-the-art, disk-based ADS for authenticating one-dimensional range queries is the *Merkle B-Tree* (*MB-Tree*) [15]. The MB-Tree is basically a B$^+$-Tree that hierarchically organizes digests, in the same way as the MH-Tree. In addition to the actual result, the *VO* transmitted to the client contains two boundary records as well as a set of digests by which the client can re-construct the digest of the root. Yang et al. [30,31] present the *Merkle R-Tree* (*MR-Tree*), which allows authentication of spatial queries by combining concepts of the R-Tree and the MH-Tree.

There also exist approaches that do not involve the construction and maintenance of an ADS. Sion [26] assumes a *unified client model*, where the only client is the DO. The DO issues a batch of queries to the SP. It also produces and transmits a *challenge token*, which captures authenticated information about the result of a secret query q from the batch. The SP responds with the result sets of all queries and the id of q. This method probabilistically ascertains the DO about the correct execution of the batch. In [29] the DO incorporates fake tuples in the data set, encrypts all records, and transmits them to the SP. It also provides the clients with the function used to generate the fake tuples. The SP cannot distinguish the fake from the real records. The encryption scheme implicitly ensures soundness. The clients can probabilistically verify completeness by checking whether all fake tuples satisfying the query are present in the result set. Note that all clients are considered trusted, because otherwise the SP could collude with a client and obtain the fake tuple generator. The above schemes cannot be applied to our model, since we assume that the DO and the client are separate entities, and that the clients are not trusted. Atallah et al. [2] introduce a theoretical approach that provides lower asymptotic bounds for the *VO* size than MH-based techniques.

Several methods study *privacy preservation* of outsourced data. NetDB2 [13] assumes that the SP is trusted and takes protection measures against theft of data. The DO sends the data (in their original form) to the SP, which encrypts them. Before a query is processed, the SP decrypts the data in order to identify qualifying records. According to [12], the SP receives already encrypted data and (given a query) returns a superset of the actual result. Decryption and filtering of false hits are performed at the client's site. In the same context, Damiani et al. [6] propose a method where the client executes a sequence of queries that retrieve encrypted index nodes at progressively deeper levels. Agrawal et al. [1] introduce OPES, a scheme where the encrypted data preserve the original order. Therefore, an index can be constructed directly on the encrypted data. Wong et al. [28] enable mining of association rules on data encrypted with 1-to-n item mapping transformations. Kundu and Bertino [14] devise a technique that provides both integrity and privacy of records organized as trees (e.g., XML data). Our schemes focus only on the integrity of data. However, the above solutions can be used in combination with our framework.

Li et al. [16] deal with range query authentication in sliding windows, i.e., a tuple expires w time units after its arrival. In this case, all updates in the system correspond to insertions, and deletions are implicit. The method reduces the communication cost at the expense of delayed result updates. The basic idea is to use an input buffer B with a capacity of b tuples, where b is a system parameter. The SP simply inserts every new record into B without notifying the clients, until B is full. When this happens, the DO constructs a MH-Tree on the search key for these b tuples in B, and transmits the root signature to the SP. The latter evaluates all queries and sends result updates to clients. Tuple expirations are handled in a similar manner. The signing cost performed by the DO, as well as the network overhead for transmitting the signatures are amortized over b tuples. The penalty is that each result update is delayed by up to b tuples.

In our work we focus on real-time reporting, and we minimize the communication overhead between the SP and the clients by reducing false transmissions and using a caching mechanism. In this sense, our work is orthogonal to [16], i.e., a system could employ both techniques. Furthermore, although our methods can also be used under sliding windows, we apply the more general positive–negative model (i.e., tuples are explicitly deleted at any arbitrary instant, instead of according to their arrival order).

A second streaming technique [32] focuses solely on *aggregate queries*. The authors assume that *both* the client and the SP monitor a data stream transmitted by the DO. The client constructs and stores a small *synopsis* of the data locally. On the other hand, the SP maintains the entire data set. The client directs its queries to the SP, which processes them and returns the results. Using the synopsis, the client can probabilistically prove result correctness. In our context, only the SP receives the data stream from the DO (i.e., the client does not interact with the DO at all). Additionally, we focus on range query authentication, rather than aggregates.

3 REF

We present a *reference solution* (REF) for continuous query authentication on relational data streams that serves as a benchmark in our experimental evaluation. REF extends conventional authentication on the following aspects: (i) it modifies the MH-Tree to handle updates, and (ii) it incorporates a module for continuous monitoring of long-running queries that supports temporal completeness. Figure 3 outlines query processing in REF. Section 3.1 describes the indexing scheme of REF, whereas Sects. 3.2 and 3.3 focus on computing and maintaining the query results, respectively.

3.1 Indexing scheme

For simplicity, we consider that each tuple r in the DO's data set has only two attributes: the primary key $r.id$ and the search key $r.k$. Clients register long-running range queries on $r.k$ to the SP. In REF, both the DO and the SP sort the tuples of the outsourced data set on the search key and construct an authenticated structure, called the *DMH-Tree* (for *Dynamic Merkle Hash-Tree*). The DMH-Tree is a dynamic version of the MH-Tree. Each node in the DMH-Tree has 2 or 3 entries.

```
REF
// Initialization phase
1. Clients register their queries with the SP
2. The SP computes the initial VO of each query and
   sends it to the respective client
3. Each client verifies the soundness and completeness of
   its query results using the VO

// Monitoring phase
1. For each batch of updates
2.     The SP generates a VO for every running query and
       sends it to the corresponding client
3.     Each client verifies the soundness, completeness and
       temporal completeness of its VO
```

Fig. 3 General framework of REF

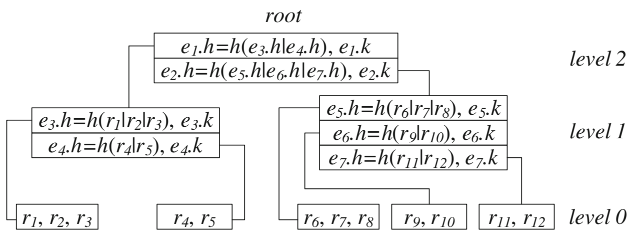
root
$e_1.h=h(e_3.h|e_4.h)$, $e_1.k$
$e_2.h=h(e_5.h|e_6.h|e_7.h)$, $e_2.k$
level 2
$e_3.h=h(r_1|r_2|r_3)$, $e_3.k$
$e_4.h=h(r_4|r_5)$, $e_4.k$
$e_5.h=h(r_6|r_7|r_8)$, $e_5.k$
$e_6.h=h(r_9|r_{10})$, $e_6.k$
$e_7.h=h(r_{11}|r_{12})$, $e_7.k$
level 1
r_1, r_2, r_3
r_4, r_5
r_6, r_7, r_8
r_9, r_{10}
r_{11}, r_{12}
level 0

Fig. 4 Example of a DMH-Tree

Figure 4 illustrates an example DMH-Tree. Each leaf node (level 0) contains 2 or 3 records. For intermediate nodes, each entry e is a triplet $(e.h, e.k, e.ptr)$, where $e.k$ is the search key of the first record in the subtree of e, and $e.ptr$ 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 $e.ptr$; for the upper levels, $e.h$ is computed on the concatenation of the digests of the entries in $e.ptr$. 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 digests contained in the root of the tree, e.g., in the example of Fig. 4, $H_{root} = h(e_1.h|e_2.h)$. Then it applies its private key to sign H_{root}, using the RSA public key cryptosystem. The SP stores a copy of this signature.

The DMH-Tree supports fast (i.e., logarithmic) updates,[2] 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

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

```
**RangeDMH** (*DMHNode n*, *ExpandedQuery 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
5.          $RangeDMH(e.ptr, q)$
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 $VO$
```

Fig. 5 Range query in the DMH-Tree

```
**Reconstruct$_{Hroot}$** (*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 $hash(B)$
```

Fig. 6 Algorithm for re-constructing H_{root}

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, i.e., updates that are reported simultaneously in batch. First, the structure is modified to accommodate all updates, without altering any digest, but temporarily marking the visited paths. Then, the marked paths are re-visited and the digests are computed bottom-up. In this way, the hash computations are performed only once.

3.2 Initial result computation

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 Fig. 5 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 digest of every pruned entry, (ii) the tuples in every visited leaf node, (iii) special tokens [and] indicating the scope of a node. Consider for example a query that retrieves records r_5 to r_8 in Fig. 4. The expanded query covers tuples r_4 to r_9. The application of *RangeDMH* to the expanded query yields the *VO*: $[[e_3.h[r_4, r_5]] [[r_6, r_7, r_8][r_9, r_{10}]e_7.h]]$. Note that the tokens in the *VO* reveal the tree structure, e.g., $[e_3.h[r_4, r_5]]$ corresponds to the first root entry and the remainder to the second one. The SP transmits the *VO* and the DO's signature to the client.

The verification process at the client utilizes the tree-structure information, encapsulated in the *VO*, to compute the digest H_{root} of the root. Figure 6 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 (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 $H_{root} = h(B)$. 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.

In this work, we consider the case that clients can issue queries freely without constraints. Nevertheless, the solution of [22] can be applied in conjunction with the proposed methods to avoid disclosure of boundary records, when the outsourced database must comply with certain access control policies. The proofs of soundness and completeness for initial result computation are identical to those of the MB-Tree [15] and omitted.

3.3 Query monitoring

Next, we discuss how REF captures *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. 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 followed by an insertion. In addition to the actual data, each transmission contains a new 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 re-construct 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. Observe 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 transmits to C a new *VO* including r_2, $LT = 3$ and $ST = 2$. Note also that the SP must send the correct *ST* and *LT* since they are incorporated in the DO's signature. C detects that there was an update at time 2, but it cannot determine if its query was affected or not (if r_1's key were within the client's range, the SP should have sent a *VO* with the new result). The only way that clients can establish temporal completeness of their results, is if the SP transmits a new *VO* along with the DO's signature and timestamps *LT* and *ST* to *every client* for *every timestamp that there is an update*.

Proof of temporal completeness Suppose that at time τ the client receives a *VO* from the SP. The client stores τ locally. Then at time $\tau' > \tau$ an update occurs that affects the query result, and the SP cheats by omitting transmitting the corresponding *VO*. After that, another update occurs and the SP sends a *VO* to the client, including timestamp *ST* (i.e., the timestamp of the second last update). Note that $ST \geq \tau'$ (multiple omissions may have occurred after τ'). The client compares *ST* with the local τ, and finds that $ST > \tau$. Note also that τ is the last timestamp before τ' at which the client received a *VO* from the SP. Therefore, the client detects that it did not receive a *VO* at *ST*. Meanwhile, if the SP sends a falsified *ST*, the client is alarmed since the DO's signature incorporates *ST*.

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* [15], according to which the DO publishes its revoked signatures on a public web site (called *directory*). A suspicious client may browse the directory and check if the lastly received signature is obsolete, in which case the SP has maliciously omitted sending a new *VO*.

4 CADS

Let D be the domain of the query attribute. CADS decomposes D into disjoint partitions, and distributes records into the partitions according to their search key values. This minimizes the number of false transmissions necessary to guarantee temporal completeness, by localizing the effect of updates. In particular, the SP needs to generate and send a new *VO* to a client, *if and only if* the client's query overlaps with the partition where an update occurs. Moreover, a virtual caching mechanism reduces the *VO* size, by utilizing results previously received by the client. Figure 7 overviews query processing in CADS. Section 4.1 summarizes the index structures, Sect. 4.2 describes the initial result computation and Sect. 4.3 presents the monitoring algorithm.

```
CADS
// Initialization phase
1-3. Same as lines 1-3 in REF (Figure 3)
4.   Each client stores its initial VO in cache

// Monitoring phase
1.  For each batch of updates
2.    The SP generates a VO only for the queries overlapping partitions that are affected by updates
3.    Each client C receiving a VO combines the cached VO with the new one to form the complete VO.
4.    C verifies the soundness, completeness and temporal completeness of the complete VO.
5.    C substitutes the VO in the cache with the complete one.
```

Fig. 7 General framework of CADS

4.1 Indexing scheme

For the following discussion, we assume that the partitions have equal length, and that their number m is a power of two. In Sect. 5 we explain how m affects the performance of CADS, and introduce cost models for computing its optimal value. CADS includes two types of structures: (i) tuples in each partition are indexed by a *TMH-Tree* (*Temporal Merkle Hash-Tree*). This structure enables the SP to efficiently process the part of the query that overlaps with the corresponding partition, and generate a *VO* with small size. (ii) All partitions are indexed by a *DPM-Tree* (*Domain Partition Merkle-Tree*). This enables the DO to produce a single signature (on the root of the DPM-Tree) in order to cryptographically "mark" the entire data set. Alternatively, the DO could sign the root of the TMH-Tree in every partition, which would lead to increased processing and communication cost for all parties involved, and high verification cost for the clients.[3] Moreover, the DPM-Tree reduces the size of the generated *VO*. Figure 8 illustrates the indexing scheme.

The TMH-Tree is a modified DMH-Tree that incorporates temporal information used by the virtual caching mechanism (discussed in Sect. 4.3). Specifically, every entry e in an

[3] The disadvantages of multiple signature generation over a single data set are discussed in [15].

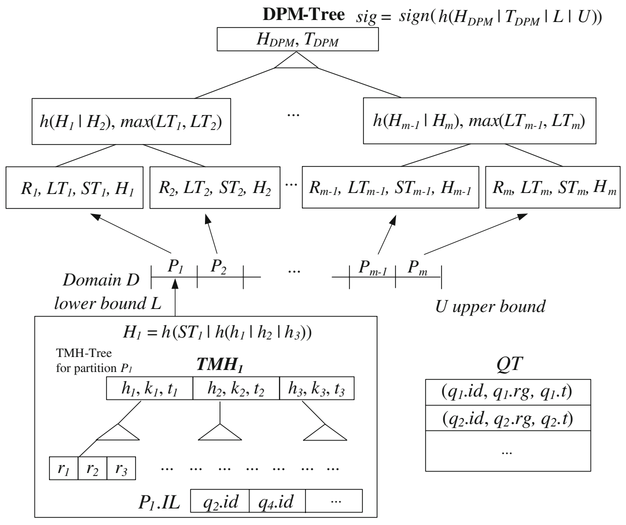

Fig. 8 Indexing and book-keeping structures

Table 1 Summary of CADS symbols

General symbols	
r: record	***r.id***: primary key of *r*
r.k: search key of *r*	***D***: domain of search key
L,(U): *D*'s lower (upper) bound	***m***: number of partitions
P: partition	***P.IL***: influence list of *P*
P.R: root of TMH-Tree of *P*	***P.H***: digest 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: digest in *e*	***e.k***: search key value in *e*
e.p: pointer to child node of *e*	***e.t***: time of last *e*'s modification
DPM-Tree symbols	
sig: signature on H_{DPM}, T_{DPM}, *D*	***N***: DPM-Tree node
N.H: digest in *N*	***N.T***: timestamp in *N*
$\boldsymbol{H_{DPM}}$: digest 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 *q*'s *VO* creation

intermediate node is a tuple ($e.h$, $e.k$, $e.ptr$, $e.t$), where $e.h$, $e.k$, $e.ptr$ have the same meaning as in the DMH-Tree (see Sect. 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 digest computed on the concatenation of $P.ST$ with the digest (H_{root}) of $P.R$. Note that if a TMH-Tree is empty, its H_{root} is simply the digest of a special null value (i.e., $H_{root} = h(null)$).

The DPM-Tree is a binary tree that organizes digests 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 intermediate node N at the next level that stores ($N.H$, $N.T$), where $N.H = h(P_i.H|P_{i+1}.H)$ 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 intermediate 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 authenticated structures. Let H_{DPM} (T_{DPM}) be the digest (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 Sect. 3, without altering any digest or timestamp, 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 re-visited 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.

CADS also maintains 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 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 Fig. 9, 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* $N.I = [L, U]$). If q does not overlap with $N.I$, the digest $N.H$ is inserted into the *VO*. Otherwise, *computeIntervals* (line 3) splits $N.I$ into two equal

```
RangeDPM (DPMNode 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 corresponding 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)
14.         Call RangeTMH(N.R, q′)
15.     Else append N.H to the VO    // q does not overlap I
16.     Append end_TMH to the VO
```

Fig. 9 Range query in the DPM-Tree

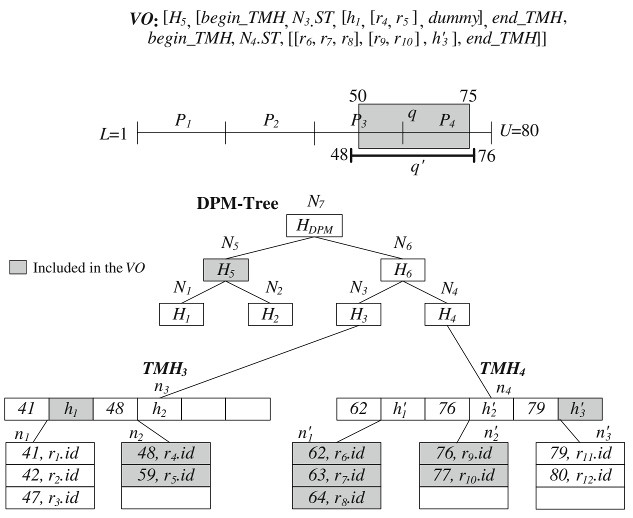

Fig. 10 Example of initial result computation

intervals I_1 and I_2, corresponding to the two subtrees 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 Sect. 3.2.

RangeTMH is similar to *RangeDMH* (in Fig. 5) 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 Sect. 4.3). Moreover, it inserts a *null* value if the TMH-Tree is empty. Tokens *begin_TMH* and *end_TMH* are appended to the *VO* to signify the *VO* components needed for re-constructing $N_l.H$. 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.

Figure 10 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 visited 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, 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*

```
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
12.             Report that completeness is violated
13.         Else
14.             Remove next entry E′ from VO // E′=end_TMH
15.             Return E // i.e., P.H
16. If E is a hash value H, return H
17. If E is [
18.     (I_1, I_2) = computeIntervals(I)
19.     Append ReconstructH_DPM(VO, I_1) to B
20.     Append ReconstructH_DPM(VO, I_2) to B
21. If E is ], return hash(B)
```

Fig. 11 Algorithm for re-constructing H_{DPM}

and expanding q to q':[50, 76]. The complete *VO* is shown at the top of Fig. 10. The SP sends the *VO* to the client, with D, T_{DPM} and *sig*.

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 *ReconstructH*$_{DPM}$(*VO*, D), shown in Fig. 11. The functionality of the algorithm is similar to that of *ReconstructH*$_{root}$ (Fig. 6), except that *ReconstructH*$_{DPM}$ 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 DO. 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 DO. This modification propagates up to the DPM-Tree root digest H_{DPM}, which is also different from the original. Consequently, the signature verification fails.

Proof of completeness Let P be a partition that overlaps with query q. We distinguish two cases: (i) the partial *VO* corresponding to the TMH-Tree associated with P is included in the *VO*. Then, the client can verify the completeness of the results residing in P, in a similar way to [15]. (ii) The SP includes only $P.H$ instead of the partial *VO* for P in the *VO*. This way it can hide potential results in P, while the client can still re-construct the H_{DPM} value that matches the DO's signature. The client detects that the *VO* should not contain $P.H$ as follows. Recall that the client obtains the bounds $(D : [L, U])$ of the domain along with the *VO*, which are also incorporated in the signature. With this information, the client computes the interval $(P.I)$ covered by P (during execution of *ReconstructH*$_{DPM}$), and finds that $P.I$ overlaps with the query. Therefore, the *VO* should contain detailed information about the respective TMH-Tree, rather than $P.H$. In both the above cases, the client is alarmed.

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 Query monitoring

Considering that the initial result has been computed, we describe its continuous monitoring in the presence of data updates. Recall from Sect. 3.3 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 re-constructing $N.H$ 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* (Fig. 12) 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

```
CombineVO (VerificationObject VO, Interval I)
1. Initialize VO to empty
2. While newVO still has entries // also for cachedVO
3.   Get 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 digest or record or dummy 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 digest or record or dummy
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 digest, 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
```

Fig. 12 *CombineVO* algorithm

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 digests, 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 subtree starting at [. Therefore, E_n is added to *VO*, and all entries of *cachedVO* up to the matching] (signifying the end of the subtree) 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 subtree in *newVO*. All entries between [and] in *newVO* that correspond to this subtree are inserted into *VO*. Finally (lines 12–16), if E_n is *Hit*, the matching value or subtree of *cachedVO* is appended to *VO*. With the new *VO*, the client re-computes H_{DPM} and verifies it against the new signature.

Figure 13 illustrates query monitoring and *VCM* by continuing the example of Fig. 10, assuming that the initial result computation occurred at time $\tau = 1 (q.t = 1)$. The diagram also includes the timestamps inside the nodes and the entries. At $\tau = 2$ 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 $\tau = 3$, 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 re-construct $N_3.H$ 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 $\tau = 1$. Note that *dummy* values are important for synchronization between the *newVO* and *cachedVO* during the execution of *CombineVO*.

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 stores it as *cachedVO*, along with τ. Now assume that at later time τ' ($>\tau$) one (or more) update(s) takes place in some partition P that overlaps with q. The SP cheats and 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*. We distinguish two cases: (i) *newVO* contains a partial *VO* corresponding to P. According to our algorithms, *newVO* also includes *P.ST*. The client compares *P.ST* with τ. Since at least a potential result update (at *P.ST*) was omitted, $P.ST > \tau$ 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 re-constructs a false $P.H$ value and the verification of the signature fails.

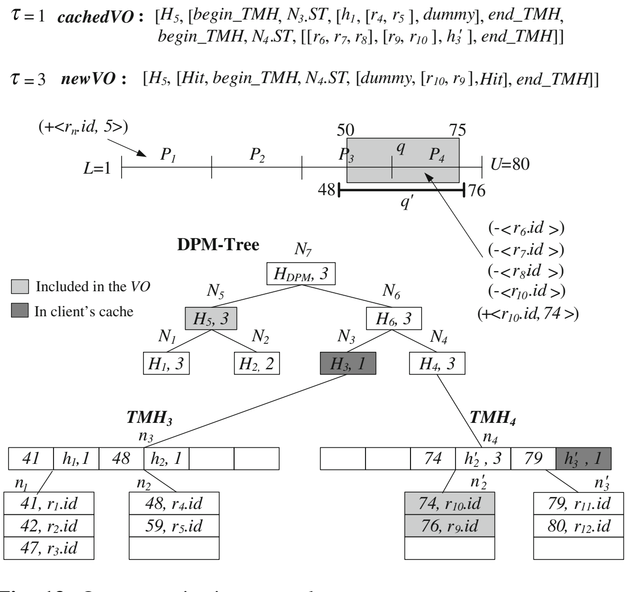

Fig. 13 Query monitoring example

5 Optimal granularity computation

Recall that in order to reduce false transmissions, CADS decomposes the domain into partitions organized into a DPM-Tree. The granularity m of this partitioning affects the efficiency of CADS. On the one hand, if m is too coarse (i.e., there are very few partitions), the ability of CADS to decrease false transmissions is subdued; in the extreme case that there is only one partition, CADS reduces to REF. On the other hand, a large number of partitions leads to a tall DPM-Tree and numerous TMH-Trees. This adversely affects performance, especially considering that most practical data sets exhibit a certain degree of skewness. Consequently, many of the partitions may contain few or no records at all. Manually (i.e., empirically) tuning m at the DO in order to maximize performance is both costly and error-prone. Therefore, in this section we establish rigorous cost models for CADS, which

Table 2 Summary of symbols in the analysis

Symbol	Meaning
m	Partitioning granularity
EVO	Expected VO size of a query
$EVO_{init}(q)$	Expected size of q's initial VO
$EVO_{upd}(q)$	Expected size of VO generated for q due to an update
$EVO_D(q)$	Expected size of VO generated for q in the DPM-Tree
$EVO_T(q)$	Expected size of VO generated for q in the TMH-Trees
NU	Number of timestamps when insertions/deletions occur
$\|U\|$	Expected number of insertions/deletions per timestamp
QS	A random sample set of queries
l_q	Extent of query q
$\|q\|$	Number of records contained in the query q
qp	Number of partitions the query q overlaps with
f	Expected fanout of the TMH-Tree
S_h	Size of a digest
S_r	Size of a record
S_t	Size of a timestamp
S_s	Size of a signature

we utilize to compute the best value of m. Table 2 summarizes common symbols used throughout this section.

Our analysis focuses on the expected *VO* (*EVO*) size for a query, 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 [7]). 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 re-construct the root digest. Hence, minimization of the *EVO* improves performance on these metrics as well.

Initially, we concentrate on the case that all partitions have the same length and discuss variable-length partitioning later. Without loss of generality, we normalize the search key values of the data space to [0, 1]. In order to keep the analysis tractable, we make the following simplifying assumptions. (i) The updates follow the distribution of the initial data set, 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 re-build the structures of CADS accordingly. (ii) Each query q has expected length $l_q \in (0, 1]$. (iii) We disable the virtual caching mechanism (*VCM*). In Sect. 7, we explain why the effects of the *VCM* are not significantly influenced by the partitioning granularity m.

In order to estimate *EVO*, we draw a random sample query set *QS*, e.g., from a past query log, or a known distribution. Let $q \in QS$ be a sample query, and $|QS|$ the cardinality of *QS*. CADS entails an initial *VO* computation for q, as well as the production 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 number of timestamps *NU* that involve updates, *EVO* is computed by:

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

Regarding $EVO_{init}(q)$, CADS includes in the *VO* five types of information: (i) the result set of q, (ii) two boundary records for 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, used by the client to verify correctness and, finally, (v) the signature of the DO. We do not consider the tokens (e.g., *begin_TMH*) 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 [3], histograms [11], probabilistic models [9]). 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 $1/m$, the expected value for qp is $\lfloor m \cdot l_q \rfloor + 1$. Regarding (iv), we use symbols $EVO_D(q)$ and $EVO_T(q)$ to denote the total size of the digests appended to the *VO* when traversing the DPM-Tree and *all* the TMH-Trees, respectively, during the processing of q. Finally, (v) equals the size of one signature (let S_s). Summarizing, $EVO_{init}(q)$ is given by:

$$\begin{aligned} EVO_{init}(q) = (|q| + 2) \cdot S_r + qp \cdot S_t \\ + EVO_D(q) + EVO_T(q) + S_s \end{aligned} \tag{2}$$

We next clarify $EVO_D(q)$ and $EVO_T(q)$. Figure 14 illustrates the traversal of the DPM-Tree and TMH-Trees for query q. For the DPM-Tree, the traversal starts from the root, it follows a single path until the *path split node*, and diverges into two paths. The two paths reach the leaf nodes, whose

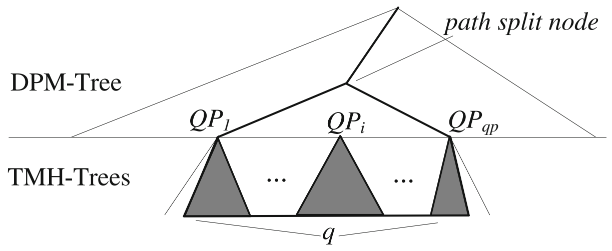

Fig. 14 CADS traversal

corresponding partitions contain the left and right boundary records of q, respectively. Since the DPM-Tree is binary, for each node on the single path, exactly one digest (i.e., that of the pruned sibling) is inserted to the *VO*. We thus focus on calculating the number of such nodes. Since the DPM-Tree indexes m partitions, its height is $\lfloor \lg m \rfloor+1$. Similarly, because the query overlaps *qp* partitions, the path split node is expected to be $\lfloor \lg qp \rfloor$ +1 levels above the leaves. Therefore, assuming that S_h is the size of a digest, $EVO_D(q)$ is:

$$\begin{aligned} &EVO_D(q) \\ &\quad = S_h \cdot ((\lfloor \lg m \rfloor - \lfloor \lg qp \rfloor) + 2 \cdot (\lfloor \lg qp \rfloor + 1)) \end{aligned} \tag{3}$$

$EVO_T(q)$ involves the digests inserted to the *VO* during the traversal of *qp* TMH-Trees, each of which corresponds to a partition overlapping with q. Let $QP_1, QP_2, \ldots, QP_{qp}$ be the partitions that intersect q. A key observation, as depicted in Fig. 14, is that all partitions except for the first (QP_1) and the last (QP_{qp}) are completely contained in q, meaning that during their corresponding TMH-Tree traversal, no digests are appended to the *VO* at all. On the other hand, for QP_1 and QP_{qp}, a root-to-leaf path of the corresponding TMH-Tree is traversed respectively, during which, the digests of the siblings of every visited node are included in the *VO*.[4] Let f ($2 \le f \le 3$) be the expected node fanout of a TMH-Tree, and $|P|$ be the number of tuples in partition P. The height of the TMH-Tree corresponding to P is then $\lfloor \log_f |P| \rfloor + 1$. For a particular q, $|QP_i|$ ($1 \le i \le qp$) is known and its EVO_T is calculated by:

$$\begin{aligned} EVO_T(q) = S_h \cdot \big(&\lfloor \log_f |QP_1| \rfloor + 1 + \\ &+ \lfloor \log_f |QP_{qp}| \rfloor + 1\big) \cdot (f-1) \end{aligned} \tag{4}$$

Combining Eqs. 2, 3 and 4 yields the complete model for $EVO_{init}(q)$. We next derive $EVO_{upd}(q)$. Recall that in CADS, the SP sends a new *VO* only when at least one update happens in a partition intersecting with q. Let $Prob_{VO}(q)$ denote the probability that the SP transmits a new *VO* (i.e., q is affected by any one of these updates). $EVO_{upd}(q)$ is obtained as follows:

$$EVO_{upd}(q) = EVO_{init}(q) \cdot Prob_{VO}(q) \tag{5}$$

Next we focus on $Prob_{VO}(q)$. According to the assumption that the updates follow the same distribution as the initial data set, the probability that an update falls in any one of $QP_1, QP_2, \ldots, QP_{qp}$ is $\Sigma_i |QP_i| / \Sigma_j |P_j|$, $1 \le i \le qp$, $1 \le j \le m$. Therefore, for a batch of $|U|$ independent update operations (i.e., insertions or deletions) occurring at a specific timestamp, $Prob_{VO}(q)$ is given by:

$$Prob_{VO}(q) = 1 - \left(1 - \left(\sum_{i=1}^{qp} |QP_i| \Big/ \sum_{j=1}^{m} |P_j|\right)\right)^{|U|} \tag{6}$$

Equipped with the above models, we present a simple and effective algorithm, called *Bestm*, to compute an appropriate value for the partitioning granularity m. Initially, *Bestm* sets m to a maximum value m_{max}. It then scans the data set once to compute the cardinality for each partition, and utilizes this information to derive *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 data set 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 smallest *EVO* is chosen as the partitioning granularity for CADS.

Finally, our cost model can be extended to estimate the *VO* size of a variable-length partitioning, as long as the distribution of the keys inside each partition is uniform. Specifically, for any sample query $q \in QS$, we compute (i) the set of partitions that intersect q, namely $QP_1, QP_2, \ldots, QP_{qp}$ and (ii) $|q|$, i.e., the number of results retrieved by q. With this information, we can apply Eqs. 1–6 to derive *EVO*. However, it is complicated to find the optimal such partitioning. In the next section we explain how we can achieve an effective variable-length partitioning.

6 A-CADS

As discussed in the previous section, a coarse m leads to false transmissions, whereas a fine one increases the number of empty partitions, which is problematic, especially for skewed data sets. Meanwhile, the optimal m according to the cost models leads to good performance, only when the data distribution remains the same as when m is calculated. If, however, the distribution changes over time, the optimal partitioning may become obsolete. In this case, the performance of CADS degrades, and the entire index must be re-built from scratch using the granularity computed with current statistics. This is a rather costly process. Motivated by these shortcomings, we next describe *Adaptive CADS* (*A-CADS*), which (i) minimizes false transmissions and empty partitions *simultaneously*, and (ii) dynamically adjusts the structures according to distribution changes, thus achieving high performance *at all times*.

[4] When q overlaps with only one partition, the corresponding TMH-Tree is analyzed similarly to the DPM-Tree; i.e., the traversal initially follows one path until a path split node.

6.1 Indexing scheme

In general, there are three major differences between A-CADS and CADS regarding the structure and maintenance of the DPM-Tree. (i) Whereas CADS employs a *full* (balanced) DPM-Tree, in A-CADS it can be *unbalanced* (i.e., leaves may reside at any level of the tree). This enables using variable partitioning granularity for distinct regions of the data space (and thus different partition sizes). (ii) The DPM-Tree in CADS is static, with a pre-defined number of leaf nodes, each corresponding to a fixed partition of the data domain D. The DPM-Tree in A-CADS, on the other hand, is *adaptive*, and its structure changes as updates occur. (iii) While CADS constructs the DPM-Tree during an initialization phase, A-CADS builds the tree *incrementally*, and maintains it while records are inserted into and deleted from the index. For this purpose, it introduces two new algorithms *InsertDPM* (for insertions) and *DeleteDPM* (for deletions), which adjust the tree structure through node splits (in case of *InsertDPM*) and merges (resp., *DeleteDPM*).

In A-CADS, the DPM-Tree initially consists of a single node (the root), whose associated partition covers the entire data space. The tree grows as new tuples are inserted, which is controlled by the recursive procedure *InsertDPM*. Figure 15 outlines the pseudo-code of *InsertDPM* that has three input parameters: (i) a DPM-Tree node N, (ii) the record to be inserted r, and (iii) an interval I, which is the partition corresponding to N. The insertion of record r starts by calling *InsertDPM* with the root of the DPM-Tree, r, and the full domain D. The procedure first locates the DPM-Tree leaf, whose corresponding partition contains the search key $r.k$ of r (lines 1–4). Specifically, as long as N is not a leaf node, it first determines the partition extents of N's children through function *computeIntervals* (line 2), which splits I into two equal-length intervals I_1 and I_2. Then, depending on whether $r.k$ lies in I_1 or I_2 (lines 3–4), it traverses to N's left (right) child, respectively, and the process continues recursively until reaching the leaf level.

Once reaching a DPM-Tree leaf N (line 5), the algorithm checks whether N should be split. The decision is based on the inequality evaluation of line 6, i.e., $EVO(I, 1) \leq EVO(I, 2)$. Function *EVO* returns the expected *VO* size after r is inserted to the tree, computed according to the cost models of Sect. 5. Its two parameters are (i) data space of concern I (i.e., N's partition), and (ii) the number of partitions to be used for this space, which is either 1 or 2. Intuitively, line 6 can be translated into: "after r is inserted, is the expected *VO* smaller (or equal) when the DPM-Tree keeps using one partition for the data space I, compared to using two partitions?" A positive answer means that one node/partition is the best choice for I. Hence, the procedure simply inserts r into the respective TMH-Tree pointed by $N.R$ (line 7). Otherwise (i.e., two nodes/partitions lead to better performance), a split occurs at node N.

To split a node N (line 8), *InsertDPM* first creates two new ones *N.lc* and *N.rc* as left and right child for N (line 9), and computes their respective partition extents (line 10). Subsequently, it destroys the TMH-tree $N.R$, retrieves the records in $N.R$, and distributes them into *N.lc* and *N.rc*, based on if their search key lies in I_1 or I_2, respectively (line 11). Then it constructs the corresponding TMH-Trees pointed by *N.lc.R* and *N.rc.R*. Finally, the procedure is recursively invoked for *N.lc* or *N.rc*, depending on which node r should be inserted in (lines 12–13), during which further splits may occur.

We conclude *InsertDPM* with the observation that propagated splits entail the unnecessary computation of TMH-Trees (in line 11), as these trees are destroyed during subsequent splits (incurred by the recursive calls in lines 12 and 13). As a further optimization, in our implementation *InsertDPM* does not construct any TMH-Tree until all propagated splits finish. Instead, it marks the modified subtrees of the DPM-Tree, buffers the sets of records to be indexed by each affected leaf, and creates the TMH-Trees at a subsequent step. For simplicity, however, the pseudo-code in Fig. 15 omits this detail.

Next we present *DeleteDPM*, shown in Fig. 16, which deals with tuple deletions and node merging. Unlike node splits which can happen on any leaf node, a merge is only performed on two leaves sharing a common parent (note that the sibling of a leaf may be an intermediate node since the tree is unbalanced). The function takes the same arguments as *InsertDPM*, where r is now the record to be deleted. *DeleteDPM* first traverses the tree until reaching the leaf N that accommodates r (lines 1–4). If N is the root, or its sibling (denoted by *N.sb*) is not a leaf, the procedure simply deletes r from TMH-Tree $N.R$ (lines 6–7). Otherwise (i.e., N is not the root and $N.sb$ is a leaf), it first computes the partition

```
InsertDPM (DPMNode N, Record r, Interval I)
1.  If N is an intermediate node
2.     ($I_1$, $I_2$) = computeIntervals(I)
3.     If r.k lies in $I_1$, InsertDPM (N.lc, r, $I_1$)
4.     Else if r.k lies in $I_2$ InsertDPM (N.rc, r, $I_2$)
5.  Else // N is a leaf node
6.     If EVO(I, 1) $\leq$ EVO(I, 2)
7.          Insert r into TMH-Tree pointed by N.R
8.     Else // EVO(I, 2) < EVO(I, 1), split N
9.        Create two new children for N
10.       ($I_1$, $I_2$) = computeIntervals(I)
11.       Destroy the TMH-Tree pointed by N.R, distribute
          all its records to N.lc and N.rc based on if their
          search keys lie in $I_1$ or $I_2$, respectively, and build
          the TMH-Trees N.lc.R and N.rc.R
12.       If r.k lies in $I_1$, InsertDPM (N.lc, r, $I_1$)
13.       Else InsertDPM (N.rc, r, $I_2$)
```

Fig. 15 Algorithm *InsertDPM* for A-CADS

```
DeleteDPM (DPMNode N, Record r, Interval I)
1.  If N is an intermediate node
2.     ($I_1$, $I_2$) = computeIntervals(I)
3.     If r.k lies in $I_1$, DeleteDPM (N.lc, r, $I_1$)
4.     Else if r.k lies in $I_2$ DeleteDPM (N.rc, r, $I_2$)
5.  Else // N is a leaf node
6.     If N is the root or N's sibling is not a leaf
7.        Delete r from N.R
8.     Else // N is not the root and N's sibling is a leaf
9.        $I_{pnt}$ = N.pnt.computeParentInterval(I, N)
10.       If $EVO(I_{pnt}, 2) \leq EVO(I_{pnt}, 1)$
11.          Delete r from N.R
12.       Else // $EVO(I, 1) < EVO(I, 2)$, merge N
13.          Let N.sb be N's sibling. Retrieve all the records
             from N.R and N.sb.R, build a TMH-Tree over
             them, store its root's pointer to N.pnt.R, and
             delete N and N.sb
14.          DeleteDPM (N.pnt, r, $I_{pnt}$)
```

Fig. 16 Algorithm *DeleteDPM* for A-CADS

I_{pnt} corresponding to N's parent (denoted by *N.pnt*) with function *computeParentInterval* (line 9). The latter checks whether N is a left or a right child of its parent node, and expands I accordingly to obtain I_{pnt}.

DeleteDPM then distinguishes two cases. (i) In line 10, *EVO* is smaller (or equal) when I_{pnt} is decomposed into two partitions, rather than when it is a single partition (considering that r is already removed). In other words, merging N's partition with *N.sb*'s to a single partition I_{pnt} (i.e., merging N with *N.sb*) is not more beneficial than maintaining the two partitions in their current form. The algorithm simply deletes r from $N.R$. (ii) If line 10 returns false, a merge between N and *N.sb* must be performed. *DeleteDPM* retrieves all records in the TMH-Trees rooted at $N.R$ and *N.sb.R* (including r), constructs a new TMH-Tree over these tuples, assigns its root pointer to *N.pnt.R* in N's parent, and deletes N and *N.sb* (line 13). Finally, the procedure recursively calls itself in order to delete r from *N.pnt* because an *upwards* propagated merge may occur. Merge can be thought of as the reverse operation of split.

Note that the DO sends the updates to the SP in batches. The latter first processes the involved insertions (deletions) using *InsertDPM* (*DeleteDPM*) as explained above, but during the process it marks all the visited DPM-Tree nodes (e.g., it stores their pointers to a temporary list). After finishing the last update operation in the batch, the SP updates the digests contained in the marked nodes bottom-up at a subsequent step, so that the instance of its index complies with that of the DO at all times.

6.2 Query processing

We now turn to query processing in A-CADS. Recall that the difference between CADS and A-CADS is that, in A-CADS the DPM-Tree is unbalanced, adaptive, and incrementally maintained. Among the query processing algorithms, *RangeDPM* (that produces the answer set and the *VO* for a query) and $ReconstructH_{DPM}$ (that re-constructs the root digest from the *VO*) do not take into account the level of the leaves in the DPM-Tree, or changes occurring to the structure. Therefore, they are not affected at all by these changes, and remain the same.

On the other hand, *CombineVO* (Fig. 12), which is the core component of *VCM*, is affected by the fact that the DPM-Tree structure may change over time. Recall that *CombineVO* scans a new *VO* in its compressed form (*newVO*) and an old one from the cache (*cachedVO*) simultaneously, distinguishing four different cases depending on the current entries E_n and E_c in *newVO* and *cachedVO*, respectively. Structural changes in the DPM-Tree introduce two situations not encountered in CADS. (i) When E_n is [and E_c is *begin_TMH*, an internal node of the DPM-Tree was a leaf when *cachedVO* was created (i.e., a split occurred in the meantime). In this case, *CombineVO* discards all the entries between E_c = *begin_TMH* until its corresponding *end_TMH* from *cachedVO*, and includes in *VO* all the entries between E_c = [to its corresponding] token from *newVO*. (ii) When E_n is *begin_TMH* and E_c is [, then a leaf node of the DPM-Tree was an internal node when *cahcedVO* was produced (i.e., a merge has occurred). To handle this, *CombineVO* discards all entries from *cachedVO* between E_c = [to its corresponding] token, and inserts in *VO* the entries between E_n = *begin_TMH* to its respective *end_TMH* from *newVO*.

Finally we comment on the proofs of soundness, completeness and temporal completeness under A-CADS. As discussed above, whether the DPM-Tree is unbalanced or not, or how it is constructed and maintained, has no impact on query processing, and, thus, does not influence result correctness. The fact that the DPM-Tree changes structure over time affects exclusively the *CombineVO* routine, which is part of *VCM*. This means that, without considering *VCM*, all correctness guarantees of CADS remain valid in A-CADS. Note that the only requirement of *VCM* is to restore the full *VO* properly from a compressed one and the cache. The above modifications capture all cases regarding an entry in the new *VO* and cached *VO*, and, therefore, synchronize the cached with the new *VO* successfully. In summary, none of the changes in data structures and algorithms introduced in A-CADS compromises the soundness, completeness or temporal completeness of query results.

7 Experimental evaluation

We implemented all methods using the Crypto++ library,[5] and deployed them on a Core 2 Duo 2.2 GHz CPU with

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

Table 3 Experimental parameters

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

2 GBytes of RAM. Each record r consumes 100 bytes and contains a search key $r.k$ which has values normalized to [0,1]. The query specifies a range on the search attribute. We generated synthetic data sets UNI and SKD, in which the key attribute follows a uniform and Zipfian distribution, respectively. In SKD, the skewness parameter is set to 0.8, so that 77% of the records fall in the 20% of the data space.

At every timestamp, *AR* updates arrive at the system (the period between timestamps can be perceived as any time interval, e.g., a second, a minute, etc.). An update involves a deletion of a random tuple and an insertion of a new one with the same id but different key. Consequently, the number of update operations is $|U| = 2 \cdot AR$, and the data set cardinality *DC* is constant at all times. The new key values follow their initial distribution. We monitor *QC* running queries. Unless otherwise stated, each query covers 0.1% of the data domain. Table 3 summarizes the system parameters under investigation along with their ranges, highlighting the default values in bold face.

Section 7.1 utilizes the cost models of Sect. 5 to identify the optimal granularity m for CADS, and assess their accuracy. Section 7.2 compares CADS against REF. Finally, Sect. 7.3 evaluates the relative performance of CADS and A-CADS.

7.1 Computation of the optimal granularity

We set the system parameters to their default values (i.e., $DC = 100$ K, $QC = 1$ K and $AR = 100$) and switch off the virtual caching mechanism (*VCM*). For the first set of experiments we perform uniform queries on UNI data set. Figure 17a plots both the estimated *VO* size calculated for all queries per timestamp using the cost models, and its actual size measured in the experiments, against the number of partitions m. The estimated values are accurate for all values of m, with a maximum error of 10.7%. When $m \leq 2^4$, most partitions are affected by updates, meaning that very few false transmissions are eliminated. Consequently, the total *VO* size is as high as that of REF (recall that when $m = 1$, CADS reduces to REF). As m grows, the amount of false transmissions (and thus the *VO* size) drops quickly until reaching $m = 2^{10}$, after which the penalty of a fine granularity (e.g., large number of timestamps and hash values in the *VO*) emerges as a significant overhead.

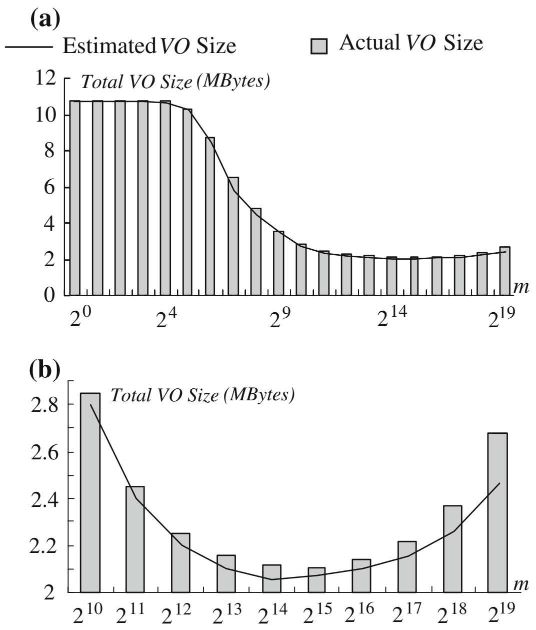

Fig. 17 Total *VO* size vs. m (UNI data set). **a** Effect of m. **b** Optimal m

The change of *VO* size for m between 2^{10} and 2^{19} is rather subtle compared to the initial drop at low values of m. Figure 17b zooms in the range $2^{10} \leq m \leq 2^{19}$. The actual *VO* size continues to drop until $m = 2^{15}$, which is the optimal granularity. Then, the *VO* size starts to increase, with an accelerating speed, signifying that the overhead introduced by CADS (e.g., timestamps) exceeds the savings achieved by reducing false transmissions. The estimated *VO* reaches its lowest point at $m = 2^{14}$, which is very close to the optimal one (2^{15}); the difference in *VO* sizes between $m = 2^{14}$ and $m = 2^{15}$ is negligible.

In order to evaluate the generality of these observations, Fig. 18 repeats the above experiment on the SKD data set (the queries are still uniform). Again, the cost models are accurate for all values of m. The *VO* size is highest at $m = 1$ (i.e., REF), then decreases sharply as m grows, and starts to increase after $m > 2^{15}$. The estimated value ($m = 2^{14}$) for the best granularity is very close to the optimal one, which is 2^{15}. Comparing with Fig. 17, the optimal granularity is not significantly affected by data skewness.

Note that the *VO* reduction achieved by higher values of m is less pronounced in Fig. 18 than in Fig. 17. This is because in the SKD data set, the initial data as well as the updates have search keys concentrated in a certain area of the data space, whereas the queries are uniformly distributed. Consequently, most queries cover sparse areas and thus have few results; the ones that fall into dense regions, on the other hand, are more likely to be affected by updates, and their results have to be re-transmitted anyway. Therefore, the ratio between the amount of false transmissions and that of necessary ones

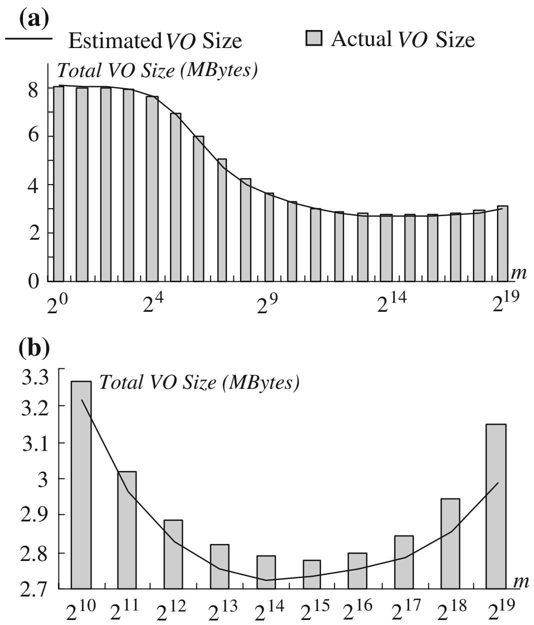

Fig. 18 Total *VO* size vs. *m* (SKD data set). **a** Effect of *m*. **b** Optimal *m*

is smaller in the SKD data set. Recall that for this set of experiments, we disabled *VCM* as it is not taken into account by our cost models. We empirically verified that the computed optimal granularity is also effective in the presence of *VCM* because for large values of $m(>2^{11})$ the effect of the granularity is not very significant.

Figure 19 investigates the effectiveness of the model for non-uniform queries. Specifically, Fig. 19a presents results for the UNI data set, whereas Fig. 19b focuses on the SKD data set. In both experiments, the left boundary of each query follows the same distribution as keys in the SKD data set. For SKD, we decreased the query length to 0.001% of the data domain (instead of 0.1% in the other experiments) because a larger query would retrieve almost all records (queries fall in very dense areas). For both data sets, our cost model achieves high accuracy (below 18% in all settings). Concerning the optimal granularity, in the UNI data set, the cost model predicts $m = 2^{14}$, while the true best value for m is 2^{15}, and the difference in performance is negligible. In the case of SKD, m does not have a significant impact over the *VO* because this is dominated by the output size (note the large *VO* size in Fig. 19, compared to the previous diagrams).

7.2 CADS vs. REF

We evaluate CADS against REF, investigating the impact of various system parameters. For the following experiments, we assume uniform queries and use the best value for m as determined by the cost models. First, we assess the effect of the data cardinality *DC*, after setting $QC = 1\text{K}$, $AR = 100$, and enabling *VCM*. Figure 20 illustrates the total processing time

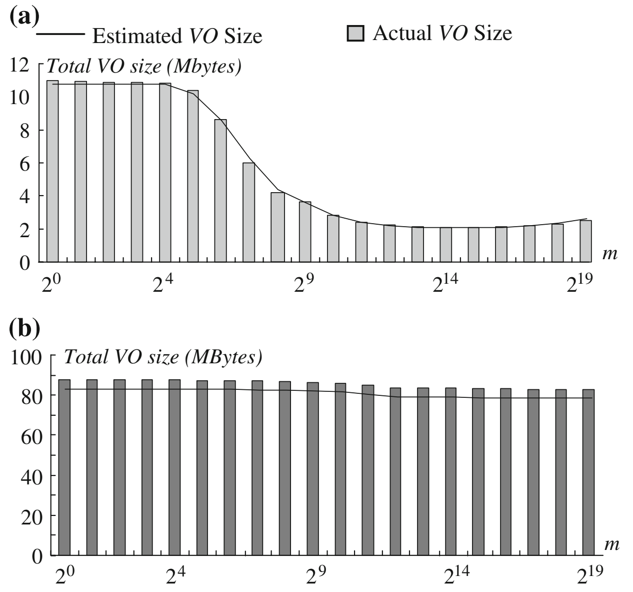

Fig. 19 Total *VO* size vs. *m* (skewed queries). **a** UNI dataset. **b** SKD dataset

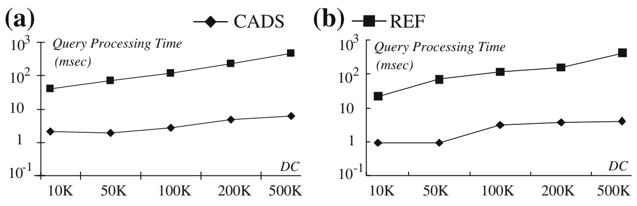

Fig. 20 Query processing time vs. *DC*. **a** UNI. **b** SKD

(milliseconds). The overhead of REF is significantly higher since it processes 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 linearly with *DC* due to the fact that it has to scan the leaf nodes of the DMH-Tree to retrieve the result sets for all queries. CADS is much less sensitive to *DC* because of *VCM*. Specifically, when a visited node has not been altered with respect to the previous transmission, the traversal of its subtree is entirely skipped. Note that CADS has better performance for SKD because the number of queries affected by updates is smaller than UNI (as most updates are concentrated in a small number of partitions).

Figure 21 shows the total *VO* size (Kbytes) for all queries, transmitted by the SP per timestamp as a function of *DC*. The communication overhead of REF again increases linearly since the number of records in the result is linear to the cardinality, and all these records are transferred to the client at each timestamp. In CADS, the growth of the result is partially absorbed by *VCM*. Specifically, since *QC* and *AR* are fixed and independent of *DC*, the size of the result matters mainly for the first transmission. Comparing UNI and

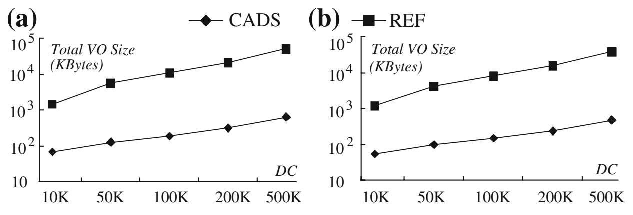

Fig. 21 Total *VO* size vs. *DC*. **a** UNI. **b** SKD

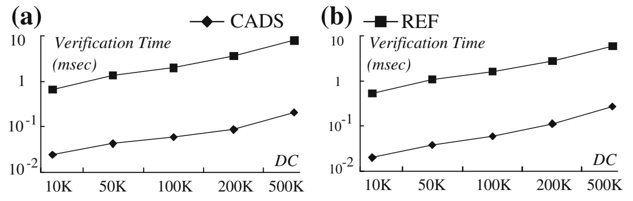

Fig. 22 Verification time vs. *DC*. **a** UNI. **b** SKD

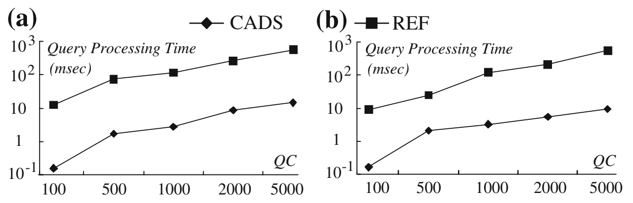

Fig. 23 Query processing time vs. *QC*. **a** UNI. **b** SKD

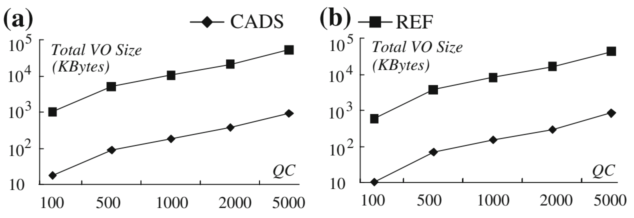

Fig. 24 Total *VO* size vs. *QC*. **a** UNI. **b** SKD

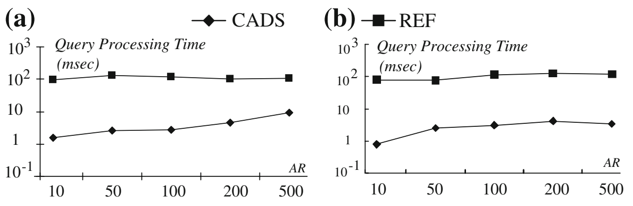

Fig. 25 Query processing time vs. *AR*. **a** UNI. **b** SKD

SKD, the effect of *VCM* is more pronounced for the latter. In particular, in Figs. 17 and 18 where *VCM* is turned off, CADS incurs a higher cost in SKD than in UNI at the best partitioning granularity, whereas in Fig. 21 the *VO* size is slightly lower in SKD. Most records in SKD reside in the first few partitions, leading to relatively tall TMH-Trees for these partitions, and thus more optimization opportunities for *VCM*.

Figure 22 depicts the verification time (milliseconds) per timestamp at each client. REF imposes a heavy burden on the clients because, due to false transmissions, the client must verify its query at every timestamp. The performance gap between CADS and REF is slightly wider in the UNI data set than in SKD, although the latter has a lower *VO* size as shown above. This is because the ratio of false transmissions is lower in SKD and the additional *VO* reduction comes mainly from *VCM*, which does not affect the verification cost. Specifically, 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.

The second set of experiments evaluates the effect of the query cardinality (*QC*) for $DC = 100\text{K}$ and $AR = 100$. The query processing cost (Fig. 23) 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 Fig. 24 follows similar trends for the same reasons.

The next set of experiments assesses the effect of the arrival rate (*AR*), after fixing *DC* to 100 K and *QC* to 1 K. Figure 25 shows the total query processing cost per timestamp at the SP. REF is insensitive to *AR* because 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 data distribution. On the other hand, the cost of CADS increases because the number of affected queries is positively correlated to *AR*. This is more obvious in UNI, where the updates are uniformly distributed in space. In SKD, updates follow the initial skewed distribution and thus focus on the first few partitions. As shown in Fig. 25, even if *AR* is high, the processing cost of CADS is relatively stable, since most additional updates happen in partitions that are already affected.

Figure 26 illustrates the total *VO* size for all queries per timestamp. As expected, *AR* does not influence the *VO* size of REF. For CADS however, the *VO* size increases with *AR*, because the more updates occur at each timestamp, the more queries are affected, whose results must be transmitted to the clients. Meanwhile, the increased number of updates also causes more nodes in the DPM-Tree and the TMH-Trees to change, invalidating their corresponding cache; consequently, *VCM* becomes less effective.

Figure 27 depicts the verification time at the client. The diagrams are similar to those in Figs. 25 and 26, except that the curves converge faster. This is due to the lack 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. In SKD the verification burden is less affected than in UNI, especially for large values of *AR*, because most updates fall in a few dense partitions, and thus influence fewer queries.

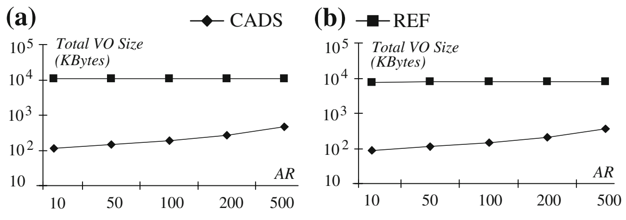

Fig. 26 Total *VO* size vs. *AR*. **a** UNI. **b** SKD

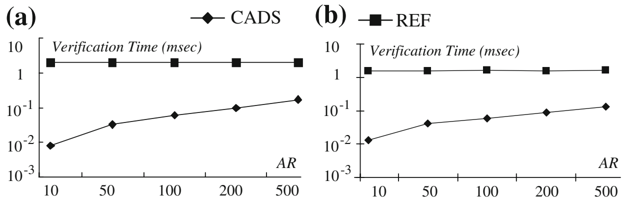

Fig. 27 Verification time vs. *AR*. **a** UNI. **b** SKD

Summarizing, 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.

7.3 A-CADS vs. CADS

We compare A-CADS with CADS focusing on a skewed data set since, as discussed in Sect. 6, A-CADS targets skewed data sets with changing distributions. Specifically, we modify SKD as follows: let A be the center, where most of the record keys are concentrated. In order to evaluate the adaptability of A-CADS, when we re-insert a record during an update, we (randomly) set its key to be skewed around another center B, which is relatively far away from A. In other words, contrary to the previous experiments, where the data distribution remains the same, here we gradually shift the center of skewness as updates occur, so that the distribution changes over time.

In our first experiment we vary the Zipfian parameter a of SKD, after setting the parameters of Table 3 to their default values (DC = 100K, QC = 1K, AR = 100), in order to investigate how A-CADS behaves under different levels of skewness (note that the larger the value of a, the higher the skewness). Figure 28a depicts the total *VO* size per timestamp, excluding the result size and focusing only on the additional authentication information (i.e., digests, boundary records, timestamps and signatures). A-CADS exhibits considerably better performance than CADS for highly skewed

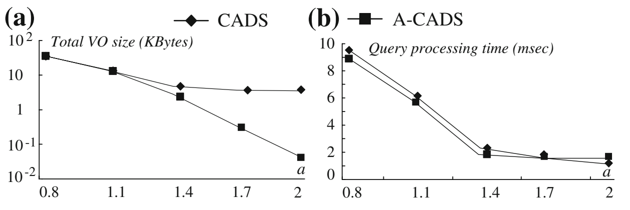

Fig. 28 Effect of skewness a. **a** Total *VO* size. **b** Query processing time

data sets, achieving up to almost two orders of magnitude smaller *VO* size when $a = 2$. This is due to two factors. (i) Since the queries are uniformly distributed in the data space, the larger the value of a, the more scarcely populated areas these queries cover. The unbalanced DPM-Tree of A-CADS indexes these areas (which involve many empty partitions) with much fewer nodes than the (balanced) DPM-Tree of CADS. Consequently, CADS visits more nodes than A-CADS and includes a larger number of digests/timestamps in the *VO*. (ii) The DPM-Tree in A-CADS is adaptive to the changes in the data set distribution and, thus, it retains its good performance at all times. On the other hand, the selected granularity m in CADS ceases to be optimal after a number of updates, and the index becomes less efficient.

Figure 28b plots the query processing cost per timestamp as a function of a. Since the data are skewed, CADS uses coarse granularity in order to reduce the number of empty partitions, which increases the number of false transmissions, and, consequently, CPU cost. A-CADS, on the other hand, applies fine granularity at dense regions, reducing false transmissions. Moreover, observe that the total *VO* size and processing time of both A-CADS and CADS exhibit a decreasing trend. This is because the queries are uniform and, thus, the updates affect fewer queries as the skewness increases. Therefore, the SP generates a smaller number of new *VO*s per timestamp.

In Fig. 29 we set $a = 1.4$, $QC = 1$K and $AR = 100$, and we vary the data set cardinality DC. As expected, both the total *VO* size and the query processing time increase with DC due to the taller trees. Most of the gains of A-CADS stem from the fact that it handles empty partitions more efficiently. However, when the data set cardinality is higher, the probability for an empty partition to be occupied by at least one record increases. This in turn raises the likelihood to create a new node in the DPM-Tree in A-CADS. Therefore, the *VO* size in A-CADS tends to converge to that of CADS when the data space is densely populated. According to Fig. 29b, A-CADS is faster than CADS for the reasons explained in the context of Fig. 28b.

Figure 30 illustrates how the arrival rate AR of the updates influences the performance of the two schemes, when $a = 1.4$, $QC = 1$ K and $DC = 100$ K. As shown in Fig. 30a, the *VO* size in A-CADS is 35–52% below than that in CADS.

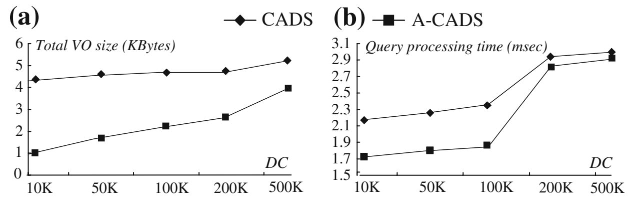

Fig. 29 Effect of *DC*. **a** Total *VO* size. **b** Query processing time

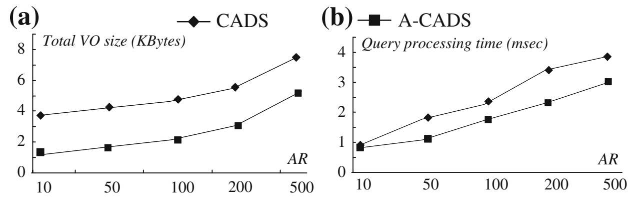

Fig. 30 Effect of *AR*. **a** Total *VO* size. **b** Query processing time

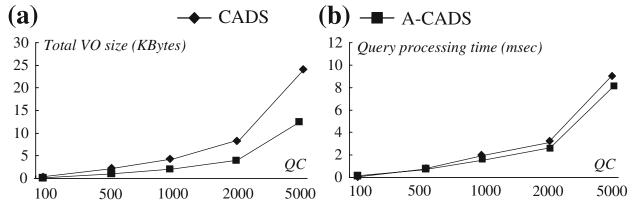

Fig. 31 Effect of *QC*. **a** Total *VO* size. **b** Query processing time

Moreover, the difference between the two *VO* sizes remains rather constant as *AR* increases, which demonstrates the effectiveness of A-CADS in the presence of fast changes in data set distribution. Once again, Fig. 30b confirms that, despite the index re-organization, A-CADS outperforms CADS in terms of processing time.

Figure 31 compares A-CADS and CADS under different query cardinalities (*QC*), while setting $a = 1.4$, $AR = 100$ and $DC = 100\text{K}$. The benefits of A-CADS against CADS increase for larger values of *QC*, since the savings during the *VO* generation for each query accumulate gradually.

Finally, Fig. 32 shows the average CPU time of an update in CADS and A-CADS. Specifically, Fig. 32(a, b) investigate the effect of parameters *a* and *DC*, respectively. In all settings, the costs of the two methods are similar, with A-CADS being slightly more expensive due to structural changes in the DPM-tree. However, this additional overhead is negligible, because split / merge operations usually involve small TMH-trees. The update cost increases with the Zipfian factor *a* because a higher degree of skewness leads to deeper TMH-trees in the denser areas and, therefore, invalidation of more digests by each update. For similar reasons, a larger data cardinality *DC* also incurs higher overhead since more records lead to deeper TMH-trees.

To conclude, in comparison with CADS, A-CADS generates a smaller total *VO* size in all the settings. The gains are especially pronounced in data sets that exhibit high level

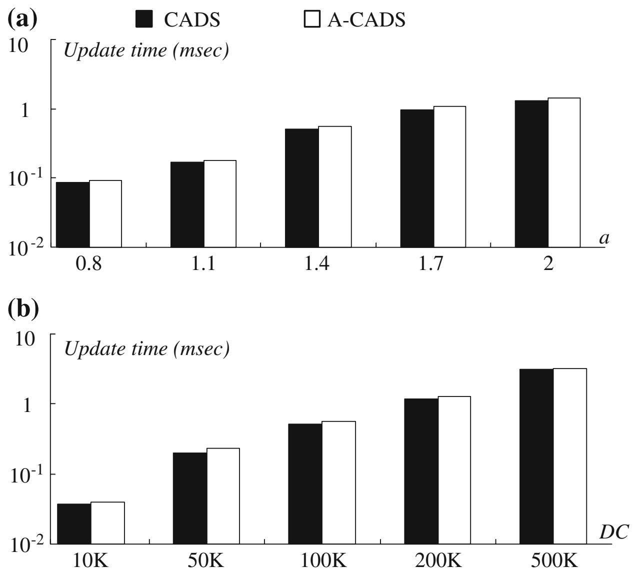

Fig. 32 Comparison of ADS update overhead. **a** Effect of skewness *a*. **b** Effect of *DC*

of skewness, where A-CADS can produce a *VO* that is up to two orders of magnitude smaller than that of CADS. At the same time, A-CADS outperforms CADS in terms of query processing time, by successfully balancing the structure reconfiguration overhead and the savings due to the better indexing of the empty partitions.

8 Conclusion

In this paper we address continuous query processing and authentication on relational data streams. We assume a service provider (SP) that constantly collects record updates (e.g., stock exchange rates) from a data owner (e.g., stock market). Numerous clients (e.g., brokers) register long-running queries directly to the SP. The SP returns to the clients the query results, as well as authentication information necessary to establish their correctness. In addition, the clients are 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 utilizes a data space partitioning technique and an efficient caching mechanism to reduce (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. CADS and REF are main memory-based in order to achieve real-time query evaluation and fast structure updating.

CADS utilizes a fixed partitioning of the data space. We study the effect of the partition granularity, and devise analytical models for minimizing the size of the generated verification object. Additionally, we develop an *adaptive* version of CADS (A-CADS) that utilizes the cost models to dynamically adjust the partitioning depending on the changes of the data distribution. We show through extensive experiments that CADS outperforms REF significantly in all aspects. We also confirm the accuracy of the analytical models, and demonstrate the gains of A-CADS in cases of continuously changing distributions.

Acknowledgment. This work was supported by grant HKUST 6181/08 from Hong Kong RGC. An early version of this paper appears in: Papadopoulos, S., Yang, Y., Papadias, D. CADS: *Continuous Authentication on Data Streams*. Proceedings of the Very Large Data Bases Conference (*VLDB*), pp. 135–146, Vienna, Austria, September 23–28, 2007.

References

1. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. SIGMOD (2004)
2. Atallah, M.J., Cho, Y., Kundu, A.: Efficient data authentication in an environment of untrusted third-party distributors. ICDE (2008)
3. Babcock, B., Chaudhuri, S., Das, G.: Dynamic sample selection for approximate query processing. SIGMOD (2003)
4. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry: Algorithms and Applications. Springer-Verlag, New York (1997)
5. Cheng, W., Pang, H., Tan, K.-L.: Authenticating multi-dimensional query results in data publishing. DBSec (2006)
6. Damiani, E., Vimercati, C., Jajodia, S., Paraboschi, S., Samarati, P.: Balancing confidentiality and efficiency in untrusted relational DBMSs. CCS (2003)
7. 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)
8. Devanbu, P., Gertz, M., Martel, C., Stubblebine, S.: Authentic data publication over the Internet. J. Comput. Secur. **11**(3), 291–314 (2003)
9. Getoor, L., Taskar, B., Koller, D.: Selectivity estimation using probability models. SIGMOD (2001)
10. Goodrich, M., Tamassia, R., Triandopoulos, N., Cohen, R.: Authenticated data structures for graph and geometric searching. CT-RSA (2003)
11. Guha, S., Shim, K., Woo, J.: Rehist: Relative Error Histogram Construction Algorithms. VLDB (2004)
12. Hacıgümüş, H., Iyer, B., Li, C., Mehrotra, S.: Executing SQL over encrypted data in the database-service-provider model. SIGMOD (2002)
13. Hacıgümüş, H., Iyer, B., Mehrotra, S.: Providing databases as a service. ICDE (2002)
14. Kundu, A., Bertino, E.: Structural signatures for tree data structures. VLDB (2008)
15. Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L.: Dynamic authenticated index structures for outsourced databases. SIGMOD (2006)
16. Li, F., Yi, K., Hadjieleftheriou, M., Kollios, G.: Proof-infused streams: enabling authentication of sliding window queries on streams. VLDB (2007)
17. 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)
18. Merkle, R.: A certified digital signature. CRYPTO (1989)
19. Mykletun, E., Narasimha, M., Tsudik, G.: Signature bouquets: immutability for aggregated/condensed signatures. ESORICS (2004)
20. Narasimha, M., Tsudik, G.: Authentication of outsourced databases using signature aggregation and chaining. DASFAA (2006)
21. National Institute of Standards and Technology. FIPS PUB 180-1: Secure Hash Standard. National Institute of Standards and Technology (1995)
22. Pang, H., Jain, A., Ramamritham, K., Tan, K.-L.: Verifying completeness of relational query results in data publishing. SIGMOD (2005)
23. Pang, H., Mouratidis, K.: Authenticating the query results of text search engines. VLDB (2008)
24. Pang, H., Tan, K.-L.: Authenticating query results in edge computing. ICDE (2004)
25. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978)
26. Sion, R.: Query execution assurance for outsourced databases. VLDB (2005)
27. Tamassia, R., Triandopoulos, N.: Efficient content authentication in peer-to-peer networks. International Conference on Applied Cryptography and Network Security (2007)
28. Wong, W.K., Cheung, D., Hung, E., Kao, B., Mamoulis, N.: Security in outsourcing of association rule mining. VLDB (2007)
29. Xie, M., Wang, H., Yin, J., Meng, X.: Integrity audit of outsourced data. VLDB (2007)
30. Yang, Y., Papadopoulos, S., Papadias, D., Kollios, G.: Authenticated indexing for outsourced spatial databases. VLDB J. **18**(3), 631 (2009)
31. Yang, Y., Papadopoulos, S., Papadias, D., Kollios, G.: Spatial outsourcing for location-based services. ICDE (2008)
32. Yi, K., Li, F., Hadjieleftheriou, M., Kollios, G., Srivastava, D.: Randomized synopses for query assurance on data streams. ICDE (2008)
33. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. **38**(2), (2006)