Authenticated Index Structures for Aggregation Queries

Feifei Li, Florida State University
Marios Hadjieleftheriou, AT&T Labs Research
George Kollios, Boston University
Leonid Reyzin, Boston University

Query authentication is an essential component in outsourced database (ODB) systems. This article introduces efficient index structures for authenticating aggregation queries over large data sets. First, we design an index that features good performance characteristics for static environments. Then, we propose more involved structures for the dynamic case. Our structures feature excellent performance for authenticating queries with multiple aggregate attributes and multiple selection predicates. Furthermore, our techniques cover a large number of aggregate types, including distributive aggregates (such as SUM, COUNT, MIN and MAX), algebraic aggregates (such as the AVG), and holistic aggregates (such as MEDIAN and QUANTILE). We have also addressed the issue of authenticating aggregation queries efficiently when the database is encrypted to protect data confidentiality. Finally, we implemented a working prototype of the proposed techniques and experimentally validated the effectiveness and efficiency of our methods.

Categories and Subject Descriptors: H.2 [**Database Management**]:

General Terms: Algorithms, Performance, Security

Additional Key Words and Phrases: Aggregation, Authentication, Indexing, Outsourced Databases

1. INTRODUCTION

The latest decade has witnessed tremendous advances in computer systems, networking technologies and, as a consequence, the Internet, sparkling a new information age where people can access and process data remotely, accomplish critical tasks from the leisure of their own home, or do business on the go. In order to guarantee availability, reliability, and good performance for a variety of network services, service providers are forced to distribute data and services across multiple servers at the edge of the network, not necessarily placed under their direct con-

F. Li, Computer Science Department, Florida State University, Tallahassee, FL , 32306. M. Hadjieleftheriou, AT&T Labs Research, Florham Park, NJ, 07932. G. Kollios, Computer Science Department, Boston University, Boston, MA, 02215. L. Reyzin, Computer Science Department, Boston University, Boston, MA, 02215. This work is based on our technical report published in August 2006 [Li et al. 2006a]. The initial submission was on September 2008. The first major revision was on August 2009 and the final acceptance was on February 2010. Feifei Li was supported in part by NSF CNS-0831278 grant. George Kollios and Leonid Reyzin were partially supported by NSF CNS-0831281 grant.

trol, e.g., the cloud computing framework. Remotely accessing data through the network inherently raises important security issues. Servers are prone to hacker attacks that can compromise the legitimacy of the data residing therein and the processing performed. The chances of even a meticulously maintained server being compromised by an adversary should not be underestimated, especially as the application environment and interactions become more complex and absolute security becomes harder, if not impossible, to achieve. This problem is exacerbated by the fact that time is in favor of attackers, since scrutinizing a system in order to find exploits proactively is an expensive and difficult task.

An adversary gaining access to the data is free to deceive users who query the data, possibly remaining unobserved for a substantial amount of time. For non-critical applications, simple audits and inspections will eventually expose any attack. On the other hand, for business critical or life critical applications such sampling based solutions are not good enough. Consider a client taking business critical decisions on financial investments (e.g., mutual funds and stocks) by accessing information from what he considers to be a trusted financial institution. A single illegitimate piece of information might have catastrophic results when large amounts of money are at stake. Similar threats apply to online banking applications and more. In these examples information can be tampered with not only due to a compromised sever, but also due to insecure communication channels and untrusted entities that already have access to the data (e.g., system administrators).

It is reasonable to assume that in critical applications users should be provided with absolute security guarantees on a per transaction basis, even at a performance cost. Such demands cannot be satisfied by putting effort and money into developing more stringent security measures and audits — in the long run subversion is always possible. Hence, a fail-safe solution needs to be engineered for tamper-proofing the data against all types of manipulation from unauthorized individuals at all levels of a given system, irrespective of where the data resides or how it was acquired, as long as it has been endorsed by the originator. Techniques for guaranteeing against illegitimate processing need to be put forth. It is exactly in this direction that this work focuses on, and more precisely, on query authentication: assuring end-users that query execution on any server, given a set of authenticated data, returns correct results, i.e., unaltered and complete answers (no answers have been modified, no answers have been dropped, no spurious answers have been introduced). Correctness is evaluated on a per query basis. Every answer is accompanied by a verification proof — an object that can be used to reconstruct the endorsement of the data originator, and built in a way that any tampering of the answer will invalidate an attempt to verify the endorsement. Verification objects are constructed using a form of chained hashing, using ideas from Merkle trees [Merkle 1980] (cf. Section 3). A by-product of ensuring query authentication is that the data is also protected from unauthorized updates, i.e., changes made by individuals that do not have access to the secret key used to endorse the data. Data owners may update their databases periodically and, hence, authentication structures should support dynamic updates. An important dimension of query authentication in a dynamic environment is result freshness. Notice that an adversary acquiring an older version of an authenticated database can provide what may appear to be correct answers,

that contain in reality outdated results. Authenticated structures should protect against this type of attack as well.

The query authentication problem has been recently examined by a variety of works (see the detailed discussion in the related work section). Noticeably, existing literature concentrated on authenticating selection, projection and join (SPJ) queries, e.g., "Retrieve all stocks with prices in the range \$300-\$600".

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 stocks sold with price between \$300 and \$600". 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 efficiently remains an open problem.

In this work, we formally define the aggregation authentication problem and 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 aggregate, we show that in static scenarios authenticating aggregation queries is equivalent to authenticating prefix sums [Ho et al. 1997]. 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 and R-tree structures. Finally, we extend the techniques for aggregates other than SUM, and discuss some issues related to query freshness and data encryption for protecting the data confidentiality and data privacy. 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, MAX, any QUANTILE and MEDIAN.

The rest of the paper is organized as follows. Section 2 gives the formal problem definition. Section 3 presents the background and related work. Section 4 discusses static outsourced database scenarios, while Section 5 presents the dynamic case. Section 6 generalizes the discussion for aggregates other than SUM, and Section 7 discusses issues related to query freshness and data encryption. An empirical evaluation is presented in Section 8. Finally, Section 9 concludes the paper.

2. PROBLEM DEFINITION

Consider the following SQL statement:

```
SELECT SUM(sales) FROM stocks WHERE price>$300 and price<$600
```

This statement contains one **aggregated** attribute (sales) and one **selection** attribute (price), in the form of a range predicate. In general, any aggregation query can be represented as follows: $Q = \langle \otimes(A_1), \cdots, \otimes(A_c) | S_1, \cdots, S_d \rangle$, where $\bigotimes$ is the associated aggregation operation, A_is correspond to the aggregated attributes and S_js to the selection attributes in the predicates of the query. Attributes A_i, S_j may correspond to any fields of a base table $\mathbf{T}$ and they could refer to the same field. For simplicity and without loss of generality, we assume that the schema of $\mathbf{T}$ consists of fields $\mathbf{T}(S_1, \ldots, S_d)$ with domains $D_1, \ldots, D_d$, respectively. Both the

aggregated attributes and the selection attributes are a subset of these fields. 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., AVG $\equiv$ SUM/COUNT. Holistic aggregates (like MEDIAN and QUANTILE) are harder to compute in general as they require knowing all values in the query attribute w.r.t. the query range. In this work, we firstly concentrate on distributive aggregates. Algebraic aggregates are easily computed once distributive aggregates are addressed. Then, we extend our techniques to authenticate holistic aggregates in Section 6.

We focus 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 (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 is defined as follows. A data owner compiles some authenticated structures for its data that are disseminated along with its database to 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: query cost, that includes the server's query execution, 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

The Merkle hash tree (MHT) [Merkle 1980] (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 (using a collision-resistant hash function). 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 published signature of the root. Given the collision resistance property of the hash function and the guarantee of the public-

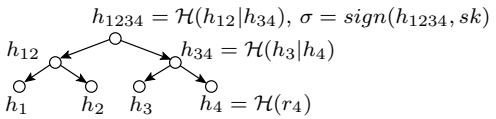


Fig. 1. A Merkle hash tree: $\mathcal{H}$ is a collision resistant hash function; sk is the owner's secret key.

key signature scheme, it is computationally infeasible for an adversary (in certain computational models) to fool the verifier by modifying any of the data in the path from the leaf to the root.

Most previous work on query authentication has focused on studying the general selection and projection queries. The proposed techniques can be categorized into two groups: the signature-based approaches [Pang et al. 2009; Pang et al. 2005; Mykletun et al. 2004a; Narasimha and Tsudik 2005] by using the aggregation signature techniques [Mykletun et al. 2004b] and the index-based approaches [Devanbu et al. 2000; Li et al. 2006b; Miklau 2005; Yang et al. 2008; Martel et al. 2004; Papadopoulos et al. 2009; Mouratidis et al. 2009; Pang and Mouratidis 2008; Singh and Prabhakar 2008; Atallah et al. 2008] by generalizing the merkle hash tree to an f-way tree and embedding it into various indexes. Injecting random records by the data owner into the database has also been proposed [Xie et al. 2007], that uses a probabilistic approach for query authentication and hence it is more flexible and easier to realize in practice. However, unlike other work in query authentication, it does not guarantee absolute correctness for query authentication.

Query authentication in multi-dimensional spaces has been studied by extending the signature-based [Narasimha and Tsudik 2005; Cheng et al. 2006] and the index-based [Yang et al. 2008; 2009] approaches. In particular, Yang et al. [Yang et al. 2008; 2009] integrated an R-tree with a merkle hash tree for authenticating multi-dimensional range queries. However, they did not address aggregation queries.

Recent studies have also addressed other challenging problems in query authentication, such as dealing with XML documents [Bertino et al. 2004], text data [Pang and Mouratidis 2008], join queries [Yang et al. 2009; Pang et al. 2009] and handling dynamic updates [Pang et al. 2009; Xie et al. 2008] efficiently. Improvements to existing techniques were proposed, such as separating the authentication from query execution when there is a trusted third party [Papadopoulos et al. 2009], or partially materializing the authenticated data structures to reduce the cost [Mouratidis et al. 2009]. The index-based approach has been implemented for the PostgreSQL database [Singh and Prabhakar 2008]. Query authentication for outsourced databases has also been studied in the context of streaming data model [Papadopoulos et al. 2007; Li et al. 2007]. However, none of these works has explored aggregation queries.

There has been some work on main memory authenticated data structures following the work of Naor and Nissim [Naor and Nissim 1998]. However, these works [Anagnostopoulos et al. 2001; Goodrich et al. 2003; Tamassia and Triandopoulos 2005; 2007; Goodrich et al. 2008; Papamanthou et al. 2008; Goodrich et al. 2010] focus on main memory structures and are not directly applicable to external memory databases.

The problem of computing aggregations over encrypted databases using homomorphic encryption has been studied in [Mykletun and Tsudik 2006; Ge and Zdonik 2007]. For most of this work, we consider unencrypted databases; one of our

proposed techniques does provide query authentication efficiently over encrypted databases and this issue is discussed in Section 7.1. Note that an encrypted database does not provide query authentication, but it guarantees data confidentiality, i.e., except the data owner and the clients that have the authorized decryption key, no one else can obtain the content for any plain record in the database.

Besides data confidentiality, other security goals can be considered in parallel with query authentication in the ODB framework. For example, it is possible to implement access control policies. How to efficiently authenticate query results without violating access control policies is an interesting and challenging problem. Pang et al. [Pang et al. 2005] has addressed this problem for the selection query; Kundu et al. [Kundu and Bertino 2008] introduced the concept of structure confidentiality while providing integrity for a tree-based index, where the goal is to enable an user to authenticate a subtree S from a tree T without leaking any other nodes from $T - S$ ("$-$" refers to the "cut" operation). In this work, we do not consider issues related with access control policies.

An orthogonal problem is query execution assurance, studied by Sion [Sion 2005] where the goal is to provide guarantees on the amount of effort the server has spent in query execution. This work, though, does not guard against malicious servers that can return incorrect query results, and hence does not provide authentication.

3.1 A Trivial Solution

Any solution for authenticating selection queries could provide a straightforward but very inefficient solution for authenticating aggregation queries. The server answers the aggregation query Q as selection queries and returns $\mathcal{SAT}(Q)$ along with the $\mathcal{VO}$ for the selection queries. The client authenticates the set $\mathcal{SAT}(Q)$ and then computes the aggregation locally. Note that by using any existing techniques for authenticating selection queries, such as the approach from [Li et al. 2006b; Pang et al. 2005], the client is able to verify both the correctness and the completeness of the result. Hence, the aggregate computed locally by the client will be the correct answer if the result of the corresponding selection query has been authenticated.

However, this approach is not desirable because: 1. The communication and verification costs are linear in $|\mathcal{SAT}(Q)|$ (e.g., if the query is a `SELECT *` statement the cost might be prohibitive); 2. The cost for multi-dimensional aggregation queries is even higher in the relative sense compared to an approach that authenticates aggregation directly. It is thus desirable to design a solution that: 1. Has communication/verification cost sub-linear in $|\mathcal{SAT}(Q)|$; 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 $D_j = [0, M_j)$ (for continuous or categorical domains existing values are just ordered and assigned distinct identifiers), and query $Q = \langle \text{SUM}(A_q) | S_1 = [a_1, b_1], \ldots, S_d = [a_d, b_d] \rangle$. Each tuple in the database can

be viewed as a point in a d-dimensional space $D_1 \times \cdots \times D_d$, and the selection query as a d-dimensional range query. The d-dimensional space can be reduced to a $|D_1| \times \cdots \times |D_d|$ 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 $\sum_{i_1=a_1}^{b_1} \cdots \sum_{i_d=a_d}^{b_d} C[i_1, \ldots, i_d]$. Answering the query requires accessing $\prod_{i=1}^{d}(b_i - a_i + 1)$ elements.

Alternatively, a prefix sums array can be used [Ho et al. 1997]. The prefix sum array PS of C has the same structure as C and in every coordinate it stores: $\forall x_j \in D_j, j \in [1, d]$:

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

In other words, an entry $PS[x_1, \ldots, x_d]$ in the prefix sum array PS stores the total sum of all entries before $C[x_1, \ldots, x_d]$, including $C[x_1, \ldots, x_d]$ itself.

It has been shown in [Ho et al. 1997] 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 $PS[x_1, \ldots, x_d] = 0$ if $x_j = -1$; then:

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

We use some examples to illustrate the meaning of Equation 1. When $d = 1$, each entry in the prefix sum array simply stores the sum of all preceding elements in the database up to this entry. Hence, a range sum query $Q = \langle \text{SUM}(A_q) | [a_1, b_1] \rangle$ can be simply answered by $PS[b_1] - PS[a_1 - 1]$. When $d = 2$, the range sum query $Q = \langle \text{SUM}(A_q) | [a_1, b_1], [a_2, b_2] \rangle$ can be computed by PS as follows:

$$PS[b_1, b_2] - PS[b_1, a_2 - 1] - PS[a_1 - 1, b_2] + PS[a_1 - 1, a_2 - 1].$$

Intuitively, in two-dimension, given the prefix sum array, a range sum query could be answered by considering only the four corner points in the prefix sum array from the rectangle area defined by the query range. Similarly, in d-dimensions, one only needs to consider the 2^d corner points in the prefix sum array from the d-dimensional hyper-cube defined by the query range.

Having computed PS for the aggregation attribute A_q, any query $Q = \langle \text{SUM}(A_q) | S_1, \ldots, S_d \rangle$ 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 PS into a one-dimensional array PS', where element $PS[i_1, \ldots, i_d]$ corresponds to element $PS'[k], k = \sum_{j=1}^{d-1}(i_j \times \prod_{n=j+1}^{d} M_n) + i_d$, and build an f-way MHT on top of PS'. We call this structure the authenticated prefix sums tree (APS-tree).

Suppose that a single element $PS'[k]$ needs to be authenticated. Traversing the tree 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

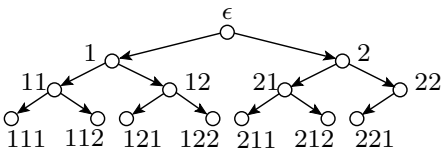


Fig. 2. The tree encoding scheme.

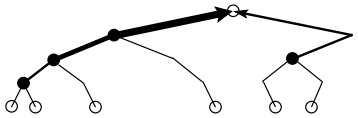


Fig. 3. Merging the overlapping paths. At every black node the remaining hashes can be inserted in the $\mathcal{VO}$ only once for all verification paths.

integer in a base-f number system. The root node has no label. The 1st level nodes have base-f representations $1, 2, \ldots f$, from left to right respectively. The 2nd level nodes have labels $11, \ldots, 1f, 21, \ldots, 2f, \ldots, f1, \ldots, ff$, and so on all the way to the leaves (see an example in Figure 2 with $f = 2$). Straightforwardly, a leaf entry with PS' 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 $\lfloor k/f \rfloor$ leaf node, and this leaf node lies in the $\lfloor k/f^2 \rfloor$ index node one level up, and so on. Retrieving $PS'[k]$ is possible now by following the node with label that is the prefix of the label 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 $PS'[k]$, 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 to 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 the signature on the tree root is unforgeable, it can be shown that any change to the structure of the APS-tree (such as tampering with the content of any index or leaf nodes, deleting an existing node or inserting a new node), or the structure of a constructed $\mathcal{VO}$ (such as changing the sequence of objects in the $\mathcal{VO}$, modifying the content of any object in the $\mathcal{VO}$, inserting or deleting objects to or from the $\mathcal{VO}$) will cause the authentication procedure to fail, in exactly the same way as for the normal MHT. The fact that the 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', 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 (note that domain sizes and fanout of the tree are static information, both could be authenticated only once directly from the owner). Essentially, the client needs to authenticate each of the 2^d elements of the answer set. First, the client 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 also infers the position k for each

element in PS' based on its encoding and f. Next, using the query ranges $[a_i, b_i]$ and the domain sizes, the client 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 (according to Equation 1) 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.

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 cost 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, $\mathcal{SAT}(Q)$ 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.

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

```
1  VO = ∅, SAT = ∅, h = T.height
2  VO.push(h), VO.push(domain sizes)
3  for k^x = i_1, ..., i_d ∈ {a_1 − 1, b_1}, ..., {a_d − 1, b_d} do
4      Compute matrix K = λ_1^1 ··· λ_h^n for keys k^1, ..., k^n
5  Compute G_1 = {11, ..., 1x} using K
   // set G_1 contains x groups named 11, ..., 1x
6  for S ∈ G_1 do
7      Recurse(root, 2, S, K)
8  Recurse(Node N, Level l, Set S, Matrix K):
9  begin
10     Compute G = {S1, ..., Sy} using K
       // group names will become {111, ..., 11z}, {1111, ..., 111w}, and so on
11     VO.push(l), VO.push(N.children - |G|)
12     for λ_l ≡ S' ∈ G do
           // for λ_ls corresponding to each S' in G
13         VO.push(λ_l)
14         if l = h then SAT.push(N[λ_l].k)
           // value λ_h^x is the offset of key k^x in the leaf
15     for 1 ≤ i ≤ N.children do
16         if i ≠ λ_l, ∀λ_l ∈ G then VO.push(N[i].η)
17     if l < h then
18         for λ_l ≡ S' ∈ G do Recurse(N[λ_l], l + 1, S')
19 end
```

Let $k^1, \ldots, k^n$ be the indices of the PS' values that need to be authenticated. The construction algorithm essentially computes the base-f numbers corresponding to indices $k^1, \ldots, k^n$ 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 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 $PS'[k]$ 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 reduce storage requirements 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 in). The APS-tree can be used to authenticate COUNT and AVG as well, as COUNT is a special case of SUM and AVG is authenticated as SUM/COUNT.

4.2 Cost Analysis

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:

$$\mathcal{C}_{communication} = |\mathcal{VO}| + |\mathcal{SAT}| \leq \sum_{j=1}^{\lceil \log_f M \rceil} [(f - |G_j|) \cdot |\mathcal{H}| + (|G_j| + 1) \cdot \Delta] + 2^d \Delta,$$

where M $(= M_1 \times \ldots \times M_d)$ is the size of the PS' array, $|\mathcal{H}|$ is the size of a hash value, Δ the size of the largest prefix sum 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:

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

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:

$$\mathcal{C}_{storage} = \sum_{l=0}^{\lceil log_f M \rceil} f^l(|\mathcal{H}| + \Delta) + M\Delta,$$

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 $PS[i_1, \ldots, i_d]$ is updated. Then, elements $PS[x_1, \ldots, x_d]$ 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:

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

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 number of tuples contained in the database is $N \leq M_1$ (recall that M_1 is the domain size of attribute S_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. The bulk-loading is done in exactly the same fashion as that of the classical bulk-loading algorithm for the normal B$^+$-tree, however, with the hash and aggregate values for entries in the leaf and index nodes computed as follows. AAB-tree nodes are extended with one hash value and one aggregate value per entry. The exact structures of a leaf and an index node are 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 $\eta = \mathcal{H}(k|\alpha)$. Hence the AAB-tree has exactly N data entries. Index entries have keys k computed in the same way as in the normal B$^+$-tree and each key is associated with an aggregate value $\alpha = \alpha_1 + \ldots + \alpha_f$ (which is the sum of the aggregate values of its children), and a hash value $\mathcal{H}(\eta_1|\alpha_1| \ldots |\eta_f|\alpha_f)$, which is to compute the hash value over the 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

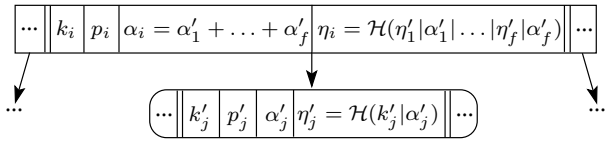


Fig. 4. The AAB-tree. On the top is an index node. On the bottom is a leaf node. k_i is a key, α_i the aggregate value, p_i the pointer, and η_i the hash value associated with the entry.

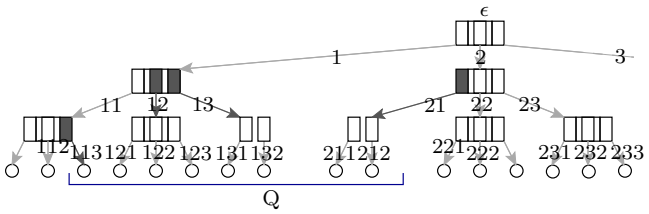


Fig. 5. Labeling scheme and the $\mathcal{MCS}$ entries.

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 labeling 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 of 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 $\lambda = \lambda_1 \cdots \lambda_l$ 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.*

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 fs 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 $\mathcal{MCS}$ of the* data entries *in query range S is the set of entries whose $\mathcal{LC}$s satisfy: 1. They are 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, $\mathcal{MCS}(Q)$ can be computed by traversing the tree top-down and inserting in the

$\mathcal{MCS}$ all entries whose $\mathcal{LC}$ is completely contained in $[\lambda^-, \lambda^+]$ (and whose ancestors are not in $\mathcal{MCS}$). 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 $\{113, 212\}$. $\mathcal{MCS}(Q)$ will be the entries with label $\{113, 12, 13, 21\}$. One can show that:

PROPOSITION 1. *Given λ^-, λ^+ as the labels for entries of the lower and upper bound of Q,*

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

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

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

Based on Proposition 1, the authentication of Q can now be converted to the problem of authenticating $\mathcal{MCS}(Q)$. Next, we discuss the algorithm for retrieving $\mathcal{MCS}(Q)$ and the sibling set $\mathcal{SIB}$ 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 $\mathcal{MCS}$, processing entries using a pre-order traversal of the tree. When a node is visited, the algorithm looks at its $\mathcal{LC}$ and $[\lambda^-, \lambda^+]$: if its $\mathcal{LC}$ is fully contained in $[\lambda^-, \lambda^+]$, then the node is added to the $\mathcal{MCS}$; if its $\mathcal{LC}$ intersects, but is not fully contained in $[\lambda^-, \lambda^+]$, then the node's children are visited recursively; and if its $\mathcal{LC}$ 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 $\mathcal{SIB}$. 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 $\mathcal{SIB}$.

The $\mathcal{VO}$ for the aggregation query contains $\mathcal{MCS}$ and $\mathcal{SIB}$. For every node in $\mathcal{MCS}$ or $\mathcal{SIB}$, 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 (for the left-most and right-most entries in the tree, dummy records are used).

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 (since the left-most label, 121, in the $\mathcal{LC}$ of entry 12, is a neighbor of entry 113). An interesting observation is that:

LEMMA 3. *All consecutive entries in $\mathcal{MCS}$ (in the increasing order of their labels) are neighbors in the tree.*

PROOF. Suppose that two *consecutive* $\mathcal{MCS}$ 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 $\mathcal{MCS}$. 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. p has at least one data entry as a descendant, since otherwise p would have been deleted. Thus, p or a descendant of p should also be an $\mathcal{MCS}$ entry since it contains a data entry in the query range. This is a contradiction since m and n are consecutive. □

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 the previous steps, the 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, the client can verify 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 the 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. This information enables the client performing the verification as claimed. If their labels are consecutive to each other, e.g. $\{11, 12\}$ or $\{13, 21\}$ the verification is trivial. If their labels are not consecutive, e.g. entries $\{212, 221\}$ from Figure 5, the auxiliary information from the $\mathcal{VO}$ must have included the information that the leaf node containing entries $\{211, 212\}$ contains only two entries. This effectively helps the client eliminate the possibility of the existence of an entry 213 in the tree. Given the fanout f ($f = 3$ in this example), the client could immediately infer that $\{212, 221\}$ are neighbors. Other cases could be similarly argued. □

The authentication at the client is a mirror process of that at the server. The client first authenticates boundary entries: $\{\lambda^-, \lambda_l^-, \lambda^+, \lambda_r^+\}$ and identifies the corresponding keys, e.g, k^+ is the key for label λ^+, etc. 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 $\mathcal{MCS}$. This is simply a reverse process of the query steps at the 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 $\mathcal{MCS}$ and checks whether consecutive $\mathcal{MCS}$ are neighbors or not using Lemma 4. If there are consecutive $\mathcal{MCS}$ entries that are not neighbors, the client rejects the result (missing $\mathcal{MCS}$ entry) based on Lemma 3. The last 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 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 in Algorithm 3.

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

```
1  Compute [λ⁻, λ⁺] from Q
2  Recurse(T.root, VO, [λ⁻, λ⁺])
3  Push information for verifying λ_l⁻, λ_r⁺ into VO
   //
4  Recurse(Node N, Stack VO, Range R):
5  begin
6      VO.push(node start); VO.push(N.children)
7      for N.children ≥ i ≥ 1 do
8          if LC(N[i]) ∈ R then
9              if N is a leaf then
10                 VO.push(N[i].k);
11             else VO.push(N[i].η)
12         else if LC(N[i]) ∩R ≠ ∅ then
13             Recurse(N[i], VO, R)
14         else VO.push(N[i].η);
15         VO.push(N[i].α)
16 end
```

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

```
1  Retrieve and verify λ⁻, λ_ℓ⁻, λ⁺, λ_r⁺ from VO
2  MCS = ∅
3  η = Recurse(VO, MCS)
4  Remove entries from MCS according to [λ⁻, λ⁺]
5  Verify neighbor integrity of MCS or Reject
6  Verify η or Reject
   //
7  Recurse(Stack VO, Stack MCS):
8  begin
9      c = VO.pop()
10     η = ∅
11     for 1 ≤ i ≤ c do
12         e = VO.pop()
13         switch e do
14             case node start:  η = η| Recurse(VO, R)
15         α = VO.pop()
16         η = η|e|α
17         MCS.push(e)
18     Return H(η)
19 end
```

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 Lemma 2, 3, 4 and Proposition 1, AAB-tree ensures correctness and completeness.

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:

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

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

$$|\mathcal{MCS}| \leq 2(f-1)\lceil \log_f (b-a+1) \rceil. \tag{3}$$

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 2 and 3 the communication cost can be bounded by:

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

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:

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

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:

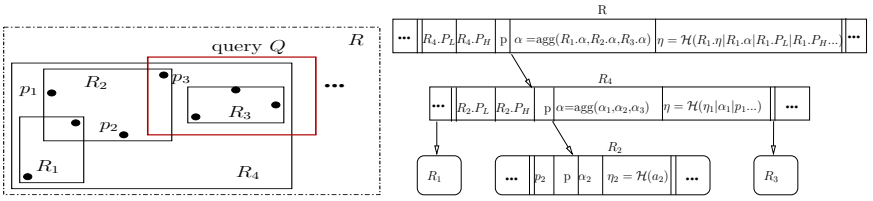
$$\mathcal{C}_{storage} = \sum_{l=1}^{\lceil log_f N \rceil} f^l(|\mathcal{H}| + 4I),$$

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, 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:

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

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



(a) Data Indexed by AAR-tree. (b) AAR-tree Nodes.

Fig. 6. AAR-tree.

5.1.2 *Optimization.* A potential optimization to reduce 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 satisfy the following: 1. They are disjoint; 2. They completely cover the data entries in $[\lambda^-, \lambda^+]$ of Q; 3. Their union has the minimum number of entries covered.*

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 212. 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

For the purpose of answering multi-dimensional queries in the dynamic environment we extend the Aggregate R-tree (AR-tree)[Lazaridis and Mehrotra 2001; Tao and Papadias 2004] to get the Authenticated Aggregation R-tree (AAR-tree).

Let $Q = \langle \text{SUM}(A_q)|S_1 = [a_1, b_1], \ldots, S_d = [a_d, b_d]\rangle$, 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 (minimum bounding rectangles) m_i, aggregate values α_i and hash values η_i ($\eta = \mathcal{H}(\ldots|m_i|\alpha_i|\eta_i|\ldots)$). 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 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 process. 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 described above gives an authentication procedure that guarantees correctness and completeness. The basic idea behind proving this, is that the server has to authenticate *every* entry of the AAR-tree that it accesses in order to answer the query. The proof is a special case of [Martel et al. 2004, 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.

5.2.1 *Cost Analysis.* Next we present the cost model for the AAR-tree for authentication of aggregation queries in the multi-dimensional space.

Query cost: Let an AAR-tree indexing N d-dimensional points, with average fanout 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:

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

(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 [Jürgens and Lenz 1999] 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 [Kamel and Faloutsos 1993] 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 [Theodoridis and Sellis 1996] 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 for any MBR m. In particular, let $P_l(m \ominus q)$, $P_l(m \oplus q)$ and $P_l(m \odot q)$ be the probabilities for an MBR m from the l-th level of tree.

Clearly, $J_l = J_{l-1} \cdot f \cdot P_l(m \ominus q), 1 \leq l \leq h, J_0 = P_0(m \ominus q)$ is the number of nodes that intersect with query q at level l, given that all its ancestors also intersect with q. The expected size of the $\mathcal{MCS}$ can be estimated as $|\mathcal{MCS}| = \sum_{l=0}^{h} J_l \cdot f \cdot P_l(m \odot q)$. In the special case when the root node is fully contained by the query, i.e., $P_0(m \ominus q) = 0$, $\mathcal{MCS} = 1$. Essentially, we estimate the number of children entries that are contained by the query, for every node that intersects with the query at a given level (and all its ancestors intersect with, but are not contained by, the query).

Hence, the expected communication cost can be estimated by:

$$\mathcal{C}_{communication} = |\mathcal{MCS}| \cdot |\mathcal{VO}|.$$

The verification cost is similarly estimated by:

$$\mathcal{C}_{verification} = |\mathcal{MCS}| \cdot \mathcal{C}_{\mathcal{H}} + \mathcal{C}_{\mathcal{V}}.$$

Storage cost: Every entry has a hash value and an aggregate value, and includes the d-dimensional MBRs and one pointer per entry. Hence, the storage cost is:

$$\mathcal{C}_{storage} = \sum_{l=0}^{h-1} \frac{N}{f^{h-l}} \cdot f \cdot [|\mathcal{H}| + 2d \cdot I + I + I],$$

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

$$\mathcal{C}_{update} = \log_P(\frac{N}{f}) \cdot \mathcal{C}_{\mathcal{H}} + \mathcal{C}_{\mathcal{S}}.$$

6. OTHER AGGREGATES AND AUTHENTICATING SELECTION QUERIES

So far, we have presented solutions for SUM queries which can support multiple aggregated predicates, with multiple selection predicates on discrete and continuous domains. Next, we extend to other aggregate queries (including holistic aggregates, such as the MEDIAN and QUANTILE). We also show how our index structures support selection queries without the need of making any changes.

Support for Authentication of Selection Queries: Both the AAB-tree and AAR-tree also support the authentication of range selection queries without the need to make any changes to their structures. Essentially, by adding the sibling hash values for the two boundary query paths to the $\mathcal{VO}$, with all leaf entries in the query range (they are returned to the client since it is a range selection query), the client is able to authenticate this result by reconstructing the subtree that covers the query range (the corresponding aggregate values in any entry can be easily computed, and hence the hash values), and eventually the hash value of the root node with the help of the sibling hash values from the boundary query paths. The overall correctness and completeness are established by the reconstructed root hash value and the root signature in the $\mathcal{VO}$.

COUNT, AVG and MIN/MAX: COUNT is a special case of SUM and is thus handled similarly. The combination of SUM and COUNT provides a solution

for AVG. AAB-tree and AAR-tree can 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.

Multiple Aggregates in One AAB-(AAR-)tree: A very appealing feature for the AAB-tree and APS-tree is that they can authenticate multiple aggregates simultaneously using just one index. For instance, we can make an AAB-tree support the authentication of both COUNT and SUM simultaneously. In order to achieve this, we modify the structure for entries in the AAB-tree node. For an index node, each entry simply stores two pairs of $(aggregate, hash)$, instead of just one. One pair is for the COUNT aggregate and the other is for the SUM aggregate. Every such pair is computed as illustrated in Figure 4, however, using the pairs that correspond to the same aggregate from its children nodes. For a leaf node, each entry also stores two pairs of $(aggregate, hash)$, one for each aggregate. The hash value in each pair is computed by applying the hash function on the concatenation of the key value for this leaf entry and the corresponding aggregate value for this entry. Finally, two signatures are produced at the root level, one for each aggregate. The authentication procedure is done in exactly the same fashion as that in the tree with just one aggregate type, however, using only the hash values for the querying aggregate to construct the $\mathcal{VO}$. The case for the AAR-tree is similar.

Authentication for MEDIAN and QUANTILE: Given the discussion above, we can finally present our solution to the authentication of holistic aggregates, like MEDIAN and QUANTILE. The trivial solution as outlined in Section 3.1 can obviously be applied, by treating the range QUANTILE query as a range selection query. Then, let the client authenticate the result of the selection query first and find the QUANTILE locally. However, as argued in Section 3.1, this approach suffers high communication and verification costs.

Since MEDIAN is a special case of QUANTILE (i.e., it is the 50% QUANTILE), we concentrate on the general QUANTILE case. Note that QUANTILE is well-defined for one-dimensional data. In higher dimensions, QUANTILE cannot be directly defined on the original data, unless some score function is specified to turn them into one dimensional data. Hence, in this work we concentrate on the classical QUANTILE definition on one-dimensional data (i.e., only one aggregate attribute). We use $\theta(A_q)$ to denote a θ-quantile value from attribute A_q, for $\theta \in [0, 1]$, and illustrate our idea using one selection attribute. In this case, a range aggregate query has the following form:

$$Q = \langle \theta(A_q) | S_1 = [a_1, b_1] \rangle, \tag{4}$$

which asks for the θ-quantile w.r.t. attribute A_q for all records whose values for the attribute S_1 are within the query range $[a_1, b_1]$.

Since any quantile requires a total ordering on the attribute A_q, there could be an ambiguity in the definition when there are duplicate values among records on their A_q field. Typically, such ambiguity is addressed by applying the standard geometric perturbation method, i.e., adding a tiny amount of random perturbation to each record's A_q value so that they are guaranteed to be distinct and a total ordering could be derived. At the same time, the perturbations are small enough so that they will not affect the ordering of records that have different A_q values to

start with. For example, for three records with their A_q values as $\{7, 7, 8\}$. They could be transformed to $\{7.01, 7.05, 8.02\}$ by the geometric perturbation and a total ordering is clearly derived. In general, any tie-breaking method by the data owner could be used first; then, the derived order of tuples is explicitly imposed using geometric perturbation. Note that such an approach has no impact to the storage cost, the query cost and the authentication cost for the query authentication in the ODB system. It also does not affect the accuracy of the query result. The data owner and the clients could easily agree on a pre-defined threshold value τ, so that once the client has authenticated the query result from the server, the client can then remove the last τ digits from the query answer to get the precise record as it was before the geometric perturbation.

That said, we assume that the aggregate attribute for the range quantile query has distinct values for all records in the database. Before discussing the solution for the general quantile query as shown in Equation 4, we first address a special case when $A_q = S_1$, i.e., when the aggregate attribute and the selection attribute are the same field. When this is the case, the θ-quantile query becomes:

$$Q = \langle \theta(A_q) | A_q = [a_1, b_1] \rangle, \tag{5}$$

We next show that the AAB-tree provides a solution to this query for any value of $\theta \in [0, 1]$. First, the data owner builds an AAB-tree $\mathcal{T}$, with both the indexing attribute and aggregate attribute as A_q. The aggregate type used in the AAB-tree construction is COUNT. $\mathcal{T}$ is then forwarded to the server and used by the server to answer queries and construct the $\mathcal{VO}$. When a client would like to request a θ-quantile query, as shown in Equation 5, it sends two queries to the server:

$$(1)\ Q_1 = \langle \text{COUNT}(*) | A_q = [a_1, b_1] \rangle, (2)\ Q_2 = \langle \theta(A_q) | A_q = [a_1, b_1] \rangle.$$

The server needs to answer both queries, but only construct $\mathcal{VO}_1$ for Q_1. Suppose the server finds c_1 as the answer to Q_1 and v as the answer to Q_2 (v can be found easily by a leaf-level sequential scan starting from a_1 in the tree $\mathcal{T}$ to b_1). It then has to answer the following two queries:

$$(3) Q_3 = \langle \text{COUNT}(*) | A_q = [a_1, v) \rangle, (4) Q_4 = \langle \text{COUNT}(*) | A_q = [v, v] \rangle,$$

The server needs to find both the answers and the corresponding $\mathcal{VO}$'s for queries Q_3 and Q_4 — suppose they are $(c_3, \mathcal{VO}_3)$ and $(c_4, \mathcal{VO}_4)$ respectively. Next, the server returns $\{(c_1, \mathcal{VO}_1), (c_3, \mathcal{VO}_3), (c_4, \mathcal{VO}_4), v\}$ to the client. Note that the client understands the protocol and hence knows the syntax for queries Q_3 and Q_4. Given all of the above, the authentication step at the client is rather straightforward. It first authenticates the correctness of c_1, c_3, c_4 using $\mathcal{VO}_1$, $\mathcal{VO}_3$, $\mathcal{VO}_4$ and the query range from the syntax of Q_1, Q_3, Q_4 respectively, by the AAB-tree's authentication algorithm (Algorithm 3). If all of them are valid (otherwise the client rejects the answer), it then checks the following:

$$(a)\ \lfloor \theta c_1 \rfloor = c_3 \text{ or } \lceil \theta c_1 \rceil = c_3, (b)\ c_4 = 1.$$

If both conditions hold, the client then accepts the answer v as the correct answer to the θ-quantile query Q_2.

The correctness of this scheme is almost immediate. First, the server cannot manipulate the values of c_1, c_3, c_4 without being caught by the authentication guar-

antee of the AAB-tree $\mathcal{T}$. Second, given that the values for c_1, c_3, c_4 are correct for their corresponding queries Q_1, Q_3, Q_4, condition (a) asserts that there are θ percent of records in the query range $[a_1, b_1]$ that are ranked before the value v; condition (b) then asserts that the value v indeed exists in the database. Note that the verification by the condition (b) is necessary, otherwise the server could return all correct values for c_1, c_3, c_4, but an incorrect value v' as the θ-quantile. The query Q_4 could also be constructed using a `SELECTION` query instead, i.e., Q_4 =`SELECT * FROM` $\mathbf{T}$ `WHERE` $A_q = v$, $\mathbf{T}$ is the base table. Since the AAB-tree can authenticate the `SELECTION` query without any changes (as discussed at the beginning of this section), this approach is also possible.

The additional challenge for the general θ-quantile query, as the one shown in Equation 4 when $A_q \neq S_1$, is that records in the AAB-tree are ordered according to their attribute S_1, but the θ-quantile verification needs to authenticate the order for records within the query range w.r.t. their attribute A_q. Our idea outlined above for the special case can still be applied, with the help of one AAB-tree and one AAR-tree, $\mathcal{T}_1$ and $\mathcal{T}_2$. The AAB-tree $\mathcal{T}_1$ has S_1 as the indexing attribute and A_q as the aggregate attribute. It stores *two aggregate types*, both the COUNT and the MIN, as shown at the beginning of this section how AAB-tree and AAR-tree support the authentication of multiple aggregates in one tree. The AAR-tree $\mathcal{T}_2$ is in the 2-dimensional space defined by attributes S_1 and A_q, and stores/authenticates the COUNT aggregate w.r.t. A_q.

For a θ-quantile query as shown in Equation 4, the client sends the following two queries to the server:

$$(1)\ Q_5 = \langle \text{COUNT}(*) | S_1 = [a_1, b_1] \rangle, (2)\ Q_6 = \langle \theta(A_q) | S_1 = [a_1, b_1] \rangle.$$

The server answers both queries with tree $\mathcal{T}_1$. For Q_5, it finds the answer c_5 and also constructs $\mathcal{VO}_5$. For Q_6, it first finds all records with their attribute S_1 having the values in the range $[a_1, b_1]$ (using a range selection query on $\mathcal{T}_1$, but there is no need to construct the $\mathcal{VO}$ for it), then sorts these records based on their attribute A_q and identifies the θ-quantile as value v. The server also finds the *smallest value* $v_{\min}$ from these records w.r.t. the attribute A_q; it needs to authenticate this result to the client. Hence, a query Q_7 is used to find $\mathcal{VO}_7$ for $v_{\min}$, using again tree $\mathcal{T}_1$ ($\mathcal{T}_1$ stores, hence can answer and authenticate, both COUNT and MIN on A_q):

$$(3)\ Q_7 = \langle \text{MIN}(A_q) | S_1 = [a_1, b_1] \rangle.$$

Next, the server processes the following queries using tree $\mathcal{T}_2$:

$$(4) Q_8 = \langle \text{COUNT}(*) | A_q = [v_{\min}, v), S_1 = [a_1, b_1] \rangle, (5) Q_9 = \langle \text{COUNT}(*) | A_q = [v, v], S_1 = [a_1, b_1] \rangle$$

and finds their answers and the corresponding $\mathcal{VO}$'s, suppose they are $(c_8, \mathcal{VO}_8)$ and $(c_9, \mathcal{VO}_9)$ respectively. Finally, the server returns $\{(c_5, \mathcal{VO}_5), (v_{\min}, \mathcal{VO}_7), (c_8, \mathcal{VO}_8), (c_9, \mathcal{VO}_9), v\}$ to the client.

Since the client understands the protocol, it can easily reconstruct the syntax for queries Q_7, Q_8, Q_9. It first authenticates $c_5, v_{\min}, c_8, c_9$ with the help of $\mathcal{VO}_5, \mathcal{VO}_7, \mathcal{VO}_8, \mathcal{VO}_9$ and the corresponding query range syntax from Q_5, Q_7, Q_8, Q_9 respectively. Next, it simply checks:

$$(a)\ \lfloor \theta c_5 \rfloor = c_8 \text{ or } \lceil \theta c_5 \rceil = c_8, (b)\ c_9 = 1.$$

If both conditions hold, the client accepts v as the correct answer for Q_6. The correctness of this approach is similarly argued as that for the special case when $A_q = S_1$. The only additional trick is that the client has to authenticate the validity of $v_{\min}$, which is achieved with the help of Q_7 and $\mathcal{VO}_7$.

An alternative solution when S_1 and A_q are different is to only use the AAR-tree $\mathcal{T}_2$. But in this case we store both COUNT and MIN w.r.t. A_q in $\mathcal{T}_2$. Clearly, the server can still answer Q_5, Q_6 and Q_7 using only $\mathcal{T}_2$ and build $\mathcal{VO}_5$, $\mathcal{VO}_7$ for Q_5 and Q_7, by adding a selection range on A_q to Q_5, Q_6 and Q_7 as $A_q = [d_l, d_r]$, where d_l (d_r) is the smallest (largest) value for the domain of A_q. We can assume that the client knows d_l and d_r as well (since they are static values that never change, the data owner can broadcast them once to all clients during the system set-up). The rest of the algorithm stays unchanged (except that the client now uses the authentication method from the AAR-tree). This solution eliminates the need of building and maintaining the AAB-tree $\mathcal{T}_1$, however, the query and authentication costs for Q_5, Q_6 (no authentication cost) and Q_7 are expected to be much higher than the previous solution (2D queries vs. 1D queries).

The cost analysis of our method for authenticating the θ-quantile query is straightforward, given the cost analysis for AAB-tree and AAR-tree. In the special case, it uses just one AAB-tree. Its authentication cost is equal to three range COUNT queries; its query cost for the server is equal to three range COUNT queries, plus the sequential scan in the leaf level starting at a_1 to find the θ-quantile which is inexpensive. The storage, update, and communication costs are the same as one AAB-tree. In the general case, it uses either one AAB-tree and one AAR-tree, or just one AAR-tree. Its authentication cost and query cost can be similarly derived (three range COUNT queries and one range MIN query). The storage, update and communication costs are equivalent to one AAB-tree plus one AAR-tree, or just one AAR-tree. Note that the one more query cost in the range quantile query — the server has to answer a range selection query (Q_2 or Q_6) — is almost necessary in order to answer a quantile query even without the authentication requirement, unless some other pre-processing methods or indices have been used. Hence, this cost is also insignificant, especially given that it can be efficiently answered by the AAB-tree (just like answering a range selection query using an B^+-tree).

The above idea easily generalizes to range quantile queries with one aggregate attribute and more than one selection attributes. We simply use one AAR-tree in the multi-dimensional space defined by the aggregate attribute and all selection attributes. The rest of the algorithm follows the same methodology.

7. ENCRYPTED DATABASES AND QUERY FRESHNESS

We next address issues related to encrypted databases and query freshness.

7.1 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 or protecting the data confidentiality [Hacigümüs et al. 2002; Agrawal et al. 2005]. 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.

APS-tree can work without modifications with encrypted data as long as the client knows the encryption key. Since the APS-tree indexes the data only based on their indices in the one dimensional array which maps the prefix sum array, and the authentication algorithm in the APS-tree only requires and depends on a few (2^d if the data is in a d-dimensional space) indices of the prefix sum array (which the user can easily infer based on the query condition without knowing either the structure or the content of the database), the data confidentiality is perfectly preserved in the APS-tree if the data owner uses any standard encryption scheme to encrypt each record in the database. Furthermore, this does no have any impact regarding query performance (both the query cost and the authentication cost) of the APS-tree on the server side even if the database is encrypted. As 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.

In general, MHT will reveal certain information for the underlying data if the index was built *directly* on the original data content (either the plaintext or the encrypted version). However, the APS-tree is built based on the indices on the prefix-sum array. In other words, the APS-tree will potentially reveal the index values in the prefix-sum array of the retrieved, encrypted element, but nothing else about the content of this element since the comparison in the APS-tree is done w.r.t. the index values in the prefix-sum array and not on the content of the elements. Hence, this information does not help to decrypt the plain-text value of the encrypted element.

On the other hand, the AAB-tree and the AAR-tree structures cannot support encrypted data. Using homomorphic encryption [Hacigümüs et al. 2004], 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 leaks information on plain-texts from a security/privacy point of view. Hence, we do not consider here techniques like order-preserving encryption [Agrawal et al. 2004].

7.2 Query Freshness

Query freshness 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. It has been shown 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.

8. 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. We also evaluate the previous solutions that can be used

for authenticating aggregation queries, namely the authentication structures for selection queries.

8.1 Setup

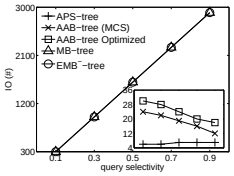
First, in all experiments, we use the density δ of the database to control how many records are in the database and how dense the database is with respect to its domain space (this is necessary in order to study the performance of the APS-tree, since the size of the prefix sum array depends on the size of the domain). The database density δ is defined as $\delta = \frac{N}{\prod_{i=1}^{d} M_i}$. Recall that N is the number of tuples in the database and M_i is the domain size for the ith dimension.

For all experiments related to query performance, we use the query selectivity, denoted as $\rho \in [0, 1]$, to control the query range. For a database with N tuples in any dimension d, the query selectivity for a range aggregate query Q is defined as $|\mathcal{SAT}(Q)|/N$, i.e., the number of tuples that satisfy the query range over N.

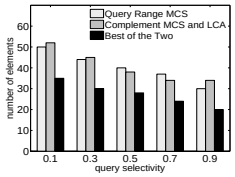
We use both real and synthetic datasets. The real data sets were obtained from the open street map project. Each data set contains the road network and streets for a state in the USA. Each point has its longitude and latitude coordinates. We normalized each data set into the space $L = (0, 0) \times (10^5, 10^5)$. For our experiments, we have used the Texas (TX) and California (CA) data sets. The TX data set has 14 million points and the CA data set has 12 million points. The real data sets are in two dimensions. We used them for the one-dimensional space to evaluate the performance of the APS-tree and AAB-tree. Given a database density value δ, since the one-dimensional domain has a size 10^5, we randomly sample $\delta \times 10^5$ number tuples from the CA or TX data sets, and retain their X coordinate values as the indexing attribute and the Y coordinate values as the aggregate attribute.

For synthetic data sets in d-dimensional space, we generate d-dimensional tuples, with different sizes M_i for the attribute domains D_i, of values that are generated based on the following two distributions: uncorrelated uniform distribution (UU) or the randomly clustered distribution (RC). The number of tuples we generate is based on the database density δ, i.e., $N = \delta \times \prod_{i=1}^{d} M_i$. We also generate synthetic query workloads with one aggregate attribute A_q and up to three selection attributes. The query range is controlled by the query selectivity as discussed above. Every workload contains 100 queries, and we report averages in the plots. All experiments were performed on a Linux box with a 2.8GHz Intel Pentium4 CPU. The page size of the structures is set to 1KByte. We use the OpenSSL [Levitte et al.] and Crypto++ [Dai] libraries for hashing, verification and signing operations (SHA1 and RSA, respectively).

The distribution of the datasets does not affect our authentication techniques. It only affects the base indexing structures. Furthermore, for the storage cost, the query cost and the authentication cost, they are determined by the database density δ and the query selectivity ρ. Hence, different data sets actually have similar results if using the same δ and ρ parameters. Therefore, in one-dimensional space, we only report the result for the CA data set. For three-dimensional space, we only report the result for the UU data set. Finally, we use the SUM as the default aggregate type, since it is the fundamental query type (COUNT is a special case for SUM) for all aggregates we have discussed. The MIN/MAX aggregate has similar

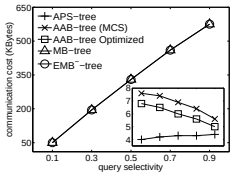


(a) I/O.

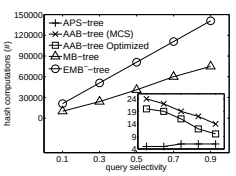


(b) $\mathcal{MCS}$ size.

Fig. 7. 1d queries: Server-side cost.



(a) Comm. cost.



(b) Verification cost.

Fig. 8. 1d queries: Comm. and verification cost.

storage, update, query, communication and authentication costs. The storage and update costs of the MEDIAN and QUANTILE aggregate types for our technique are about two times of the cost for the COUNT aggregate in the general case. Its query, communication and authentication costs are about $3-4$ times of the corresponding COUNT aggregate with the same query range. For details, please refer to our relevant cost analysis in Section 6.

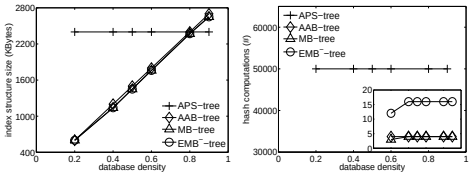
Lastly, in some figures, the performance of competing methods is significantly different. To clearly illustrate the trend for every method, we use a smaller figure inside a larger figure to have different scale on the y-axes. However, in all these cases, both the smaller and larger figures have the same scale on their x-axes.

8.2 One-dimensional Queries

First, we evaluate one-dimensional queries. Candidates are the APS-tree, the AAB-tree, and the structures for selection queries, like the MB-tree, and EMB$^-$-tree [Li et al. 2006b]. For the naive approaches to authenticate a query, first we answer the range query and report all values to the client, who authenticates them and computes the aggregate. 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 $M = 100,000$, $N = \delta M$. We vary the density of the database $\delta \in [0.1, 0.9]$ and the query selectivity $\rho \in [0.1, 0.9]$.

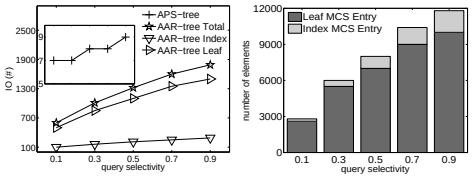
Figure 7(a) shows the I/O cost at the server side. The best structure overall is the APS-tree, since it only needs to verify two PS entries, with a cost proportional to the height of the tree. The naive approaches are one to two orders of magnitude more expensive than the other techniques, since they 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



(a) Storage cost (b) Update cost

Fig. 9. 1d queries: Storage and update cost.



(a) I/O (b) $\mathcal{MCS}$ size

Fig. 10. 3d queries: Server-side cost.

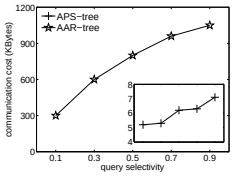
communication cost increases linearly with the query selectivity, which is simply explained by the fact that they have to return ρN number of aggregate values and keys. The verification cost is several orders of magnitude slower in terms of hash computations, due to the fact that all the results in the query range need to be authenticated, which is an overhead when queries are not very selective.

The query efficiency of the APS-tree comes 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, since 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 cost of one update. 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 just one update in the one-dimensional space. 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 compared to the naive approaches, and is orders of magnitude better than the APS-tree in terms of update cost.

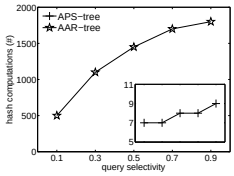
8.3 Multi-dimensional Queries

We next compare the APS-tree and AAR-tree for 3-dimensional queries. The naive approach based on authenticating range selection queries becomes very 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 unique elements of $\prod_{i=1}^{3} M_i = 125,000$ tuples, query workload with 100 queries with variable ρ values on a 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 overall, since it only needs to authenticate eight PS elements. The AAR-tree has much higher I/O since it needs to construct the $\mathcal{MCS}$ 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 $\mathcal{MCS}$ as indicated in Figure 10(b). This becomes clear in Figure 10(a) which

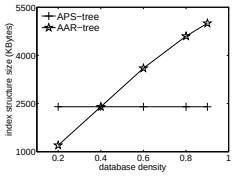


(a) Comm. cost

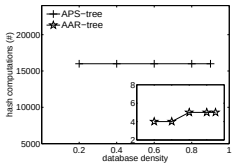


(b) Verification cost

Fig. 11. 3d queries: Comm. and verification cost.



(a) Storage cost



(b) Update cost

Fig. 12. 3d queries: Storage and update cost.

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 $\mathcal{MCS}$ sizes and because it has to return the MBRs of all $\mathcal{MCS}$ entries. The verification costs follow similar trends, since the number of hash computations is directly related to the number of entries in the $\mathcal{MCS}$.

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. 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) plots the update cost as a function of δ. We generate 100 updates uniformly at random and report the average update cost of one update. The superior query cost of the APS-tree is offset by its extremely high update cost. Regardless of database density, the APS-tree has to update 1/8-th of its data entries on average for one update in the three dimensional space. In contrast, the AAR-tree inherits the good update characteristics of the R-tree.

8.4 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 and supports encrypted data, but it has 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.

9. 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 high 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. Our approach could handle literally all

aggregate types, such as SUM, COUNT, AVG, MIN, MAX, MEDIAN and QUANTILE. We have also shown how to authenticate aggregation queries efficiently when the database is encrypted to protect data confidentiality.

REFERENCES

Agrawal, R., Kiernan, J., Srikant, R., and Xu, Y. 2004. Order preserving encryption for numeric data. In *SIGMOD*. 563–574.

Agrawal, R., Srikant, R., and Thomas, D. 2005. Privacy preserving OLAP. In *SIGMOD*. 251–262.

Anagnostopoulos, A., Goodrich, M., and Tamassia, R. 2001. Persistent authenticated dictionaries and their applications. In *ISC*. 379–393.

Atallah, M. J., Cho, Y., and Kundu, A. 2008. Efficient data authentication in an environment of untrusted third-party distributors. In *ICDE*. 696–704.

Bertino, E., Carminati, B., Ferrari, E., Thuraisingham, B., and Gupta, A. 2004. Selective and authentic third-party distribution of XML documents. *TKDE 16,* 10, 1263–1278.

Cheng, W., Pang, H., and Tan, K. 2006. Authenticating multi-dimensional query results in data publishing. In *DBSec*. 60–73.

Dai, W. Crypto++ Library. http://www.eskimo.com/∼weidai/cryptlib.html.

Devanbu, P., Gertz, M., Martel, C., and Stubblebine, S. G. 2000. Authentic third-party data publication. In *IFIP Workshop on Database Security (DBSec)*. 101–112.

Ge, T. and Zdonik, S. B. 2007. Answering aggregation queries in a secure system model. In *VLDB*. 519–530.

Goodrich, M. T., Tamassia, R., and Triandopoulos, N. 2008. Super-efficient verification of dynamic outsourced databases. In *CT-RSA*. 407–424.

Goodrich, M. T., Tamassia, R., and Triandopoulos, N. 2010. Efficient authenticated data structures for graph connectivity and geometric search problems. *Algorithmica To appear*.

Goodrich, M. T., Tamassia, R., Triandopoulos, N., and Cohen, R. 2003. Authenticated data structures for graph and geometric searching. In *CT-RSA*. 295–313.

Hacigümüs, H., Iyer, B. R., Li, C., and Mehrotra, S. 2002. Executing SQL over encrypted data in the database service provider model. In *SIGMOD*. 216–227.

Hacigümüs, H., Iyer, B. R., and Mehrotra, S. 2004. Efficient execution of aggregation queries over encrypted relational databases. In *DASFAA*. 125–136.

Ho, C.-T., Agrawal, R., Megiddo, N., and Srikant, R. 1997. Range queries in OLAP data cubes. In *SIGMOD*. 73–88.

Jürgens, M. and Lenz, H. 1999. PISA: Performance models for index structures with and without aggregated data. In *SSDBM*. 78–87.

Kamel, I. and Faloutsos, C. 1993. On packing R-Trees. In *CIKM*. 490–499.

Kundu, A. and Bertino, E. 2008. Structural signatures for tree data structures. *Proc. VLDB Endow. 1,* 1, 138–150.

Lazaridis, I. and Mehrotra, S. 2001. Progressive approximate aggregate queries with a multi-resolution tree structure. In *SIGMOD*. 401–412.

Levitte, R., Henson, S., and et al. OpenSSL. http://www.openssl.org.

Li, F., Hadjieleftheriou, M., Kollios, G., and Reyzin, L. 2006a. Authenticated index sturctures for aggregation queries in outsourced databases. Tech. rep., Computer Science Department, Boston University. 2006-011, http://www.cs.bu.edu/techreports.

Li, F., Hadjieleftheriou, M., Kollios, G., and Reyzin, L. 2006b. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*. 121–132.

Li, F., Yi, K., Hadjieleftheriou, M., and Kollios, G. 2007. Proof-infused streams: Enabling authentication of sliding window queries on streams. In *VLDB*. 147–158.

Martel, C., Nuckolls, G., Devanbu, P., Gertz, M., Kwong, A., and Stubblebine, S. 2004. A general model for authenticated data structures. *Algorithmica 39,* 1, 21–41.

Merkle, R. 1980. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy*. 122–134.

Miklau, G. 2005. Confidentiality and integrity in data exchange. Ph.D. thesis, University of Washington.

Mouratidis, K., Sacharidis, D., and Pang, H. 2009. Partially materialized digest scheme: an efficient verification method for outsourced databases. *The VLDB Journal 18,* 1, 363–381.

Mykletun, E., Narasimha, M., and Tsudik, G. 2004a. Authentication and integrity in outsourced databases. In *NDSS*.

Mykletun, E., Narasimha, M., and Tsudik, G. 2004b. Signature bouquets: Immutability for aggregated/condensed signatures. In *ESORICS*. 160–176.

Mykletun, E. and Tsudik, G. 2006. Aggregation queries in the database-as-a-service model. In *DBSec*. 89–103.

Naor, M. and Nissim, K. 1998. Certificate revocation and certificate update. In *Proceedings of the USENIX Security Symposium*.

Narasimha, M. and Tsudik, G. 2005. DSAC: Integrity of outsourced databases with signature aggregation and chaining. In *CIKM*. 235–236.

Pang, H., Jain, A., Ramamritham, K., and Tan, K.-L. 2005. Verifying completeness of relational query results in data publishing. In *SIGMOD*. 407–418.

Pang, H. and Mouratidis, K. 2008. Authenticating the query results of text search engines. *Proc. VLDB Endow. 1,* 1, 126–137.

Pang, H., Zhang, J., and Mouratidis, K. 2009. Scalable verification for outsourced dynamic databases. *Proc. VLDB Endow. 2,* 1, 802–813.

Papadopoulos, S., Papadias, D., Cheng, W., and Tan, K.-L. 2009. Separating authentication from query execution in outsourced databases. In *ICDE*. 1148–1151.

Papadopoulos, S., Yang, Y., and Papadias, D. 2007. CADS: Continuous authentication on data streams. In *VLDB*. 135–146.

Papamanthou, C., Tamassia, R., and Triandopoulos, N. 2008. Authenticated hash tables. In *CCS*. 437–448.

Singh, S. and Prabhakar, S. 2008. Ensuring correctness over untrusted private database. In *EDBT*. 476–486.

Sion, R. 2005. Query execution assurance for outsourced databases. In *VLDB*. 601–612.

Tamassia, R. and Triandopoulos, N. 2005. Computational bounds on hierarchical data processing with applications to information security. In *ICALP*. 153–165.

Tamassia, R. and Triandopoulos, N. 2007. Efficient content authentication in peer-to-peer networks. In *ACNS*. 354–372.

Tao, Y. and Papadias, D. 2004. Range aggregate processing in spatial databases. *TKDE 16,* 12, 1555–1570.

Theodoridis, Y. and Sellis, T. K. 1996. A model for the prediction of R-tree performance. In *PODS*. 161–171.

Xie, M., Wang, H., Yin, J., and Meng, X. 2007. Integrity auditing of outsourced data. In *VLDB*. 782–793.

Xie, M., Wang, H., Yin, J., and Meng, X. 2008. Providing freshness guarantees for outsourced databases. In *EDBT*. 323–332.

Yang, Y., Papadias, D., Papadopoulos, S., and Kalnis, P. 2009. Authenticated join processing in outsourced databases. In *SIGMOD*. 5–18.

Yang, Y., Papadopoulos, S., Papadias, D., and Kollios, G. 2008. Spatial outsoucing for location-based services. In *ICDE*. 1082–1091.

Yang, Y., Papadopoulos, S., Papadias, D., and Kollios, G. 2009. Authenticated indexing for outsourced spatial databases. *The VLDB Journal 18,* 3, 631–648.

Received September 2008; revised August 2009; accepted February 2010