

Authenticated Index Structures for Aggregation Queries in Outsourced Databases

Feifei Li[†] Marios Hadjieleftheriou[‡] George Kollios[†] Leonid Reyzin[†]
[†]CS Department, Boston University, USA. [‡]AT&T Labs-Research, USA.
(lifeifei, gkollios, reyzin)@cs.bu.edu, marioh@research.att.com

Technical Report BUCS-TR-2006-011

July 20, 2006

Abstract

In an outsourced database system the data owner publishes information through a number of remote, untrusted servers with the goal of enabling clients to access and query the data more efficiently. As clients cannot trust servers, query authentication is an essential component in any outsourced database system. Clients should be given the capability to verify that the answers provided by the servers are correct with respect to the actual data published by the owner. While existing work provides authentication techniques for *selection and projection* queries, there is a lack of techniques for authenticating *aggregation* queries. This article introduces the first known authenticated index structures for aggregation queries. First, we design an index that features good performance characteristics for static environments, where few or no updates occur to the data. Then, we extend these ideas and propose more involved structures for the dynamic case, where the database owner is allowed to update the data arbitrarily. Our structures feature excellent average case performance for authenticating queries with multiple aggregate attributes and multiple selection predicates. We also implement working prototypes of the proposed techniques and experimentally validate the correctness of our ideas.

1 Introduction

Consider an automobile company, Speed, that wishes to make its warehouse information accessible for sales representatives from different places

of the world but does not want to invest time and money in maintaining its own servers in foreign countries. Alternatively, Speed could disseminate its inventory information to various service providers, which will assume the responsibility of handling queries about product availability, sales information, and so on. Distributing Speed’s database helps in two respects. First, queries are answered closer to the source, reducing network latency. Second, queries are not managed at a central location which poses a central point of failure, as well as a bottleneck. Such Outsourced Database Systems [12] have attracted considerable attention recently. Abstractly, three entities interact in such systems: Data owners that need to publish their information, a number of remote servers that help disseminate the information more efficiently, and clients that issue queries about the data. Since remote servers cannot be fully trusted and moreover they could even be compromised by third-parties, security concerns must be addressed if such systems are ever to become viable in the real world. As a concrete example, a third-party may compromise the systems of a service provider hosting Speed’s inventory and tamper the database such that users are provided with incorrect sales information.

To guard against malicious/compromised servers, the owner must give the clients the ability to authenticate the answers they receive without having to trust the servers. From that point of view, query authentication has three important dimensions: *correctness*, *completeness* and *freshness*. Correctness means that the client must be able to validate that the answers to queries really do exist in the database of the owner (guard against fabricated results), and have not been modified in any way (guard against tampering of values). Completeness means that no valid answers are omitted. Freshness means that the results are based on the most current version of the database that incorporates the latest owner updates. These three aspects constitute the basic query authentication problem that has been recently examined by a variety of works [8, 18, 19, 24, 27, 30, 31]. Existing literature concentrated on authenticating selection and projection queries, e.g., “Retrieve all cars with prices in the range \$5000-\$7000”.

An important aspect of query authentication in outsourced database systems that has not been considered yet is handling aggregation queries. For example, “Retrieve the total number of cars sold with price between \$5000 and \$7000”. Currently available techniques for selection and projection queries can be straightforwardly applied to answer aggregation queries on a single selection attribute. Albeit, they exhibit very poor performance. Additionally, they cannot be generalized to multiple selection attributes without incurring high query cost as we discuss later. Hence authenticating

aggregation queries remains an open problem.

In this work, a first attempt is made to formally define the aggregation authentication problem and to provide efficient solutions that can be deployed in practice. We categorize outsourced database scenarios into two classes based on data update characteristics, and design solutions suited for each case. Concentrating on SUM aggregates, we show that in static scenarios authenticating aggregation queries is equivalent to authenticating prefix sums [14]. When updates become an issue, maintaining the prefix sums and the corresponding authentication structure becomes expensive. Hence, we propose more involved structures for efficiently handling the updates, based on authenticated B-tree [6] and R-tree structures [10]. Finally, we extend the techniques for aggregates other than SUM, and discuss some issues related to query freshness and data encryption for privacy preservation. Overall, we present solutions for handling multi-aggregate queries with multiple selection predicates, that work for a variety of aggregates like SUM, COUNT, AVG, MIN and MAX.

The rest of the paper is organized as follows. Section 2 gives the formal problem definition. Section 3 presents the necessary background and a brief overview of related work. Section 4 discusses static outsourced database scenarios, while Section 5 presents our structures for the dynamic case. Section 6 generalizes the discussion for aggregates other than SUM, and discusses advanced issues related to freshness and data encryption. An empirical evaluation is presented in Section 7. Finally, Section 8 concludes the paper.

2 Problem Definition

Consider the following SQL statement:

```
SELECT SUM(sales) FROM cars
WHERE price>$5000 and price<$7000
```

This statement contains one **aggregated** attribute (rating) and one **selection** attribute (price), in the form of a range predicate. In general, any aggregation query can be represented as follows:

$$Q = \langle \otimes(A_1), \dots, \otimes(A_c) | S_1, \dots, S_d \rangle,$$

where \otimes is the associated aggregation operation, A_i correspond to the aggregated attributes and S_j to the selection attributes contained in the predicates of the query. Attributes A_i, S_j may correspond to any of the fields

of a base table \mathbf{T} . For simplicity and without loss of generality, we assume that the schema of \mathbf{T} consists of fields $\mathbf{T}(S_1, \dots, S_d)$ with domains D_1, \dots, D_d , respectively. We refer to a query Q with d selection attributes as a d -dimensional aggregation query.

In general, there are three different types of aggregation operations: distributive, algebraic, and holistic. Distributive aggregates (like SUM, COUNT, MAX, MIN) can be computed in a divide and conquer fashion, i.e., by partitioning the data into disjoint sets, aggregating each set individually and combining partial results to get the final answer. Algebraic aggregates can be expressed as a function of distributive aggregates, e.g., $\text{AVG} \equiv \text{SUM}/\text{COUNT}$. Holistic aggregates (like MEDIAN) are harder to compute as they usually require global knowledge on the input. In the rest, we focus only on distributive aggregates. Algebraic aggregates are easily computed once distributive aggregates are addressed. Holistic aggregates are left as future work.

Finally, we concentrate on SQL queries with range predicates only. That is, given selection attributes S_j , each predicate that S_j appears in is of the form $a_j \leq S_j \leq b_j$, $a_j, b_j \in D_j$, $j \in [1, d]$. For simplicity and without loss of generality, we let S_j denote both the attribute name and the query predicate in which S_j appears in (assuming that each attribute appears in one predicate only). The meaning will be clear from context. The set of tuples from \mathbf{T} that satisfy all query predicates S_j is denoted by $\mathcal{SAT}(Q)$, and the final answer to Q as $\mathcal{ANS}(Q)$.

The problem of authenticating aggregation queries in outsourced database systems can now be defined as follows. A data owner compiles a set of authenticated structures for its data that are disseminated along with its database to a set of servers. Clients pose queries Q to the servers, which in turn use the authenticated structures to provide users with the answer to Q and special Verification Objects \mathcal{VO} w.r.t. $\mathcal{ANS}(Q)$. \mathcal{VO} s enable the clients to verify the correctness, completeness and freshness of $\mathcal{ANS}(Q)$, meaning that clients can be assured that $\mathcal{ANS}(Q)$ has been indeed computed solely from $\mathcal{SAT}(Q)$. The problem is to design efficient authentication structures for aggregation queries, as well as to define the appropriate verification objects for this purpose. In an outsourced database scenario we measure efficiency using the following metrics [18]: query cost, that includes the communication between server and clients and the verification at the client side, storage cost, and update cost of the authentication structures at the server side.

3 Background and Related Work

This section briefly reviews the necessary background material for understanding how one can achieve query authentication in outsourced database scenarios. Then, related work on selection queries is revisited, and a straightforward but quite expensive solution for authenticating aggregation queries is presented.

3.1 Cryptographic Primitives

Collision-resistant hash functions: For our purposes, a hash function h is an efficiently computable function that takes a variable-length input x to a fixed-length output $y = \mathcal{H}(x)$. *Collision resistance* states that it is computationally infeasible to find two inputs, $x_1 \neq x_2$, such that $h(x_1) = h(x_2)$. Collision-resistant hash functions can be built provably based on various cryptographic assumptions, such as hardness of discrete logarithms [20]. However, in this work we concentrate on using heuristic hash functions, which have the advantage of being very fast to evaluate, and specifically focus on SHA1 [28] which takes variable-length inputs to 160-bit outputs (and approximately 3-6 μ s to compute on our testbed computer). SHA1 is currently considered collision-resistant in practice; we also note that any eventual replacement to SHA1 developed by the cryptographic community can be used instead of SHA1 in our solution.

Public-key digital signature schemes: A public-key digital signature scheme, formally defined in [9], is a tool for authenticating the integrity and ownership of the signed message. In such a scheme, the signer generates a pair of keys (SK, PK) , keeps the secret key SK secret, and publishes the public key PK associated with her identity. Subsequently, for any message m that she sends, a signature s_m is produced by: $s_m = \mathcal{S}(SK, m)$. The recipient of s_m and m can verify s_m via $\mathcal{V}(PK, m, s_m)$ that outputs “valid” or “invalid.” A valid signature on a message assures the recipient that the owner of the secret key intended to authenticate the message, and that the message has not been changed. The most commonly used public digital signature scheme is RSA [32]. Existing solutions [30, 31, 24, 27, 18] for the query authentication problem chose to use this scheme, hence we adopt the common 1024-bit RSA. Its signing and verification cost is one hash computation and one modular exponentiation with 1024-bit modulus and exponent. It produces a signature with 1024-bit length. In general, signing

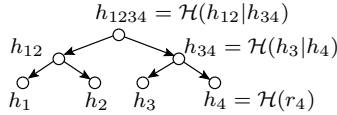


Figure 1: Example of a Merkle hash tree.

and verifying are much more expensive operations than hashing (four orders of magnitude more expensive in our testbed computer).

The Merkle hash tree: The Merkle hash tree [21] (see Figure 1) is used for authenticating a set of data values. It is a binary tree where each leaf contains the hash of a data value, and each internal node contains the hash of the concatenation of its two children. The hash value of the root is signed and published. To prove the authenticity of any data value the prover provides the verifier, in addition to the data value itself, with a Verification Object \mathcal{VO} that contains the hashes stored in the siblings of the path that leads from the root of the tree to the requested value. The verifier, by iteratively computing all the appropriate hashes up the tree, at the end can simply check if the hash computed for the root node matches the authentically published signature of the root. Given the collision resistance properties of the hash function and the guarantees of the signature scheme, it can be shown that it is computationally infeasible for an adversary (under certain computational models) to fool the verifier by modifying any of the data in the path from the leaf to the root. The Merkle tree can be straightforwardly extended to an f -way tree, where during construction of the \mathcal{VO} the prover needs to insert all the $f - 1$ siblings of every leaf and index entry involved in the computation.

3.2 Previous Work

Previous work [8, 18, 19, 24, 27, 30, 31] on query authentication has focused on studying the general selection and projection queries. The proposed techniques can be categorized into two groups, namely signature-based approaches and index-based approaches. In general, the projection queries are handled using the same techniques as selection queries. The only difference is at the granularity level (i.e. tuple level or attribute level) that the signature/hash of the database tuple is computed. Hence, next, we focus on discussing only selection queries.

In signature-based approaches [27, 30], the general idea is to produce a signature for each tuple in the database. Suppose that there exists an

authenticated index with respect to a query attribute A_q . The signatures are produced by forcing the owner to sign the hash value of the concatenation of every tuple with its right neighbor in the total ordering of A_q . To answer a selection query for $A_q \in [a, b]$, the server returns all tuples t with $A_q \in [a - 1, b + 1]$ along with a \mathcal{VO} that contains a set of signatures $s(t_i)$ corresponding to tuples in $[a - 1, b]$. The client authenticates the result by verifying these signatures for all consecutive pairs of tuples t_i, t_{i+1} in the result. It is possible to reduce the number of signatures transmitted by using special aggregation signature techniques [25].

Index-based approaches [8, 19, 24] utilize the Merkle hash tree to provide authentication. The owner builds a Merkle tree on the tuples in the database, based on the query attribute. Subsequently, the server answers the selection query using the tree, by returning all tuples t covering the result. In addition, the server also returns the minimum set of hashes necessary for the client to reconstruct the subtree of the Merkle tree corresponding to the query result and compute the hash of the root. The reconstruction of the whole subtree is necessary for providing the proof of completeness. An extension of the Merkle tree to multi-way trees which are more appropriate for database indices that are usually stored on disks appeared in [31]. A thorough comparison of all possible alternatives, as well as a novel improved structure (the EMB-tree) for authenticating selection queries, appeared in [18]. In the same work, the important issue of result freshness was also raised for the first time. A more detailed analysis of these techniques is beyond the scope of this paper. The reader is referred to [18] for a more detailed analysis of selection queries. To the best of our knowledge none of the existing works has explored aggregation queries.

Furthermore, most of previous work focused on the one dimensional selection queries except [27, 5]. Both works utilize the signature chaining idea developed in [30] and extend it for multi-dimensional range selection queries.

Related to query authentication in general (but not using the ODB model), [4] has studied authentication techniques for publishing XML documents. [23] has studied implementing the hash tree based authenticated index structure for a tamper-evident database system. Techniques for integrity in data exchange can be applied to query authentication and we refer readers to an excellent thesis [22] for more details. Also, [26] has studied the problem of computing aggregations over encrypted databases. For most of the work, we consider unencrypted databases; encrypted databases are discussed in Section 6.

3.3 A Trivial Solution

Any solution for authenticating selection queries could provide a straightforward but very inefficient solution for authenticating aggregation queries. The server simply answers the aggregation query Q as selection queries and returns $\mathcal{SAT}(Q)$ along with the \mathcal{VO} for the selection queries. The client verifies the set $\mathcal{SAT}(Q)$ and then computes the aggregation locally. However, this approach is not desirable because: 1. The communication and verification costs are linear to $|\mathcal{SAT}(Q)|$ (e.g., if the query is a SELECT * statement the cost might be prohibitive); 2. The cost for multi-dimensional aggregation queries is extremely high as a result of filtering (by different query predicates) could only be done at client side. It is thus desirable to design a solution that: 1. Has communication/verification cost sub-linear to $|\mathcal{SAT}(Q)|$. 2. Supports multi-dimensional aggregation queries efficiently.

4 The Static Case

In the *static case*, once the owner has initially created the database and published it to the servers there are no or very few updates in the system. In this section we address the problem of authenticating aggregation queries in such environments.

4.1 The APS-tree: Authenticated Prefix Sums

Assume for simplicity discrete domains $D_j = [0, M_j)$ ¹, and query $Q = \langle \text{SUM}(A_q) | S_1 = [a_1, b_1], \dots, S_d = [a_d, b_d] \rangle$. Each tuple in the database can be viewed as a point in a d -dimensional space $D_1 \times \dots \times D_d$, and the selection query as a d -dimensional range query. The d -dimensional space can be reduced to a $D_1 \times \dots \times D_d$ array C . Every coordinate of the array that contains one or more database tuples stores the SUM of attribute A_q of these tuples. The rest of the elements are initialized to zero. The answer of the query is equal to $\sum_{i_1=a_1}^{b_1} \dots \sum_{i_d=a_d}^{b_d} C[i_1, \dots, i_d]$. Answering the query requires accessing $\prod_{i=1}^d (b_i - a_i + 1)$ elements.

Alternatively, a prefix sums array can be used [14]. The prefix sum array PS of C has the same structure as C and in every coordinate it

¹For continuous or categorical domains existing values are just ordered and assigned distinct identifiers, since we are dealing with static environments. In the next section structures that do not require the domains to be discrete are presented.

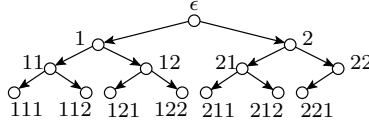


Figure 2: The tree encoding scheme.

stores: $\forall x_j \in D_j, j \in [1, d]$:

$$PS[x_1, \dots, x_d] = \sum_{i_1=0}^{x_1} \dots \sum_{i_d=0}^{x_d} C[i_1, i_2, \dots, i_d],$$

It has been shown in [14] that any range sum query on PS requires at most 2^d element accesses. For all $j \in [1, d]$, let $I(j) = 1$ if $x_j = b_j$ and $I(j) = -1$ if $x_j = a_j - 1$, and $PS[x_1, \dots, x_d] = 0$ if $x_j = -1$, then:

$$\langle \text{SUM}(A_q) | S_1 = [a_1, b_1], \dots, S_d = [a_d, b_d] \rangle = \sum_{\forall x_j \in \{a_j-1, b_j\}} \left\{ \left(\prod_{i=1}^d I(i) \right) * PS[x_1, \dots, x_d] \right\}. \quad (1)$$

Having computed PS for the aggregation attribute A_q , any query $Q = \langle \text{SUM}(A_q) | S_1, \dots, S_d \rangle$ can be answered efficiently. Furthermore, authenticating the answers becomes equivalent to authenticating these 2^d prefix sums required by equation 1. However, as we discuss next, we need to authenticate **both** their values and their locations in the array. To authenticate the elements of PS , we convert the PS into a one-dimensional array PS_{1d} , where element $PS[i_1, \dots, i_d]$ corresponds to element $PS_{1d}[k], k = \sum_{j=1}^{d-1} (i_j \prod_{n=j+1}^d M_n) + i_d$, and build an f -way MHT on top of PS_{1d} , as described in Section 3.1. We call this structure the authenticated prefix sums tree (APS-tree).

Suppose that a single element $PS_{1d}[k]$ needs to be authenticated. Traversing the tree in order to find the k -th element requires computing the correct path from the root to the leaf containing the entry, and can happen efficiently by using the following tree encoding scheme. Let h be the height of the tree (with 0 being the height of the root) and f its fanout. Every node is encoded using a unique label, which is an integer in a base- f number system. The root node has no label. The 1st level nodes have base- f representations $0, 2, \dots, f$, from left to right respectively.² The 2nd level nodes

²We use numbers from 1 to f instead of from 0 to $f - 1$ to simplify notation for the purposes of exposition.

have labels $11, \dots, 1f, 21, \dots, 2f, \dots, f1, \dots, ff$, and so on all the way to the leaves. An example is shown in Figure 2 with $f = 2$. Straightforwardly, a leaf entry with PS_{1d} offset k is converted into a base- f number $\lambda_1 \cdots \lambda_h$ with h digits (each digit, in our notation, ranging from 1 to f , and computed as $\lambda_i = 1 + \lfloor k/f^{h-i} \rfloor \bmod f$). Since the tree is full, element k lies in the k/f leaf node, and this leaf node lies in the k/f^2 index node one level up, and so on. Retrieving $PS_{1d}[k]$ is possible now by following the node with label that is the prefix of the labels for k .

Given a query Q , the server will find all the 2^d elements that are needed to answer the query and for each one of them will create a part of the \mathcal{VO} object. In particular, the \mathcal{VO} will contain the hash values of the MHT that are needed to authenticate each such element k , i.e. hash values for the sibling entries in the nodes along the query path from the root to leaf node k . In addition, the \mathcal{VO} will include the encoding of the path for each element. That is, for the element $PS_{1d}[k]$, the encoding is exactly the label $\lambda_1 \cdots \lambda_h$ of k . After the retrieval of all the elements and the creation of the \mathcal{VO} object, the server returns all of them to the client. The encoding must be included in the \mathcal{VO} to allow the client correctly recompute the hash values for nodes along the query path back to the root, as at each level, the client must know where the computed hash value for the node from the lower level should be placed.

Assuming that the hash function is collision resistant and signature on the tree root is unforgeable, it can be shown that any change to the structure of the APS-tree or the structure of a constructed \mathcal{VO} will cause the authentication procedure to fail, in exactly the same way as for the normal MHT. The fact that encoding path of an element must be included in the \mathcal{VO} for successful verification ensures the next lemma:

Lemma 1. *Given $\prod_{i=1}^d M_i$ number of ordered elements in PS_{1d} , the APS tree can authenticate both the value and the position of the k -th element $\forall k \in [1, \prod_{i=1}^d M_i]$.*

For the client to authenticate the query result, it needs to know (i) the signature of the MHT, (ii) the size of each domain (M_j), and (iii) the fanout f^3 . Essentially, the client needs to authenticate each of the 2^d elements of the answer set. First, the client, using the MHT hashes and the encodings, verifies each element, by computing the hash of the root for each path and then comparing it with the digital signature. During this step, the client

³Domain sizes and fanout of the tree are static information. Both could be authenticated only once directly from the owner at the system setup stage.

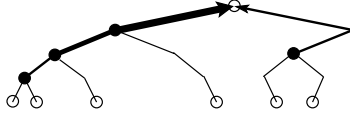


Figure 3: Merging the overlapping paths in the APS-tree. At every black node the remaining hashes can be inserted in the \mathcal{VO} only once for all verification paths towards the root.

also infers the position k for each element in PS_{1d} based on its encoding and f . Next, using the query ranges $[a_i, b_i]$ and the domain sizes, maps each element's position value k back to the coordinate in the d dimensional prefix sum array. Then, if all the elements are verified correctly, the client can check whether all required elements are returned and compute the answer to the query using equation 1.

Correctness and Completeness: Based on Lemma 1 and equation 1, we can claim that the APS-tree guarantees both completeness and correctness of the provided results.

Optimizations: A naive \mathcal{VO} construction algorithm would return an individual \mathcal{VO} for each of the prefix sum values needed. Since the authentication paths of these values may share a significant number of common edges (as shown in Figure 3), a substantial improvement in the communication and authentication costs can be achieved by combining their \mathcal{VO} s using one tree traversal. A sketch of this process is shown in Algorithm 1. In the algorithm, $SAT(Q)$ refers to the set of prefix sums required to answer the aggregation query, which is equivalent to the actual set of tuples satisfying the query, according to equation 1.

Let k^1, \dots, k^n be the indices of the PS_{1d} values that need to be authenticated. The construction algorithm essentially computes the base- f numbers corresponding to indices k^1, \dots, k^n as already explained, and defines a $n \times h$ matrix K with the base- f representations:

$$K = \begin{bmatrix} \lambda_1^1 & \lambda_2^1 & \dots & \lambda_h^1 \\ \lambda_1^2 & \lambda_2^2 & \dots & \lambda_h^2 \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_1^n & \lambda_2^n & \dots & \lambda_h^n \end{bmatrix}$$

The paths that need to be followed at every step can be found by calculating the longest common prefixes in the rows of K . Let $G_1 = \{11, \dots, 1x\}$ be a set of groups, where each group contains all elements with equal λ_1^i values in

Algorithm 1: APSQUERY(Query Q; APS-tree T; Stack \mathcal{VO} ; Stack \mathcal{SAT})

```

1  $\mathcal{VO} = \emptyset, \mathcal{SAT} = \emptyset, h = T.height$ 
2  $\mathcal{VO}.push(h), \mathcal{VO}.push(\text{domain sizes})$ 
3 for  $k^x = i_1, \dots, i_d \in \{a_1 - 1, b_1\}, \dots, \{a_d - 1, b_d\}$  do
4    $\lfloor$  Compute matrix  $K = \lambda_1^1 \cdots \lambda_h^n$  for keys  $k^1, \dots, k^n$ 
5   Compute  $G_1 = \{11, \dots, 1x\}$  using  $K$ 
   // set  $G_1$  contains  $x$  groups named  $11, \dots, 1x$ 
6   for  $S \in G_1$  do
7      $\lfloor$  Recurse(root, 2,  $S, K$ )
8   Recurse(Node  $N$ , Level  $l$ , Set  $S$ , Matrix  $K$ ):
9   begin
10    Compute  $G = \{S1, \dots, Sy\}$  using  $K$ 
    // group names will become  $\{111, \dots, 11z\}, \{1111, \dots, 111w\}$ , and
    // so on
11     $\mathcal{VO}.push(l), \mathcal{VO}.push(N.children - |G|)$ 
12    for  $\lambda_l \equiv S' \in G$  do
13      // for  $\lambda_l$ s corresponding to each  $S'$  in  $G$ 
14       $\mathcal{VO}.push(\lambda_l)$ 
15      if  $l = h$  then  $\mathcal{SAT}.push(N[\lambda_l].k)$ 
16      ; // value  $\lambda_h^x$  is the offset of key  $k^x$  in the leaf
17      for  $1 \leq i \leq N.children$  do
18         $\lfloor$  if  $i \neq \lambda_l, \forall \lambda_l \in G$  then  $\mathcal{VO}.push(N[i].\eta)$ 
19      if  $l < h$  then
20         $\lfloor$  for  $\lambda_l \equiv S' \in G$  do Recurse( $N[\lambda_l], l + 1, S'$ )
19 end

```

the first column of K , and continue recursively for each of these groups and for all remaining columns. Continue accordingly for all $G_j, j \leq h$. The size of every set G_j gives the number of paths that need to be followed every time a split occurs in the verification paths of elements k^1, \dots, k^n . For every group G_j the algorithm proceeds by normally constructing a \mathcal{VO} for the common nodes until a split occurs. The procedure is repeated recursively for all subtrees that need to be explored, according to the remaining digits of the base- f numbers. The verification procedure at the client follows similar reasoning, but in a bottom-up fashion.

Furthermore, extending the APS-tree to support multiple aggregate attributes is straightforward. A set of aggregate values and hash values is

associated with every data entry $PS_{1d}[k]$ at the leaf level of the tree, one pair of values for every aggregate attribute one wishes to be able to answer queries for. This enables answering multi-aggregate queries with only one traversal of the tree. (Alternatively, to save on storage costs at the expense of larger \mathcal{VO} s, a single hash value per node, hashing all the aggregate attributes together, can be stored; then the \mathcal{VO} will have to include all the aggregate attributes, not just the ones in which the client is interested.) APS-tree could be used to authenticate COUNT and AVG as well, as COUNT is a special case of SUM and AVG is authenticated as SUM/COUNT.

4.1.1 Cost Analysis

Three cost factors affect the performance of any authenticated index structure: query, storage and update cost.

Query cost: The query cost can be broken up into communication and verification costs. The communication depends on the size of sets \mathcal{VO} and \mathcal{SAT} . From Algorithm 1, the worst case communication cost can be expressed as:

$$\begin{aligned} \mathcal{C}_{communication} &= |\mathcal{VO}| + |\mathcal{SAT}| \\ &\leq \sum_{j=1}^{\lceil \log_f N \rceil} [(f - |G_j|) \cdot |\mathcal{H}| + (|G_j| + 1) \cdot |I|] + 2^d |I|, \end{aligned} \quad (2)$$

where $N (= M_1 \times \dots \times M_d)$ is the total size of the PS_{1d} array, $|\mathcal{H}|$ is the size of a hash value, $|I|$ the size of an integer value (all in bytes), f is the fanout of the tree and G_j are the longest common prefix groups at each column of matrix K .

The verification cost at the client in the worst case is:

$$\mathcal{C}_{verification} \leq \sum_{j=1}^{\lceil \log_f N \rceil} |G_j| \cdot \mathcal{C}_{\mathcal{H}} + \mathcal{C}_{\mathcal{V}}, \quad (3)$$

where $\mathcal{C}_{\mathcal{H}}$ and $\mathcal{C}_{\mathcal{V}}$ denote the cost of one hashing operation and the cost of one verification operation respectively.

Storage cost: The size of an APS-tree is equal to:

$$\mathcal{C}_{storage} = \sum_{l=0}^{\lceil \log_f N \rceil} f^l (|\mathcal{H}| + |I|) + N|I|, \quad (4)$$

including one hash and one pointer per tree entry. Clearly, overall the APS-tree is storage-expensive, especially if the original d -dimensional array C is sparse (i.e., when only a few coordinates contain database tuples).

Update cost: The update cost of the APS-tree depends on the update properties of the prefix sums array. Updating a single element of the prefix sums array requires updating the values of all other elements that dominate this entry. Assume that element $PS[i_1, \dots, i_d]$ is updated. Then, elements $PS[x_1, \dots, x_d]$ for $i_j < x_j < M_j, 1 \leq j \leq d$ also need to be updated, for a total of $\prod_{j=1}^d (M_j - i_j)$ values. Hence, the cost of updating the APS-tree is:

$$\mathcal{C}_{update} = \prod_{j=1}^d (M_j - i_j) \lceil \log_f N \rceil \cdot \mathcal{C}_{\mathcal{H}} + \mathcal{C}_{\mathcal{S}}, \quad (5)$$

where $\mathcal{C}_{\mathcal{S}}$ denotes the cost of a signing operation.

5 The Dynamic Case

The APS-tree is a good solution for non-sparse, static environments because it has very small querying cost. It will not work well though for dynamic settings. In the worst case, updating a single tuple in the database might necessitate updating the whole tree. This section creates advanced structures that overcome this limitation.

5.1 One dimensional Queries: Authenticated Aggregation B-tree

Consider $Q = \langle \text{SUM}(A_q) | S_1 = [a, b] \rangle$, that has one selection predicate with continuous or discrete domain D_1 , where the *distinct number* of values of field S_1 given the tuples contained in the database is $N \leq M_1$. An Authenticated Aggregation B-tree (AAB-tree) is an extended B⁺-tree structure of fanout f with key attribute S_1 , bulk-loaded bottom-up on the base table tuples. AAB-tree nodes are extended with one hash value and one aggregate value per entry. The exact structure of a leaf and an index node is shown in Figure 4. Each leaf entry corresponds to one database tuple t with key $k = t.S_1$, aggregate value $\alpha = t.A_q$, and an associated hash value $\eta = \mathcal{H}(k|\alpha)$. If tuples with duplicate keys exist, then the tree stores only one entry for that key and aggregates all A_q values in α . Hence the AAB-tree has exactly N data entries. Index entries have keys k computed in the same way as

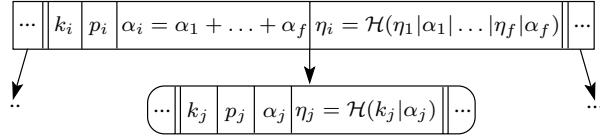


Figure 4: The AAB-tree. On the top is an index node. On the bottom is a leaf node. k_i is a key, α_i the aggregate value, p_i the pointer, and η_i the hash value associated with the entry.

in the normal B^+ -tree and each key is associated with an aggregate value $\alpha = \alpha_1 + \dots + \alpha_f$ (which is the sum of the aggregate values of its children), and a hash value $\mathcal{H}(\eta_1|\alpha_1|\dots|\eta_f|\alpha_f)$, which is a concatenation of both the hash values and the aggregate values of the children.

To locate an entry with key k , a point B^+ -tree query is issued. Authenticating this entry is done in a MHT fashion. The only difference is that the \mathcal{VO} includes both the hash values η and aggregate values α associated with every index entry, and the key values k associated with every leaf entry. In addition, auxiliary information is stored in the \mathcal{VO} , so that the client can find the right location of each hash value during the verification phase. For ease of discussion, we use the same tree encoding scheme as in the previous section (see Figure 5). The only difference is that in an AAB-tree any node could be incomplete and contain fewer than f entries. However, the labelling scheme is imposed on the logical complete tree. As the auxiliary information tells the client at each level where the computed hash value should be placed, this ensures that:

Lemma 2. *The AAB-tree can authenticate both the value associated with the aggregate attribute and the label of any entry in the tree, including entries at the index nodes.*

Next, we present a method to authenticate aggregation queries efficiently using the AAB-tree. The basic idea is that the aggregate information at the index nodes of the tree can be used to answer and authenticate range queries without having to traverse the tree all the way to the leaves. The next two definitions and proposition are not new; they apply to aggregation trees in general, with or without authentication. As we shall see shortly, however, authentication interacts very nicely with the aggregation-related structures.

Definition 1. *The Label Cover \mathcal{LC} of entry $\lambda = \lambda_1 \dots \lambda_l$ is the range of labels of all data entries that have λ as an ancestor. The label cover of a data entry is the label of the entry itself.*

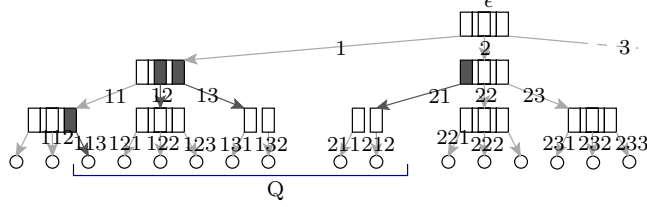


Figure 5: Labelling scheme and the MCS entries.

Given a label λ , the range of labels in its \mathcal{LC} can be computed by padding it with enough 1s to get an h -digit number for the lower bound, and enough f s to get an h -digit number for the upper bound. For example, \mathcal{LC} of $\lambda = 12$ in figure 5 is $\{121, 122, 123\}$.

Definition 2. *The Minimum Covering Set MCS of the data entries in query range S is the set of entries whose \mathcal{LC} s are: 1. Disjoint; 2. Their union covers S completely; 3. Their union covers only entries in S and no more.*

Given the labels λ^-, λ^+ of the entries as the lower and upper bound of Q , $MCS(Q)$ can be computed by traversing the tree top-down and inserting in the MCS all entries whose \mathcal{LC} is completely contained in $[\lambda^-, \lambda^+]$ (and whose ancestors are not in MCS). An entry with \mathcal{LC} that intersects with $[\lambda^-, \lambda^+]$ is followed to the next level. An entry with \mathcal{LC} that does not intersect with $[\lambda^-, \lambda^+]$ is ignored. An example is shown in Figure 5. $\{\lambda^-, \lambda^+\}$ for Q is $\{113, 212\}$. $MCS(Q)$ will be the entries with label $\{113, 12, 13, 21\}$. One can show that:

Proposition 1.

$$\mathcal{ANS}(Q) = \sum_{n \in MCS(Q)} \alpha_n, \quad (6)$$

$$\{\lambda^-, \lambda^+\} \in \bigcup_{n \in MCS(Q)} \mathcal{LC}(n) \in [\lambda^-, \lambda^+], \quad (7)$$

$$\mathcal{LC}(m) \cap \mathcal{LC}(n) = \emptyset, \forall n, m \in MCS(Q), m \neq n. \quad (8)$$

Based on Proposition 1, the authentication of Q can now be converted to the problem of authenticating $MCS(Q)$. Next, we discuss the algorithm for retrieving $MCS(Q)$ and the sibling set STB needed to verify it, in one pass of the tree.

Given a query Q , the server first identifies the labels of the lower λ^- and upper λ^+ bounds of the query range using two point B^+ -tree queries (note

that these labels might correspond to keys with values $a \leq k^-$ and $k^+ \leq b$). Starting from the root, the server follows the following modified algorithm for constructing the MCS , processing entries using a pre-order traversal of the tree. When a node is visited, the algorithm looks at \mathcal{LC} and $[\lambda^-, \lambda^+]$: if \mathcal{LC} is fully contained in $[\lambda^-, \lambda^+]$, then the node is added to the MCS ; if \mathcal{LC} intersects, but is not fully contained in $[\lambda^-, \lambda^+]$, then the node's children are visited recursively; and if \mathcal{LC} and $[\lambda^-, \lambda^+]$ do not intersect at all, then the node's hash value (or key value for leaf nodes) and aggregate value is added to the SIB . In our running example, the hash values (or key values for leaf entries) and the aggregate values of entries $\{111, 112, 22, 23, 3\}$ are included in SIB .

The \mathcal{VO} for the aggregation query contains MCS and SIB . For every node in MCS or SIB , we include its label; this will enable the client to find its correct position in the tree and reconstruct the hash value of its ancestors. Finally, to ensure completeness, the server also includes in the \mathcal{VO} verification information for the two boundary data entries that lie exactly to the left of λ^- and to the right of λ^+ . Denote these entries by λ_l^-, λ_r^+ respectively.⁴

Before discussing the verification algorithm at the client side, we define:

Definition 3. *Two entries (or their labels) are neighbors if and only if: 1. They are at the same level of the tree and no other entry at that level exists in-between, or 2. Their \mathcal{LC} s are disjoint and the left-most and right-most labels in the \mathcal{LC} of the right and left entry respectively are neighbors.*

For example, in figure 5 entries with labels $\{11, 12\}$ are neighbors, same for $\{212, 221\}$. Entries $\{113, 12\}$ are neighbors too (by second part of the definition). An interesting observation is that:

Lemma 3. *All consecutive MCS entries (in increasing order of labels) are neighbors in the tree.*

Proof. Suppose that two consecutive MCS entries m, n are not neighbors. Hence, at some level of the tree there exists an entry p that is a neighbor of m and is not contained in the MCS . Clearly, the \mathcal{LC} of p contains a data entry that is in-between two data entries that belong to the \mathcal{LC} s of m and n . This also stems from two B^+ -tree construction properties: 1. The fact that in an incomplete B^+ -tree the missing subtrees are always the right-most entries of a node and never intermediate entries; 2. Given that p has at

⁴For the left-most and right-most entries in the tree, dummy records are used.

least one data entry as a descendant, otherwise p would have been deleted. Thus, p or a descendant of p should also be an MCS entry since it contains a data entry in the query range. This is a contradiction since m and n are consecutive. \square

The client is able to check whether two entries are neighbors or not if both entries are authenticated. The key point is that the client could infer and authenticate these entries' labels. And with the help of the auxiliary information, which is ensured to be correct if authentication succeeds in previous step, client could check whether it is possible to have another entry in the tree between the two.

Lemma 4. *Given two entries and associated \mathcal{VO} from AAB tree, if the \mathcal{VO} authenticates both entries, client could check whether these two entries are neighbors in the AAB tree or not, given the knowledge of the fanout f .*

Proof. Successful authentication of these two entries provides client with: 1. their labels, by lemma 2; 2. the auxiliary information in \mathcal{VO} is correct and complete, otherwise the authentication should have failed. These two information enables the client performing the check as claimed. If their labels are consecutive to each other, e.g. $\{11, 12\}$ or $\{13, 21\}$ the check is trivial. If their labels are not consecutive, e.g. $\{212, 221\}$, the auxiliary information will help client infer the result. For example, auxiliary information in the \mathcal{VO} for $\{212, 221\}$ should contain information such as the node contains entry 212 has two entries. This effectively eliminates the possibility of the existence of entry 213 in the tree and the client could infer that $\{212, 221\}$ are neighbors. Other cases could be similarly argued. \square

The authentication at the client is a mirror process of that at the server. The client first authenticates boundary entries: $\{\lambda^-, \lambda_l^-, \lambda^+, \lambda_r^+\}$. After successful authentication, the client first checks that $k_l^- < a \leq k^-$ and $k^+ \leq b < k_r^+$ and that the entries $\{k_l^-, k^-\}$ (similarly for $\{k^+, k_r^+\}$) are neighbors. If this is satisfied (otherwise the client rejects the answer), the client derives the labels of $\{\lambda_l^-, \lambda^+\}$. The second step is to verify each entry in the MCS . This is simply a reverse process of the query steps in server side. With the returned \mathcal{VO} , the client can recompute the hash value of the root node and verify it against the signature of the tree. If it fails, the client rejects the answer. Otherwise the client infers the label for each entry in the MCS and check whether consecutive MCS are neighbors or not using lemma 4. If there are consecutive MCS entries that are not neighbors, client rejects the result (missing MCS entry) based on lemma 3. The last

Algorithm 2: AABQUERY(Query Q; AAB-tree T; Stack \mathcal{VO})

```
1 Compute  $[\lambda^-, \lambda^+]$  from Q
2 Recurse(T.root,  $\mathcal{VO}$ ,  $[\lambda^-, \lambda^+]$ )
3 Push information for verifying  $\lambda_l^-, \lambda_r^+$  into  $\mathcal{VO}$ 
  //
4 Recurse(Node  $N$ , Stack  $\mathcal{VO}$ , Range  $R$ ):
5 begin
6    $\mathcal{VO}$ .push(node start);  $\mathcal{VO}$ .push( $N$ .children)
7   for  $N$ .children  $\geq i \geq 1$  do
8     if  $\mathcal{LC}(N[i]) \in R$  then
9       if  $N$  is a leaf then
10         $\mathcal{VO}$ .push( $N[i].k$ );
11      else  $\mathcal{VO}$ .push( $N[i].\eta$ )
12    else if  $\mathcal{LC}(N[i]) \cap R \neq \emptyset$  then
13      Recurse( $N[i]$ ,  $\mathcal{VO}$ ,  $R$ )
14    else  $\mathcal{VO}$ .push( $N[i].\eta$ );
15     $\mathcal{VO}$ .push( $N[i].\alpha$ )
16 end
```

step is to infer the \mathcal{LC} s of all \mathcal{MCS} entries using their labels, and check proposition 1. If it is satisfied, the client simply computes the final result from \mathcal{MCS} . Otherwise, the client rejects the answer.

The complete algorithm for query and \mathcal{VO} construction is presented in algorithm 2 and the algorithm for client side verification is presented by algorithm 3.

The AAB-tree can be used for authenticating one-dimensional aggregate queries in a dynamic setting since the owner can easily issue deletions, insertions and updates to the tree, which handles them similarly to a normal B^+ -tree. In addition, extending the AAB-tree for multiple aggregate attributes A_q can happen similarly to the APS-tree. Other than COUNT and AVG, AAB-tree supports authentication of MIN and MAX as well, simply replacing the SUM aggregate in each entry with the MIN/MAX aggregate. The final answer could be, again, computed and authenticated using \mathcal{MCS} .

Correctness and Completeness: Based on lemmata 2, 3, 4 and proposition 1, AAB-tree ensures the both correctness and completeness.

Algorithm 3: AABAUTHENTICATE(Query Q; Stack \mathcal{VO})

```
1 Retrieve and verify  $\lambda^-, \lambda^+$  from  $\mathcal{VO}$ 
2  $\mathcal{MCS} = \emptyset$ 
3  $\eta = \text{Recurse}(\mathcal{VO}, \mathcal{MCS})$ 
4 Remove entries from  $\mathcal{MCS}$  according to  $[\lambda^-, \lambda^+]$ 
5 Verify neighbor integrity of  $\mathcal{MCS}$  or Reject
6 Verify  $\eta$  or Reject
  //
7 Recurse(Stack  $\mathcal{VO}$ , Stack  $\mathcal{MCS}$ ):
8 begin
9    $c = \mathcal{VO}.\text{pop}()$ 
10   $\eta = \emptyset$ 
11  for  $1 \leq i \leq c$  do
12     $e = \mathcal{VO}.\text{pop}()$ 
13    switch  $e$  do
14       $\lfloor$  case node start:  $\eta = \eta \mid \text{Recurse}(\mathcal{VO}, R)$ 
15       $\alpha = \mathcal{VO}.\text{pop}()$ 
16       $\eta = \eta \mid e \mid \alpha$ 
17       $\mathcal{MCS}.\text{push}(e)$ 
18     $\text{Return } \mathcal{H}(\eta)$ 
19 end
```

5.1.1 Cost Analysis

To authenticate any *aggregate value* either in a leaf entry or an index entry, or the *key* of a leaf entry, in the worst case the \mathcal{VO} constructed by the AAB-tree has size:

$$|\mathcal{VO}| \leq \lceil \log_f N \rceil [f(|\mathcal{H}| + 2|I|) + 2|I|], \quad (9)$$

where N is the distinct number of values in attribute S . In addition, the size of the \mathcal{MCS} can be upper bounded as well. For any key range $[a, b]$:

$$|\mathcal{MCS}| \leq 2(f - 1) \lceil \log_f (b - a + 1) \rceil. \quad (10)$$

The subtree containing all entries in range $[a, b]$ has height $\lceil \log_f (b - a + 1) \rceil$. In the worst case at every level of the tree the \mathcal{MCS} includes $f - 1$ entries for the left sub-range, and $f - 1$ for the right sub-range, until the two paths meet.

Query cost: By combining Equations 9 and 10 the communication cost

can be bounded by:

$$\mathcal{C}_{communication} \leq 2|\mathcal{VO}| + |\mathcal{MCS}| \cdot |\mathcal{VO}|, \quad (11)$$

for the \mathcal{VO} s corresponding to the boundary labels, and the \mathcal{VO} for the \mathcal{MCS} . The verification at the client, counting hashing and verification operations only, is bounded by:

$$\begin{aligned} \mathcal{C}_{verification} \leq & (|\mathcal{MCS}| + \lceil \log_f \frac{N}{b-a+1} \rceil) \cdot \mathcal{C}_{\mathcal{H}} \\ & + 2 \log_f N \cdot \mathcal{C}_{\mathcal{H}} + 3\mathcal{C}_{\mathcal{V}}, \end{aligned} \quad (12)$$

including the hashes for the nodes containing \mathcal{MCS} entries, the remaining hashes in the path to the root, and the authentication cost of the boundary entries.

Storage cost: The size of the AAB-tree is:

$$\mathcal{C}_{storage} = \sum_{l=1}^{\lceil \log_f N \rceil} f^l (|\mathcal{H}| + 4|I|), \quad (13)$$

which includes the hash value, aggregate value, key and one pointer per entry. The AAB-tree has much better space utilization than the APS-tree, especially given that the size of the tree is a function of the base table size and not of the domain size.

Update cost: Updating the AAB-tree is similar to updating a normal B^+ -tree with the additional cost of recomputing the hash values and aggregate values when nodes merge or split. The cost is bounded by:

$$\mathcal{C}_{update} \leq 2\lceil \log_f N \rceil \mathcal{C}_{\mathcal{H}} + \mathcal{C}_{\mathcal{S}}, \quad (14)$$

given the worst case update cost of a B^+ -tree.

5.1.2 Optimizations

A potential optimization for reducing the \mathcal{VO} size of a given range Q , is to authenticate a special complement range of Q . Define the following:

Definition 4. *The Least Covering Ancestors \mathcal{LCA} of the data entries in range Q is the two entries whose \mathcal{LC} s are: 1. Disjoint; 2. Completely cover the data entries in $[\lambda^-, \lambda^+]$ of Q ; 3. their union of \mathcal{LC} is the minimum.*

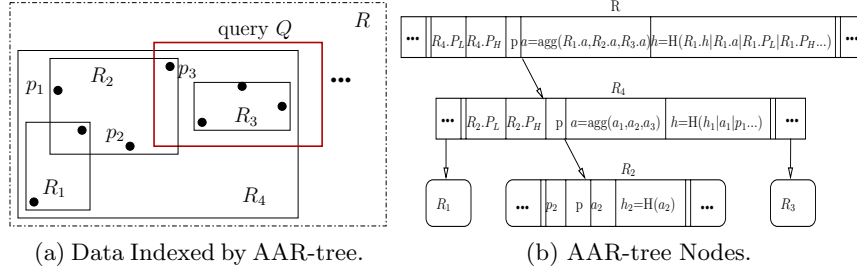


Figure 6: AAR-tree.

It can be shown that set \mathcal{LCA} contains at most two entries in the worst case. Denote with R the range of data entries covered by $\mathcal{LCA}(Q)$. In Figure 5, $\mathcal{LCA}(Q)$ contains entries 1 and 21. Range R covers data entries 111 to 213. Depending on the size of $\mathcal{MCS}(Q)$, it might be beneficial to answer the query by authenticating the aggregate of range R , then the aggregate of range $R-Q$ (denoted by \overline{Q}), and subtract the result for the final answer. It is possible to estimate the size of these sets using statistical information about the average per level utilization of the tree. Hence the server can decide without having to traverse the tree. Furthermore, if the tree is complete, the exact size of these sets can be analytically computed. Nevertheless, for both cases the server first has to run two point B⁺-tree queries for identifying the labels of the boundary entries, which in some cases might negatively affect the server side querying cost.

5.2 Multi-dimensional Queries: Authenticated Aggregation R-tree

The AAB-tree can answer only one-dimensional queries. For the purpose of answering multi-dimensional queries we extend the Aggregate R-tree (AR-tree)[17] to get the Authenticated Aggregation R-tree (AAR-tree).

Let $Q = \langle \text{SUM}(A_q) | S_1 = [a_1, b_1], \dots, S_d = [a_d, b_d] \rangle$, be a d -dimensional aggregate query. AAR-tree indexes all tuples in the base table, according to the selection attributes S_i where $i \in [1, d]$. Every dimension of the tree corresponds to a single attribute, and every node entry is associated with an aggregate value α and a hash value η . The hash value is computed on the concatenation of the entry's children node MBRs m_i , aggregate values α_i and hash values η_i ($\eta = \mathcal{H}(\dots | m_i | \alpha_i | \eta_i | \dots)$). The structure of an AAR-tree node looks the same with that of the AAB-tree in Figure 4 after replacing keys k with MBRs m . The MBR of each entry is included in the hash

computation because the client should have the capability to authenticate the extent of the tree nodes in order to verify completeness of the results, as will be seen shortly. Notice that in a d -dimensional space, the MBR m is simply a d -dimensional rectangle represented by two d -dimensional points. The query Q becomes a d -dimensional query rectangle. An example of AAR-tree is shown in figure 6.

We can define the concept of \mathcal{MCS} similarly to the AAB-tree. It is the minimum set of nodes whose MBRs totally contain the points covered by the query rectangle, not less and not more. The \mathcal{VO} construction is similar to that of the AAB-tree, and uses the concept of computing the answer by authenticating the \mathcal{MCS} entries. Even though correctness verification for any range query can be achieved simply by authenticating the root of the tree, completeness in the AAR-tree requires extra effort by the client. Specifically, the client needs to check if the MBR associated with each node in the \mathcal{VO} intersects or is contained in the query MBR. If it is contained, it belongs in the \mathcal{MCS} . If it intersects, then the client expects to have already encountered a descendant of this node which is either fully contained in the query range or disjoint. This check is possible since the MBRs of all entries are included in the hash computation. The server has to return all MBRs of the entries encountered during the querying phase in order for the client to be able to recompute the hash of the root, and to be able to check for completeness. Therefore, the \mathcal{VO} contains all the MBRs (with their hash values), for all the nodes of the R-tree visited during the search procedure. Extending the AAR-tree to support multi-aggregate queries can be achieved with the techniques discussed for the APS-tree and AAB-trees.

Correctness and Completeness: The method that we describe above gives an authentication procedure that guarantees correctness and completeness. The basic idea behind proving this, is the fact that the server has to authenticate *every* entry of the the AAR-tree that it accesses in order to answer the query. The proof is a special case of [19, Theorem 3], which holds for more general structures (any DAG with a single entry node) and can be directly applied to the AAR-tree method that we use here.

5.2.1 Cost Analysis

Query cost: Let an AAR-tree indexing N d -dimensional points, with average fan-out f and height $h = \log_P(\frac{N}{f})$ (where P is the page size, and once more the level of the root being zero and the conceptual level of the data

entries being h). The size of the \mathcal{VO} for authenticating one AAR-tree entry at level l (equivalent to a node at level $l+1$), either a data entry or an index entry, is upper bounded by:

$$|\mathcal{VO}| \leq fl[2d \cdot |I| + |I| + |\mathcal{H}|], \quad (15)$$

(assuming that MBRs are represented by $|I|$ -byte floating point numbers). The cost is equal to the level of the entry times the amount of information needed for computing the hash of every node on the path from the entry to the root.

The size of the \mathcal{MCS} is clearly related to the relationship between the query range MBR q and the MBRs of the tree nodes. Let m be a node MBR, and $P(m \odot q)$ represent the probability that the query MBR fully contains m . Assuming uniformly distributed data, it has been shown in [15] that

$$P(m \odot q) = \begin{cases} \prod_{j=1}^d (q_j - m_j) & , \text{ if } \forall j : q_j > m_j \\ 0 & , \text{ otherwise.} \end{cases}$$

where q_j, m_j represent the length of q and m on the j -th dimension. Furthermore, it has been shown in [16] that the probability of intersection between q and m is $P(m \oplus q) = \prod_{j=1}^d (q_j + m_j)$. Thus, the probability of an intersection but not containment is equal to $P(m \ominus q) = P(m \oplus q) - P(m \odot q)$. Let m_l be the average length of the side of an MBR at the l -th level of the tree. It has been shown by [33] that $m_l = \min \{(f^{h-l}/N)^{1/d}, 1\}$, $0 \leq l \leq h-1$, which enables us to estimate the above probability.

The expected number of nodes at level l is $N_l = \frac{N}{f^{h-l}}$. It can be shown that $J_l = J_{l-1} \cdot P(m \ominus q)$, $1 \leq l \leq h$, $J_0 = f \cdot P(m \ominus q)$ is the number of nodes that intersect with query q at level l , given that all its ancestors also intersect with q . The size of the \mathcal{MCS} is upper bounds by $|\mathcal{MCS}| \leq [N_1 + \sum_{l=0}^{h-1} J_l] \cdot P(m \odot q)$. Essentially, we estimate the number of children entries that are contained in the query, for every node that intersects with the query at a given level.

Hence, the communication cost in the worst case can be expressed by the following estimate:

$$\mathcal{C}_{communication} \leq |\mathcal{MCS}| \cdot |\mathcal{VO}|. \quad (16)$$

The verification cost is similarly bounded by:

$$\mathcal{C}_{verification} \leq |\mathcal{MCS}| \cdot \mathcal{C}_{\mathcal{H}} + \mathcal{C}_{\mathcal{V}}. \quad (17)$$

Storage cost: The storage cost is:

$$C_{storage} = \sum_{l=0}^{h-1} \frac{N}{f^{h-l}} \cdot f \cdot [|\mathcal{H}| + 2d \cdot |I| + |I| + |I|], \quad (18)$$

since we extend every entry with a hash value and an aggregate value, and include the d -dimensional MBRs and one pointer per entry.

Update cost: Updating the AAR-tree is similar to updating an R-tree. Hence:

$$C_{update} = \log_P\left(\frac{N}{f}\right) \cdot C_{\mathcal{H}} + C_{\mathcal{S}}. \quad (19)$$

6 Extensions

This section extends our discussion to other interesting topics that are related to the problem. So far, we have presented solutions for SUM queries which can support multiple aggregated predicates, with multiple selection predicates on discrete and continuous domains.

6.1 Other Aggregates

COUNT is a special case of SUM and is thus handled similarly. The combination of SUM and COUNT provides a solution for AVG as well. AAB-tree and AAR-tree can be modified to support MIN and MAX queries, simply by replacing the aggregate values stored in the index nodes of the trees, with the MIN/MAX of their children. The APS-tree cannot handle MIN/MAX aggregates. Authentication of holistic aggregates, like MEDIAN, is much harder and left as future work.

6.2 Handling Encrypted Data

In some scenarios it might be necessary for the owner to publish only encrypted versions of its data for privacy preservation purposes [11, 2]. It should be made clear that an encrypted database does not provide a solution to the query authentication problem — the servers could still purposely omit from the results tuples that actually satisfy the query conditions. It

is interesting though to mention that the APS-tree can work without modifications with encrypted data as long as the client knows the encryption key. The server does not need to access the data or perform any computations or comparisons on the data. It only needs to retrieve the encrypted data at a specific location of the one-dimensional prefix sums array, which can be provided by the client. On the other hand, the AAB-tree and the AAR-tree structures cannot support encrypted data in a strong sense. Using homomorphic encryption, such as [13], we can allow the server to compute aggregate values without learning any information. However, the difficulty arises in enabling the server to traverse the index structure for finding the data entries contained in the query result. This requires the server to perform comparisons on encrypted data, which by definition reveals too much information from a security point of view. Hence, we do not consider here techniques like order-preserving encryption [1] to be applicable in our setting.

6.3 Query Freshness

Query freshness was introduced by [18], and is a problem that stems from the fact that in dynamic environments the servers, having obtained a correct authenticated structure, may choose to ignore further updates from the owners and answer client queries on stale data. In this situation the client has no way of knowing that the query answers are not fresh. Various solutions for solving this problem are based on certain signature certificate techniques. Li et al. [18] show that the cost of solving the query freshness problem is proportional to the number of signatures used to authenticate the structure. Since all of the techniques proposed here utilize only one signature, query freshness can easily be addressed using the same techniques. For further details the reader is referred to [18].

7 Performance Evaluation

In this section we evaluate the performance of the proposed approaches with respect to query, authentication, storage, and update cost. We implemented the APS-tree, AAB-tree and AAR-tree as part of the Authenticated Index Structures Library which can be download from here [3]. We also evaluate the only known previous solutions that can be used for authenticating aggregation queries, namely the authentication structures for selection queries.

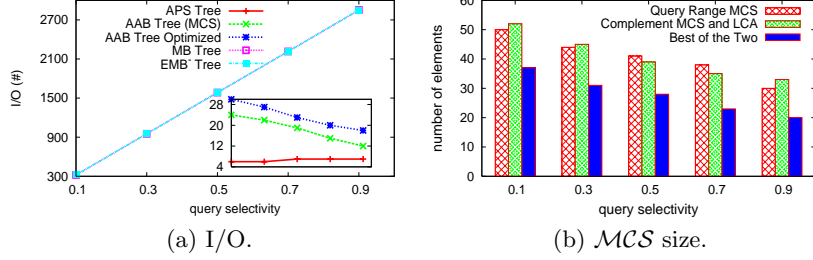


Figure 7: One-dimensional queries. Server-side cost.

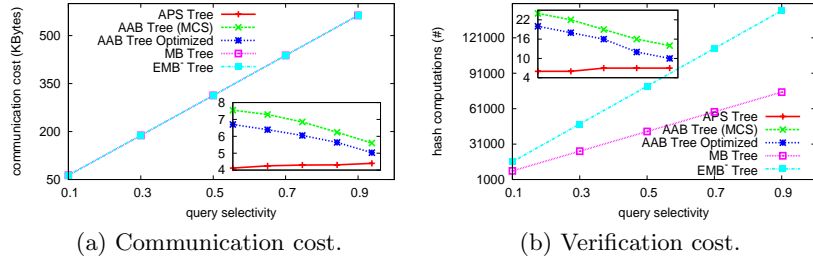


Figure 8: One-dimensional queries. Communication and verification cost.

7.1 Setup

We use synthetic datasets for our experiments. We generate d -dimensional tuples, with multiple sizes for the attribute domains D_i , of decimal values that are generated uniformly at random (the distribution of the data does not affect our authentication techniques, only the underlying structures). We also vary the density δ of the data, where $\delta = \frac{N}{\prod_{i=1}^d M_i}$, and N the number of tuples. We also generate synthetic query workloads with one aggregate attribute A_q and up to 3 selection attributes. Every workload contains 100 queries, and we report averages in the plots. All experiments are performed on a Linux box with a 2.8GHz Intel Pentium4 CPU. The page size of the structures is set to be 1KByte. We use the OpenSSL [29] and Crypto++ [7] libraries for hashing, verification and signing operations (SHA1 and RSA, respectively).

7.2 One-dimensional Queries

First, we evaluate the structures for one-dimensional queries. Candidates are the APS-tree, the AAB-tree, and the structures for selection queries,

like the MB-tree, and EMB^- -tree [18]. For the naive approaches in order to authenticate a query, first we answer the range query and report all values to the client, which then reconstructs the result. For the AAB-tree we evaluate both the structure based on \mathcal{MCS} and the optimization based on \mathcal{LCA} . We generate a database with domain size of $M = 100,000$, $N = \delta M$ and vary the density of the database $\delta \in [0.1, 0.9]$, as well as the query selectivity $\rho \in [0.1, 0.9]$.

Figure 7(a) shows the I/O cost at the server side. The best structure overall is the APS-tree, since it only needs to verify two PS entries, with a cost proportional to the height of the tree. The naive approaches are one to two orders of magnitude more expensive than the other techniques, since they need to do a linear scan on the leaf pages, which increases the cost as the queries become less selective. For the AAB-trees it is interesting to note that the optimized version has slightly higher query I/O, due to the extra point queries that are needed for retrieving the query range boundaries. In this case, reducing the size of the \mathcal{VO} did not reduce the I/O overall. In addition, for both AAB-tree approaches the cost decreases as queries become less selective, since the \mathcal{MCS} (and its complement $\overline{\mathcal{MCS}} \cup \mathcal{LCA}$) become smaller at the same time due to larger aggregation opportunities. This is clearly illustrated in Figure 7(b) that shows the actual sizes of \mathcal{MCS} , $\overline{\mathcal{MCS}} \cup \mathcal{LCA}$, and the best of the two for the average case over 100 queries.

Figures 8(a) and 8(b) show the communication cost and verification cost at the client side. For the communication cost we observe similar trends with the \mathcal{MCS} size, since the size of the \mathcal{VO} is directly related to the size of \mathcal{MCS} . Notice also that the optimized version has smaller communication cost. The same is true for the verification cost for the APS-tree and AAB-tree. For the naive approaches the communication cost increases linearly with the query selectivity, which is simply explained by the fact that they have to return ρN number of aggregate values and keys. The verification cost is several orders of magnitude slower in terms of hash computations, due to the fact that all the results in the query range need to be authenticated, which is an overhead when queries are not very selective.

The query efficiency of the APS-tree is achieved with the penalty of high storage and update cost. This is indicated in Figures 9(a) and 9(b). For this set of experiments we vary the database density δ . The storage cost of the APS-tree depends only on the domain sizes and is not affected by δ . The other approaches have storage that increases linearly with δ . Notice that, as expected, for very dense datasets all the trees have comparable storage cost. AAB-tree consumes slightly more space than MB-tree and EMB^- -tree, the

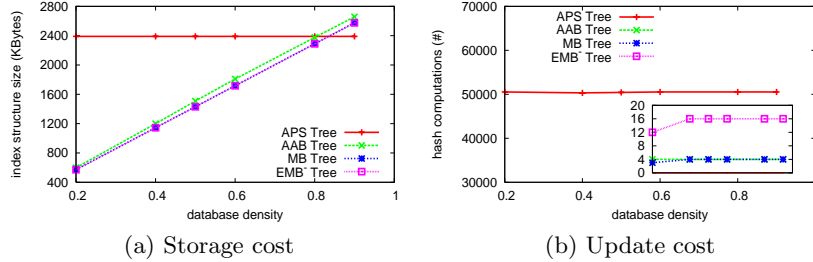


Figure 9: One-dimensional queries. Storage and update cost.

reason is being that it has to store aggregate values for entries in index nodes. For the update experiment, we uniformly at random generate 100 updates and report the average. The update cost is measured in number of hash computations required. The APS-tree has to update half of the data entries on average. For the AAB-tree and MB-tree the update cost is bounded by the height of the tree. The EMB-tree has slightly increased cost due to the embedded trees stored in every node that conceptually increase the height of the tree. We can see that the AAB-tree has competitive storage and update cost comparing to the naive approaches, and is orders of magnitude better than the APS-tree in terms of update cost.

7.3 Multi-dimensional Queries

In this section we compare the APS-tree and AAR-tree approaches for 3-dimensional queries. The naive approaches can be used for answering multi-dimensional queries by constructing one tree for every dimension, querying all the trees, and sending back all the results along with the necessary \mathcal{VO} . Finally, the client verifies the result at each dimension and finds the intersection of the results. This approach becomes extremely expensive in high-dimensional spaces, so we do not consider it here. To have similar database size as the experimental study in one-dimensional queries, we generate synthetic datasets with maximum of unique elements of $\prod_{i=1}^3 M_i = 125,000$ tuples, query workload with 100 queries with varying values of ρ on database with $\delta = 0.8$, the storage and update costs are studied for databases with different density values of δ .

The I/O cost of the structures as a function of query selectivity is reported in Figure 10(a). The APS-tree once again has the best performance

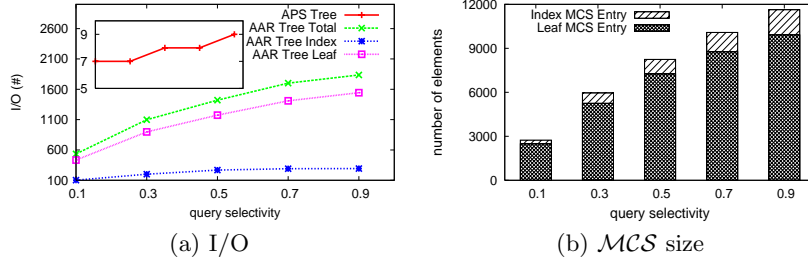


Figure 10: 3-dimensional queries. Server-side cost.

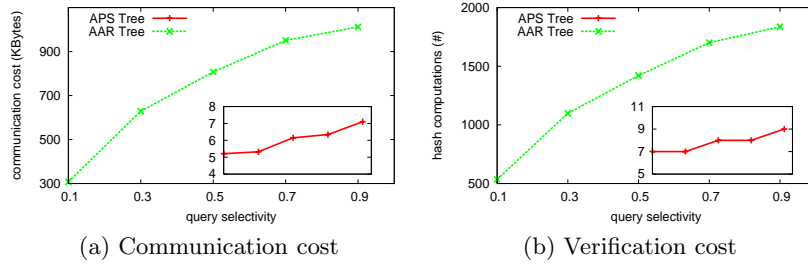


Figure 11: 3-dimensional queries. Communication and verification cost.

overall, since it only needs to authenticate eight PS elements. The AAR-tree has much higher I/O since it needs to construct the MCS which requires traversing many different paths of the R-tree structure, due to multiple MBR overlaps at every level of the tree. Notice also that the I/O of the AAR-tree increases as queries become less selective, since larger query rectangles result into a larger number of leaf node MBRs included in the MCS as indicated in Figure 10(b). This becomes clear in Figure 10(a) which shows that most I/O operations come from the leaf level of the tree.

The authentication and verification costs are shown in Figure 11. The APS-tree has small communication cost and verification cost, both bounded by eight times the height of the tree; notice that our algorithm by merging the common paths has reduced the costs from this worst case bound. The AAR-tree has much higher communication cost due to larger MCS sizes and because it has to return the MBRs of all MCS entries. The verification costs follow similar trends, since the number of hash computations is directly related to the number of entries in the MCS .

Figure 12(a) shows the storage cost as a function of δ , for the APS-tree and AAR-tree. In higher dimensions the AAR-tree consumes more space when the database becomes relatively dense, in contrast to the one-

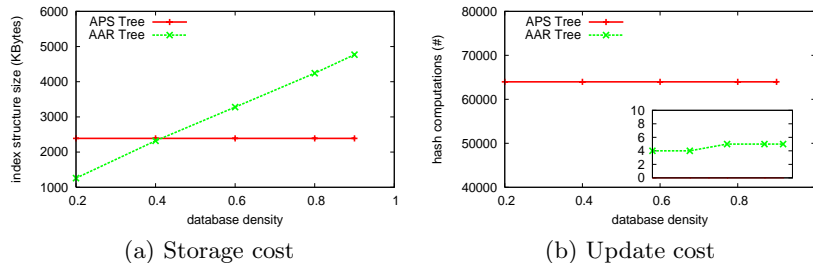


Figure 12: 3-dimensional queries. Storage and update cost.

dimensional case. This is due to the fact that the AAR-tree has to store the 3-dimensional MBRs for all nodes. In our experiments we use 8-byte floating point numbers and a small page size, but the trend is indicative.

Figure 12(b) plot the update cost as a function of δ . The superior query cost of the APS-tree is offset by its extremely high update cost. For 100 updates generated uniformly at random, regardless of database density the APS-tree has to update half of the data entries on average. In contrast, the AAR-tree inherits the good update characteristics of the R-tree.

7.4 Discussion

Our experimental results clearly indicate the efficiency of the proposed authenticated aggregation index structures over the straightforward approaches. In general, our approach has multiple orders of magnitude smaller query cost with almost the same storage and update cost as the existing approaches. Among the new techniques proposed, the APS-tree has very small query cost but expensive updates, and considerable space overhead in the case of sparse datasets. The AAB-tree and AAR-tree have higher query cost but better space usage, especially for sparse datasets, and superior update cost.

8 Conclusion

In this paper, we proposed several authenticated indexing schemes for aggregation queries. We provided a structure with excellent query performance in static environments. However, it has increased space utilization for sparse databases and high update overhead. Therefore, we presented structures for dynamic settings that gracefully adapt to data updates and have better space utilization for sparse datasets. We also showed how to extend these

techniques to handle multiple aggregates and multiple selection predicates per query. For future work, we plan to explore solutions for holistic aggregates and investigate the application of our techniques to authenticate data cubes in OLAP system.

References

- [1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *SIGMOD*, pages 563–574, 2004.
- [2] R. Agrawal, R. Srikant, and D. Thomas. Privacy preserving OLAP. In *SIGMOD*, pages 251–262, 2005.
- [3] Authenticated Index Structures Library. <http://cs-people.bu.edu/lifeifei/aisl/>.
- [4] E. Bertino, B. Carminati, E. Ferrari, B. Thuraisingham, and A. Gupta. Selective and authentic third-party distribution of XML documents. *TKDE*, 16(10):1263–1278, 2004.
- [5] W. Cheng, H. Pang, and K. Tan. Authenticating multi-dimensional query results in data publishing. In *DBSec*, 2006.
- [6] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
- [7] Crypto++ Library. <http://www.eskimo.com/~weidai/cryptlib.html>.
- [8] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic third-party data publication. In *IFIP Workshop on Database Security (DBSec)*, pages 101–112, 2000.
- [9] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):96–99, April 1988.
- [10] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.
- [11] H. Hacigümüs, B. R. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database service provider model. In *SIGMOD*, pages 216–227, 2002.

- [12] H. Hacigümüs, B. R. Iyer, and S. Mehrotra. Providing database as a service. In *ICDE*, pages 29–40, 2002.
- [13] H. Hacigümüs, B. R. Iyer, and S. Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In *DASFAA*, pages 125–136, 2004.
- [14] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *SIGMOD*, pages 73–88, 1997.
- [15] M. Jürgens and H. Lenz. Pisa: Performance models for index structures with and without aggregated data. In *SSDBM*, pages 78–87, 1999.
- [16] I. Kamel and C. Faloutsos. On packing R-Trees. In *CIKM*, pages 490–499, 1993.
- [17] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *SIGMOD*, pages 401–412, 2001.
- [18] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*, 2006.
- [19] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.
- [20] K. McCurley. The discrete logarithm problem. In *Proc. of the Symposium in Applied Mathematics*, pages 49–74. American Mathematical Society, 1990.
- [21] R. C. Merkle. A certified digital signature. In *CRYPTO*, pages 218–238, 1989.
- [22] G. Miklau. *Confidentiality and Integrity in Data Exchange*. PhD thesis, University of Washington, 2005.
- [23] G. Miklau and D. Suciu. Implementing a tamper-evident database system. In *ASIAN*, 2005.
- [24] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In *NDSS*, 2004.

- [25] E. Mykletun, M. Narasimha, and G. Tsudik. Signature bouquets: Immutability for aggregated/condensed signatures. In *ESORICS*, pages 160–176, 2004.
- [26] E. Mykletun and G. Tsudik. Aggregation queries in the database-as-a-service model. In *DBSec*, 2006.
- [27] M. Narasimha and G. Tsudik. Dsac: Integrity of outsourced databases with signature aggregation and chaining. In *CIKM*, pages 235–236, 2005.
- [28] National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard*. National Institute of Standards and Technology, 1995.
- [29] OpenSSL. <http://www.openssl.org>.
- [30] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. In *SIGMOD*, pages 407–418, 2005.
- [31] H. Pang and K.-L. Tan. Authenticating query results in edge computing. In *ICDE*, pages 560–571, 2004.
- [32] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *CACM*, 21(2):120–126, 1978.
- [33] Y. Theodoridis and T. K. Sellis. A model for the prediction of r-tree performance. In *PODS*, pages 161–171, 1996.