Dynamic Authenticated Index Structures for Outsourced Databases

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

ABSTRACT

In outsourced database (ODB) systems the database owner publishes its data through a number of remote servers, with the goal of enabling clients at the edge of the network to access and query the data more efficiently. As servers might be untrusted or can be compromised, *query authentication* becomes an essential component of ODB systems. Existing solutions for this problem concentrate mostly on static scenarios and are based on idealistic properties for certain cryptographic primitives. In this work, first we define a variety of essential and practical cost metrics associated with ODB systems. Then, we analytically evaluate a number of different approaches, in search for a solution that best leverages all metrics. Most importantly, we look at solutions that can handle *dynamic scenarios*, where owners periodically update the data residing at the servers. Finally, we discuss *query freshness*, a new dimension in data authentication that has not been explored before. A comprehensive experimental evaluation of the proposed and existing approaches is used to validate the analytical models and verify our claims. Our findings exhibit that the proposed solutions improve performance substantially over existing approaches, both for static and dynamic environments.

1. INTRODUCTION

Database outsourcing [13] is a new paradigm that has been proposed recently and received considerable attention. The basic idea is that data owners delegate their database needs and functionalities to a third-party that provides services to the users of the database. Since the third party can be untrusted or can be compromised, security concerns must be addressed before this delegation.

There are three main entities in the Outsourced Database (ODB) model: the data owner, the database service provider (a.k.a. server) and the client. In general, many instances of each entity may exist. In practice, usually there is a single or a few data owners, a few servers, and many clients. The data owner first creates the database, along with the associated index and authentication structures and uploads it to the servers. It is assumed that the data owner may update the database periodically or occasionally, and that the data management and retrieval happens only at the servers. Clients submit queries about the owner's data to the servers and get back results through the network.

It is much cheaper to maintain ordinary servers than to maintain truly secure ones, particularly in the distributed setting. To guard against malicious/compromised servers, the owner must give the clients the ability to authenticate the answers they receive without having to trust the servers. In that respect, query authentication has three important dimensions: *correctness*, *completeness* and *freshness*. Correctness means that the client must be able to validate that the returned records do exist in the owner's database and have not been modified in any way. Completeness means that no answers have been omitted from the result. Finally, freshness means that the results are based on the most current version of the database, that incorporates the latest owner updates. It should be stressed here that query freshness is an important dimension of query authentication that has not been extensively explored in the past, since it is a requirement arising from updates to the ODB systems, an aspect that has not been sufficiently studied yet.

There are a number of important costs relating to the construction, query, and update phases of the aforementioned model. In particular, in this work the following metrics are considered: 1. The computation overhead for the owner, 2. The owner-server communication cost, 3. The storage overhead for the server, 4. The computation overhead for the server, 5. The client-server communication cost, and 6. The computation cost for the client (for verification).

Previous work has addressed the problem of query authentication mostly for static scenarios, where owners never issue data updates. In addition, existing solutions take into account only a subset of the metrics proposed here, and hence are optimized only for particular scenarios and not the general case. Finally, previous work was mostly of theoretical nature, analyzing the performance of the proposed techniques using analytical cost formulas, and not taking into account the fact that certain cryptographic primitives do not feature idealistic characteristics in practice. For example, trying to minimize the I/O cost associated with the construction of an authenticated structure does not take into account the fact that generating signatures using popular public signature schemes is two times slower than a random disk page access on today's computers. To the best of our knowledge, no previous work ever conducted empirical evaluations on a working prototype of existing techniques.

Our contributions. In this work, we: 1. Conduct a methodical analysis of existing approaches over all six metrics,

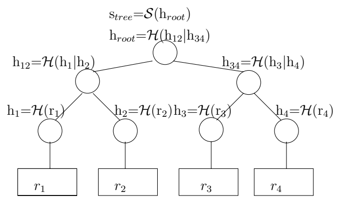

$s_{tree}=\mathcal{S}(h_{root})$
$h_{root}=\mathcal{H}(h_{12}|h_{34})$
$h_{12}=\mathcal{H}(h_1|h_2)$
$h_{34}=\mathcal{H}(h_3|h_4)$
$h_1=\mathcal{H}(r_1)$
$h_2=\mathcal{H}(r_2)$
$h_3=\mathcal{H}(r_3)$
$h_4=\mathcal{H}(r_4)$
r_1
r_2
r_3
r_4

Figure 1: Example of a Merkle hash tree.

2. Propose a novel authenticated structure that best leverages all metrics, 3. Formulate detailed cost models for all techniques that take into account not only the usual structural maintenance overheads, but the cost of cryptographic operations as well, 4. Discuss the extensions of the proposed techniques for dynamic environments (where data is frequently updated), 5. Consider possible solutions for guaranteeing query freshness, 6. Implement a fully working prototype and perform a comprehensive experimental evaluation and comparison of all alternatives.

We would like to point out that there are other security issues in ODB systems that are orthogonal to the problems considered here. Examples include: privacy-preservation issues [14, 1, 10], secure query execution [12], security in conjunction with access control requirements [20, 29, 5] and query execution assurance [30]. In particular, query execution assurance of [30] does not provide authentication: the server could pass the challenges and yet still return false query results.

The rest of the paper is organized as follows. Section 2 presents background on essential cryptography tools, and a brief review of related work. Section 3 discusses the authenticated index structures for static ODB scenarios. Section 4 extends the discussion to the dynamic case and Section 5 addresses query freshness. Finally, the empirical evaluation is presented in Section 6. Section 7 concludes the paper.

2. PRELIMINARIES

The basic idea of the existing solutions to the query authentication problem is the following. The owner creates a specialized data structure over the original database that is stored at the servers together with the database. The structure is used by a server to provide a verification object $\mathcal{VO}$ along with the answers, which the client can use for authenticating the results. Verification usually occurs by means of using collision-resistant hash functions and digital signature schemes. Note that in any solution, some information that is authentic to the owner must be made available to the client; else, from the client's point of view, the owner cannot be differentiated from a (potentially malicious) server. Examples of such information include the owner's public signature verification key or a token that in some way authenticates the database. Any successful scheme must make it computationally infeasible for a malicious server to find incorrect query results and verification object that will be accepted by a client who has the appropriate authentication information from the owner.

2.1 Cryptography essentials

Collision-resistant hash functions. For our purposes, a hash function $\mathcal{H}$ is an efficiently computable function that takes a variable-length input x to a fixed-length output $y = \mathcal{H}(x)$. *Collision resistance* states that it is computationally infeasible to find two inputs, $x_1 \neq x_2$, such that $\mathcal{H}(x_1) = \mathcal{H}(x_2)$. Collision-resistant hash functions can be built provably based on various cryptographic assumptions, such as hardness of discrete logarithms [17]. However, in this work we concentrate on using heuristic hash functions, which have the advantage of being very fast to evaluate, and specifically focus on SHA1 [24], which takes variable-length inputs to 160-bit (20-byte) outputs. SHA1 is currently considered collision-resistant in practice; we also note that any eventual replacement to SHA1 developed by the cryptographic community can be used instead of SHA1 in our solution.

Public-key digital signature schemes. A public-key digital signature scheme, formally defined in [11], is a tool for authenticating the integrity and ownership of the signed message. In such a scheme, the signer generates a pair of keys (SK, PK), keeps the secret key SK secret, and publishes the public key PK associated with her identity. Subsequently, for any message m that she sends, a signature s_m is produced by: $s_m = \mathcal{S}(SK, m)$. The recipient of s_m and m can verify s_m via $\mathcal{V}(PK, m, s_m)$ that outputs "valid" or "invalid." A valid signature on a message assures the recipient that the owner of the secret key intended to authenticate the message, and that the message has not been changed. The most commonly used public digital signature scheme is RSA [28]. Existing solutions [26, 27, 21, 23] for the query authentication problem chose to use this scheme, hence we adopt the common 1024-bit (128-byte) RSA. Its signing and verification cost is one hash computation and one modular exponentiation with 1024-bit modulus and exponent.

Aggregating several signatures. In the case when t signatures $s_1, \ldots, s_t$ on t messages $m_1, \ldots, m_t$ signed by the same signer need to be verified all at once, certain signature schemes allow for more efficient communication and verification than t individual signatures. Namely, for RSA it is possible to combine the t signatures into a single aggregated signature $s_{1,t}$ that has the same size as an individual signature and that can be verified (almost) as fast as an individual signature. This technique is called Condensed-RSA [22]. The combining operation can be done by anyone, as it does not require knowledge of SK; moreover, the security of the combined signature is the same as the security of individual signatures. In particular, aggregation of t RSA signatures can be done at the cost of $t-1$ modular multiplications, and verification can be performed at the cost of $t-1$ multiplications, t hashing operations, and one modular exponentiation (thus, the computational gain is that $t-1$ modular exponentiations are replaced by modular multiplications). Note that aggregating signatures is possible only for some digital signature schemes.

The Merkle Hash Tree. An improvement on the straightforward solution for authenticating a set of data values is the Merkle hash tree (see Figure 1), first proposed by [18]. It solves the simplest form of the query authentication problem for point queries and datasets that can fit in main memory. The Merkle hash tree 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. Verification of data values is based on the fact that the hash value of the root of the tree is authentically published (authenticity can be established by a digital signature). To prove the authenticity of any data value, all the prover has to do is to provide the verifier, in addition to the data value itself, with the values stored in the siblings of the path that leads from the root of the tree to that value. The veri-

Table 1: Notation used.

Symbol	Description
r	A database record
k	A B^+-tree key
p	A B^+-tree pointer
h	A hash value
s	A signature
$\lvert x \rvert$	Size of object x
N_D	Total number of database records
N_R	Total number of query results
P	Page size
f_x	Node fanout of structure x
d_x	Height of structure x
$\mathcal{H}_l(x)$	A hash operation on input x of length l
$\mathcal{S}_l(x)$	A signing operation on input x of length l
$\mathcal{V}_l(x)$	A verifying operation on input x of length l
$\mathcal{C}_x$	Cost of operation x
$\mathcal{VO}$	The verification object

fier, by iteratively computing all the appropriate hashes up the tree, at the end can simply check if the hash she has computed for the root matches the authentically published value. The security of the Merkle hash tree is based on the collision-resistance of the hash function used: it is computationally infeasible for a malicious prover to fake a data value, since this would require finding a hash collision somewhere in the tree (because the root remains the same and the leaf is different—hence, there must be a collision somewhere in between). Thus, the authenticity of any one of n data values can be proven at the cost of providing and computing $\log_2 n$ hash values, which is generally much cheaper than storing and verifying one digital signature per data value. Furthermore, the relative position (leaf number) of any of the data values within the tree is authenticated along with the value itself.

Cost models for SHA1, RSA and Condensed-RSA. Since all existing authenticated structures are based on SHA1 and RSA, it is imperative to evaluate the relative cost of these operations in order to be able to draw conclusions about which is the best alternative in practice. Based on experiments with two widely used cryptography libraries, Crypto++ [7] and OpenSSL [25], we obtained results for hashing, signing, verifying and performing modulo multiplications. Evidently, one hashing operation on our testbed computer takes approximately 2 to 3 μs. Modular multiplication, signing and verifying are, respectively, approximately 100, 10,000 and 1,000 times slower than hashing (verification is faster than signing due to the fact that the public verification exponent can be fixed to a small value).

Thus, it is clear that multiplication, signing and verification operations are very expensive, and comparable to random disk page accesses. The cost of these operations needs to be taken into account in practice, for the proper design of authenticated structures. In addition, since the cost of hashing is orders of magnitude smaller than that of singing, it is essential to design structures that use as few signing operations as possible, and hashing instead.

2.2 Previous work

There are several notable works that are related to our problem. A good survey is provided in [23]; our review here is brief. The first set of attempts to address query authentication problems in ODB systems appeared in [9, 8, 16]. The focus of these works is on designing solutions for query correctness only, creating structures that are based on Merkle trees. The work of [16] generalized the Merkle hash tree ideas to work with any DAG (directed acyclic graph) structure. With similar techniques, the work of [3] uses the Merkle tree to authenticate XML documents in the ODB model. The work of [27] further extended the idea and introduced the *VB-tree* which was suitable for structures stored on secondary storage. However, this approach is expensive and was later subsumed by [26]. Several proposals for signature-based approaches addressing both query correctness and completeness appear in [23, 21, 26]. We are not aware of work that specifically addresses the query freshness issue.

Hardware support for secure data accesses is investigated in [5, 4]. It offers a promising research direction for designing query authentication schemes with special hardware support. Lastly, distributed content authentication has been addressed in [31], where a distributed version of the Merkle hash tree is applied.

3. THE STATIC CASE

In this section we illustrate three approaches for query correctness and completeness: a signature-based approach similar to the ones described in [26, 23], a Merkle-tree-like approach based on the ideas presented in [16], and our novel embedded tree approach. We present them for the *static scenario* where no data updates occur between the owner and the servers on the outsourced database. We also present *analytical cost models* for all techniques, given a variety of performance metrics. As already mentioned, detailed analytical modeling was not considered in related literature and is an important contribution of this work. It gives the ability to the owners to decide which structure best satisfies their needs, using a variety of performance measures.

In the following, we derive models for the *storage*, *construction*, *query*, and *authentication* cost of each technique, taking into account the overhead of hashing, signing, verifying data, and performing expensive computations (like modular multiplications of large numbers). The analysis considers range queries on a specific database attribute A indexed by a B^+-tree [6]. The size of the structure is important first for quantifying the storage overhead on the servers, and second for possibly quantifying the owner/server communication cost. The construction cost is useful for quantifying the overhead incurred by the database owner for outsourcing the data. The query cost quantifies the incurred server cost for answering client queries, and hence the potential query throughput. The authentication cost quantifies the server/client communication cost and, in addition, the client side incurred cost for verifying the query results. The notation used is summarized in Table 1. In the rest, for ease of exposition, it is assumed that all structures are bulk-loaded in a bottom-up fashion and that all index nodes are completely full. In addition, all divisions are assumed to have residual zero. Our results can easily be extended to the general case.

3.1 Aggregated Signatures with B$^+$-trees

The first authenticated data structure for static environments is a direct extension of aggregated signatures and ideas that appeared in [23, 26]. To guarantee correctness and completeness the following technique can be used: First, the owner individually hashes and signs all consecutive pairs of tuples in the database, assuming some sorted order on a

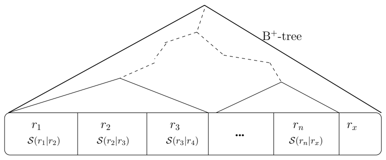


Figure 2: The signature-based approach.

given attribute A. For example, given two consecutive tuples r_i, r_j the owner transmits to the servers the pair (r_i, s_i), where $s_i = \mathcal{S}(r_i|r_j)$ ('|' denotes some canonical pairing of strings that can be uniquely parsed back into its two components; e.g., simple string concatenation if the lengths are fixed). The first and last tuples can be paired with special marker records. Chaining tuples in this way will enable the clients to verify that no in-between tuples have been dropped from the results or modified in any way. An example of this scheme is shown in Figure 2.

In order to speed up query execution on the server side a B^+-tree is constructed on top of attribute A. To answer a query the server constructs a $\mathcal{VO}$ that contains one pair $r_q|s_q$ per query result. In addition, one tuple to the left of the lower-bound of the query results and one to the right of the upper-bound is returned, in order for the client to be able to guarantee that no boundary results have been dropped. Notice that since our completeness requirements are less stringent than those of [26] (where they assume that database access permissions restrict which tuples the database can expose to the user), for fairness we have simplified the query algorithm substantially here.

There are two obvious and serious drawbacks associated with this approach. First, the extremely large $\mathcal{VO}$ size that contains a linear number of signatures w.r.t. N_R, taking into account that signatures sizes are very large. Second, the high verification cost for the clients. Authentication requires N_R verification operations which, as mentioned earlier, are very expensive. To solve this problem one can use the aggregated signature scheme discussed in Section 2.1. Instead of sending one signature per query result the server can send one combined signature s^π for all results, and the client can use an aggregate verification instead of individual verifications.

By using aggregated RSA signatures, the client can authenticate the results by hashing consecutive pairs of tuples in the result-set, and calculating the product $m^\pi = \prod_{\forall q} h_q \pmod{n}$ (where n is the RSA modulus from the public key of the owner). It is important to notice that both s^π and m^π require a linear number of modular multiplications (w.r.t. N_R). The cost models of the aggregated signature scheme for the metrics considered are as follows:

Node fanout: The node fanout of the B^+-tree structure is:

$$f_a = \frac{P - |p|}{|k| + |p|} + 1\,. \tag{1}$$

Storage cost: The total size of the authenticated structure (excluding the database itself) is equal to the size of the B^+-tree plus the size of the signatures. For a total of N_D tuples the height of the tree is equal to $d_a = \log_{f_a} N_D$, consisting of $N_I = \frac{f_a^{d_a}-1}{f_a-1}$ $(= \sum_{i=0}^{d_a-1} f_a^i)$ nodes in total. Hence:

$$\mathcal{C}_s^a = P \cdot \frac{f_a^{d_a} - 1}{f_a - 1} + N_D \cdot |s|. \tag{2}$$

The storage cost also reflects the initial communication cost between the owner and servers. Notice that the owner does not have to upload the B$^+$-tree to the servers, since the latter can rebuild it by themselves, which will reduce the owner/server communication cost but increase the computation cost at the servers. Nevertheless, the cost of sending the signatures cannot be avoided.

Construction cost: The cost incurred by the owner for constructing the structure has three components: the signature computation cost, bulk-loading the B^+-tree, and the I/O cost for storing the structure. Since the signing operation is very expensive, it dominates the overall cost. Bulk-loading the B^+-tree in main memory is much less expensive and its cost can be omitted. Hence:

$$\mathcal{C}_c^a = N_D \cdot (\mathcal{C}_{\mathcal{H}_{|r|}} + \mathcal{C}_{\mathcal{S}_{2|h|}}) + \frac{\mathcal{C}_s^a}{P} \cdot \mathcal{C}_{IO}. \tag{3}$$

$\mathcal{VO}$ construction cost: The cost of constructing the $\mathcal{VO}$ for a range query depends on the total disk I/O for traversing the B^+-tree and retrieving all necessary record/signature pairs, as well as on the computation cost of s^π. Assuming that the total number of leaf pages accessed is $N_Q = \frac{N_R}{f_a}$, the $\mathcal{VO}$ construction cost is:

$$\mathcal{C}_q^a = (N_Q + d_a - 1 + \frac{N_R \cdot |r|}{P} + \frac{N_R \cdot |s|}{P}) \cdot \mathcal{C}_{IO} + \mathcal{C}_{s^\pi}, \tag{4}$$

where the last term is the modular multiplication cost for computing the aggregated signature, which is linear to N_R. The I/O overhead for retrieving the signatures is also large.

Authentication cost: The size of the $\mathcal{VO}$ is equal to the result-set size plus the size of one signature:

$$|\mathcal{VO}|^a = N_R \cdot |r| + |s|. \tag{5}$$

The cost of verifying the query results is dominated by the hash function computations and modular multiplications at the client:

$$\mathcal{C}_v^a = N_R \cdot \mathcal{C}_{\mathcal{H}_{|r|}} + \mathcal{C}_{m^\pi} + \mathcal{C}_{\mathcal{V}_{|n|}}, \tag{6}$$

where the modular multiplication cost for computing the aggregated hash value is linear to the result-set size N_R, and the size of the final product has length in the order of $|n|$ (the RSA modulus). The final term is the cost of verifying the product using s^π and the owner's public key.

It becomes obvious now that one advantage of the aggregated signature scheme is that it features small $\mathcal{VO}$ sizes and hence small client/server communication cost. On the other hand it has the following serious drawbacks: 1. Large storage overhead on the servers, dominated by the large signature sizes, 2. Large communication overhead between the owners and the servers that cannot be reduced, 3. A very high initial construction cost, dominated by the cost of computing the signatures, 4. Added I/O cost for retrieving signatures, linear to N_R, 5. An added modular multiplication cost, linear to the result-set size, for constructing the $\mathcal{VO}$ and authenticating the results. Our experimental evaluation shows that this cost is significant compared to other operations, 6. The requirement for a public key signature scheme that supports aggregated signatures. For the rest of the paper, this approach is denoted as Aggregated Signatures with B^+-trees (ASB-tree).

3.2 The Merkle B-tree

Motivated by the drawbacks of the ASB-tree, we present a different approach for building authenticated structures

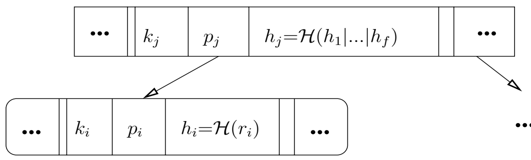
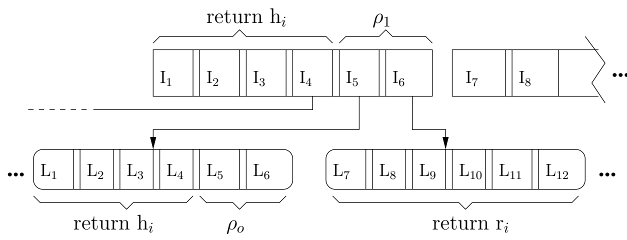


Figure 3: An MB-tree node.



return h$_i$
ρ_1
I$_1$ I$_2$ I$_3$ I$_4$ I$_5$ I$_6$ I$_7$ I$_8$
L$_1$ L$_2$ L$_3$ L$_4$ L$_5$ L$_6$ L$_7$ L$_8$ L$_9$ L$_{10}$ L$_{11}$ L$_{12}$
return h$_i$
ρ_o
return r$_i$

Figure 4: A query traversal on an MB-tree. At every level the hashes of the residual entries on the left and right boundary nodes need to be returned.

that is based on the general ideas of [16] (which utilize the Merkle hash tree) applied in our case on a B^+-tree structure. We term this structure the Merkle B-tree (MB-tree).

As already explained in Section 2.1, the Merkle hash tree uses a hierarchical hashing scheme in the form of a binary tree to achieve query authentication. Clearly, one can use a similar hashing scheme with trees of *higher fanout and with different organization algorithms*, like the B^+-tree, to achieve the same goal. An MB-tree works like a B^+-tree and also consists of ordinary B^+-tree nodes that are extended with one hash value associated with every pointer entry. The hash values associated with entries on leaf nodes are computed on the database records themselves. The hash values associated with index node entries are computed on the concatenation of the hash values of their children. For example, an MB-tree is illustrated in Figure 3. A leaf node entry is associated with a hash value $h = \mathcal{H}(r_i)$, while an index node entry with $h = \mathcal{H}(h_1|\cdots|h_{f_m})$, where $h_1, \ldots, h_{f_m}$ are the hash values of the node's children, assuming fanout f_m per node. After computing all hash values, the owner has to sign the hash of the root using the private key.

To answer a range query the server builds a $\mathcal{VO}$ by initiating two top-down B^+-tree like traversals, one to find the left-most and one the right-most query result. At the leaf level, the data contained in the nodes between the two discovered boundary leaves are returned, as in the normal B^+-tree. The server also needs to include in the $\mathcal{VO}$ the hash values of the entries contained in each index node that is visited by the lower and upper boundary traversals of the tree, except the hashes to the right (left) of the pointers that are traversed during the lower (upper) boundary traversals. At the leaf level, the server inserts only the answers to the query, along with the hash values of the residual entries to the left and to the right parts of the boundary leaves. The result is also increased with one tuple to the left and one to the right of the lower-bound and upper-bound of the query result respectively, for completeness verification. Finally, the signed root of the tree is inserted as well. An example query traversal is shown in Figure 4.

The client can iteratively compute all the hashes of the sub-tree corresponding to the query result, all the way up to the root using the $\mathcal{VO}$. The hashes of the query results are computed first and grouped into their corresponding leaf nodes[1], and the process continues iteratively, until all the hashes of the query sub-tree have been computed. After the hash value of the root has been computed, the client can verify the correctness of the computation using the owner's public key and the signed hash of the root. It is easy to see that since the client is forced to recompute the whole query sub-tree, both correctness and completeness is guaranteed. It is interesting to note here that one could avoid building the whole query sub-tree during verification by individually signing all database tuples as well as each node of the B^+-tree. This approach, called VB-tree, was proposed in [27] but it is subsumed by the signature based approach. The analytical cost models of the MB-tree are as follows:

Node fanout: The node fanout in this case is:

$$f_m = \frac{P - |p| - |h|}{|k| + |p| + |h|} + 1. \tag{7}$$

Notice that the maximum node fanout of the MB-trees is considerably smaller than that of the ASB-tree, since the nodes here are extended with one hash value per entry. This adversely affects the total height of the MB-tree.

Storage cost: The total size is equal to:

$$\mathcal{C}_s^m = P \cdot \frac{f_m^{d_m} - 1}{f_m - 1} + |s|. \tag{8}$$

An important advantage of the MB-tree is that the storage cost does not necessarily reflect the owner/server communication cost. The owner, after computing the final signature of the root, does not have to transmit all hash values to the server, but only the database tuples. The server can recompute the hash values incrementally by recreating the MB-tree. Since hash computations are cheap, for a small increase in the server's computation cost this technique will reduce the owner/sever communication cost drastically.

Construction cost: The construction cost for building an MB-tree depends on the hash function computations and the total I/Os. Since the tree is bulk-loaded, building the leaf level requires N_D hash computations of length input $|r|$. In addition, for every tree node one hash of input length $f_m \cdot |h|$ is computed. Since there are a total of $N_I = \frac{f_m^{d_m}-1}{f_m-1}$ nodes on average (given height $d_m = \log_{f_m} N_D$), the total number of hash function computations, and hence the total cost for constructing the tree is given by:

$$\mathcal{C}_c^m = N_D \cdot \mathcal{C}_{\mathcal{H}_{|r|}} + N_I \cdot \mathcal{C}_{\mathcal{H}_{f_m|h|}} + \mathcal{C}_{\mathcal{S}_{|h|}} + \frac{\mathcal{C}_s^m}{P} \cdot \mathcal{C}_{IO}. \tag{9}$$

$\mathcal{VO}$ construction cost: The $\mathcal{VO}$ construction cost is dominated by the total disk I/O. Let the total number of leaf pages accessed be equal to $N_Q = \frac{N_R}{f_m}$, $d_m = \log_{f_m} N_D$ and $d_q = \log_{f_m} N_R$ be the height of the MB-tree and the query sub-tree respectively. In the general case the index traversal cost is:

$$\mathcal{C}_q^m = [(d_m - d_q + 1) + 2(d_q - 2) + N_Q + \frac{N_R \cdot |r|}{P}] \cdot \mathcal{C}_{IO}, \tag{10}$$

taking into account the fact that the query traversal at some point splits into two paths. It is assumed here that the query range spans at least two leaf nodes. The first term corresponds to the hashes inserted for the common path of the two traversal from the root of the tree to the root of

[1]Extra node boundary information can be inserted in the $\mathcal{VO}$ for this purpose with a very small overhead.

the query sub-tree. The second term corresponds to the two traversals after they split. The last two terms correspond to the leaf level of the tree and the database records.

Authentication cost: Assuming that ρ_0 is the total number of query results contained in the left boundary leaf node of the query sub-tree, σ_0 on the right boundary leaf node, and ρ_i, σ_i the total number of entries of the left and right boundary nodes on level $i, 1 \leq i \leq d_q$, that point towards leaves that contain query results (see Figure 4), the size of the $\mathcal{VO}$ is:

$$\begin{aligned}|\mathcal{VO}|^m = \\ &(2f_m - \rho_0 - \sigma_0)|h| + N_R \cdot |r| + |s| + \\ &(d_m - d_q) \cdot (f_m - 1)|h| + \\ &\sum_{i=1}^{d_q-2} (2f_m - \rho_i - \sigma_i)|h| + \\ &(f_m - \rho_{d_q-1} - \sigma_{d_q-1})|h|. \qquad (11)\end{aligned}$$

This cost does not include the extra boundary information needed by the client in order to group hashes correctly, but this overhead is very small (one byte per node in the $\mathcal{VO}$) especially when compared with the hash value size. Consequently, the verification cost on the client is:

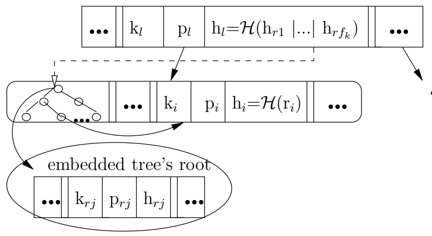
$$\mathcal{C}_v^m = N_R \cdot \mathcal{C}_{\mathcal{H}_{|r|}} + \sum_{i=0}^{d_q-1} f_m^i \cdot \mathcal{C}_{\mathcal{H}_{f_m|h|}} + (d_m - d_q) \cdot \mathcal{C}_{\mathcal{H}_{f_m|h|}} + \mathcal{C}_{\mathcal{V}_{|h|}}. \qquad (12)$$

Given that the computation cost of hashing versus signing is orders of magnitude smaller, the initial construction cost of the MB-tree is expected to be orders of magnitude less expensive than that of the ASB-tree. Given that the size of hash values is much smaller than that of signatures and that the fanout of the MB-tree will be smaller than that of the ASB-tree, it is not easy to quantify the exact difference in the storage cost of these techniques, but it is expected that the structures will have comparable storage cost, with the MB-tree being smaller. The $\mathcal{VO}$ construction cost of the MB-tree will be much smaller than that of the ASB-tree, since the ASB-tree requires many I/Os for retrieving signatures, and also some expensive modular multiplications. The MB-tree will have smaller verification cost as well since: 1. Hashing operations are orders of magnitude cheaper than modular multiplications, 2. The ASB-tree requires N_R modular multiplications for verification. The only drawback of the MB-tree is the large $\mathcal{VO}$ size, which increases the client/server communication cost. Notice that the $\mathcal{VO}$ size of the MB-tree is bounded by $f_m \cdot \log_{f_m} N_D$. Since generally $f_m \gg \log_{f_m} N_D$, the $\mathcal{VO}$ size is essentially determined by f_m, resulting in large sizes.

3.3 The Embedded Merkle B-tree

In this section we propose a novel data structure, the Embedded Merkle B-tree (EMB-tree), that provides a nice, adjustable trade-off between robust initial construction and storage cost versus improved $\mathcal{VO}$ construction and verification cost. The main idea is to have different fanouts for storage and authentication and yet combine them in the same data structure.

Every EMB-tree node consists of regular B^+-tree entries, augmented with an embedded MB-tree. Let f_e be the fanout of the EMB-tree. Then each node stores up to f_e triplets $k_i|p_i|h_i$, and an embedded MB-tree with fanout $f_k < f_e$. The leaf level of this embedded tree consists of the f_e entries of the node. The hash value at the root level of this embedded tree is stored as an h_i value in the parent of the node, thus authenticating this node to its parent. Essentially, we are collapsing an MB-tree with height $d_e \cdot d_k = \log_{f_k} N_D$ into a tree with height d_e that stores smaller MB-trees of height d_k within each node. Here, $d_e = \log_{f_e} N_D$ is the height of the EMB-tree and $d_k = \log_{f_k} f_e$ is the height of each small embedded MB-tree. An example EMB-tree node is shown in Figure 5.



k$_l$
p$_l$
h$_l$=$\mathcal{H}$(h$_{r1}$ |...| h$_{rf_k}$)
k$_i$
p$_i$
h$_i$=$\mathcal{H}$(r$_i$)
embedded tree's root
k$_{rj}$
p$_{rj}$
h$_{rj}$

Figure 5: An EMB-tree node.

For ease of exposition, in the rest of this discussion it will be assumed that f_e is a power of f_k such that the embedded trees when bulk-loaded are always full. The technical details if this is not the case can be worked out easily. The exact relation between f_e and f_k will be discussed shortly. After choosing f_k and f_e, bulk-loading the EMB-tree is straightforward: Simply group the N_D tuples in groups of size f_e to form the leaves and build their embedded trees on the fly. Continue iteratively in a bottom up fashion.

When querying the structure the server follows a path from the root to the leaves of the external tree as in the normal B^+-tree. For every node visited, the algorithm scans all $f_e - 1$ triplets $k_i|p_i|h_i$ on the data level of the embedded tree to find the key that needs to be followed to the next level. When the right key is found the server also initiates a point query on the embedded tree of the node using this key. The point query will return all the needed hash values for computing the concatenated hash of the node, exactly like for the MB-tree. Essentially, these hash values would be the equivalent of the $f_e - 1$ sibling hashes that would be returned per node if the embedded tree was not used. However, since now the hashes are arranged hierarchically in an f_k-way tree, the total number of values inserted in the $\mathcal{VO}$ per node is reduced to $(f_k - 1)d_k$.

To authenticate the query results the client uses the normal MB-tree authentication algorithm to construct the hash value of the root node of each embedded tree (assuming that proper boundary information has been included in the $\mathcal{VO}$ for separating groups of hash values into different nodes) and then follows the same algorithm once more for computing the final hash value of the root of the EMB-tree.

The EMB-tree structure uses extra space for storing the index levels of the embedded trees. Hence, by construction it has increased height compared with the MB-tree due to smaller fanout f_e. A first, simple optimization for improving the fanout of the EMB-tree is to avoid storing the embedded trees altogether. Instead, each embedded tree can be instantiated by computing fewer than $f_e/(f_k - 1)$ hashes on the fly, only when a node is accessed during the querying phase. We call this the EMB^--tree. The EMB^--tree is logically the same as the EMB-tree, however its physical structure is equivalent to an MB-tree with the hash values computed differently. With this optimization the storage overhead is minimized and the height of the EMB^--tree becomes equal to the height of the equivalent MB-tree. The trade-off is an increased computation cost for constructing the $\mathcal{VO}$. However, this cost is minimal as the number of embedded trees that need to be reconstructed is bounded by the height of the EMB^--tree.

As a second optimization, one can create a slightly more complicated embedded tree to reduce the total size of the index levels and increase fanout f_e. We call this the EMB^*-tree. Essentially, instead of using a B^+-tree as the base structure for the embedded trees, one can use a multi-way search tree with fanout f_k while keeping the structure of the external EMB-tree intact. The embedded tree based on B^+-trees has a total of $N_i = \frac{f_k^{d_k}-1}{f_k-1}$ nodes while, for example, a B-tree based embedded tree (recall that a B-tree is equivalent to a balanced multi-way search tree) would contain $N_i = \frac{f_e-1}{f_k-1}$ nodes instead. A side effect of using multi-way search trees is that the cost for querying the embedded tree on average will decrease, since the search for a particular key might stop before reaching the leaf level. This will reduce the expected cost of $\mathcal{VO}$ construction substantially. The technical details associated with embedding the multi-way tree in the EMB^*-tree nodes are included in the full version of the paper [15].

Below we give the analytical cost models of the EMB-tree. The cost models for the EMB^*-tree and EMB^--tree are similar and for brevity their detailed analysis appears in the full version of the paper [15].

Node fanout: For the EMB-tree, the relationship between f_e and f_k is given by:

$$P \geq \frac{f_k^{\log_{f_k} f_e - 1} - 1}{f_k - 1}[f_k(|k| + |p| + |h|) - |k|] + [f_e(|k| + |p| + |h|) - |k|]. \quad (13)$$

First, a suitable f_k is chosen such that the requirements for authentication cost and storage overhead are met. Then, the maximum value for f_e satisfying (13) can be determined.

Storage cost: The storage cost is equal to:

$$\mathcal{C}_s^e = P \cdot \frac{f_e^{d_e} - 1}{f_e - 1} + |s|. \quad (14)$$

Construction cost: The total construction cost is the cost of constructing all the embedded trees plus the I/Os to write the tree back to disk. Given a total of $N_I = \frac{f_e^{d_e}-1}{f_e-1}$ nodes in the tree and $N_i = \frac{f_k^{d_k}-1}{f_k-1}$ nodes per embedded tree, the cost is:

$$\mathcal{C}_c^e = N_D \cdot \mathcal{C}_{\mathcal{H}_{|r|}} + N_I \cdot N_i \cdot \mathcal{C}_{\mathcal{H}_{f_k|h|}} + \mathcal{C}_{\mathcal{S}_{|h|}} + \frac{\mathcal{C}_s^e}{P} \cdot \mathcal{C}_{IO}. \quad (15)$$

It should be mentioned here that the cost for constructing the EMB^--tree is exactly the same, since in order to find the hash values for the index entries of the trees one needs to instantiate all embedded trees. The cost of the EMB^*-tree is somewhat smaller than (15), due to the smaller number of nodes in the embedded trees.

$\mathcal{VO}$ construction cost: The $\mathcal{VO}$ construction cost is dominated by the total I/O for locating and reading all the nodes containing the query results. Similarly to the MB-tree case:

$$\mathcal{C}_q^e = [(d_e - d_q + 1) + 2(d_q - 2) + N_Q + \frac{N_R \cdot |r|}{P}] \cdot \mathcal{C}_{IO}, \quad (16)$$

where d_q is the height of the query sub-tree and $N_Q = \frac{N_R}{f_e}$ is the number of leaf pages to be accessed. Since the embedded trees are loaded with each node, the querying computation cost associated with finding the needed hash values is expected to be dominated by the cost of loading the node in memory, and hence it is omitted. It should be restated here that for the EMB^*-tree the expected $\mathcal{VO}$ construction cost will be smaller, since not all embedded tree searches will reach the leaf level of the structure.

Authentication cost: The embedded trees work exactly like MB-trees for point queries. Hence, each embedded tree returns $(f_k - 1)d_k$ hashes. Similarly with the MB-tree the total size of the $\mathcal{VO}$ is:

$$|\mathcal{VO}|^e = N_R \cdot |r| + |s| + \sum_{0}^{d_q-2} 2|\mathcal{VO}|^m + |\mathcal{VO}|^m + \sum_{d_q}^{d_m-1} (f_k - 1)d_k|h|, \quad (17)$$

where $|\mathcal{VO}|^m$ is the cost of a range query on the embedded trees of the boundary nodes contained in the query sub-tree given by equation (11), with query range that covers all pointers to children that cover the query result-set.

The verification cost is:

$$\mathcal{C}_v^e = N_R \cdot \mathcal{C}_{\mathcal{H}_{|r|}} + \sum_{i=0}^{d_q-1} f_e^i \cdot \mathcal{C}_k + (d_e - d_q) \cdot \mathcal{C}_k + \mathcal{C}_{\mathcal{V}_{|h|}}, \quad (18)$$

where $\mathcal{C}_k = N_i \cdot \mathcal{C}_{\mathcal{H}_{f_k|h|}}$ is the cost for constructing the concatenated hash of each node using the embedded tree.

For $f_k = 2$ the authentication cost becomes equal to a Merkle hash tree, which has the minimal $\mathcal{VO}$ size but higher verification time. For $f_k \geq f_e$ the embedded tree consists of only one node which can actually be discarded, hence the authentication cost becomes equal to that of an MB-tree, which has larger $\mathcal{VO}$ size but smaller verification cost. Notice that, as f_k becomes smaller, f_e becomes smaller as well. This has an impact on $\mathcal{VO}$ construction cost and size, since with smaller fanout the height of the EMB-tree increases. Nevertheless, since there is only a logarithmic dependence on f_e versus a linear dependence on f_k, it is expected that with smaller f_k the authentication related operations will become faster.

3.4 General Query Types

The proposed authenticated structures can support other query types as well. Due to lack of space we briefly discuss here a possible extension of these techniques for join queries. Other query types that can be supported are projections and multi-attribute queries.

Assume that we would like to provide authenticated results for join queries such as $R \bowtie_{A_i=A_j} S$, where $A_i \in R$ and $A_j \in S$ (R and S could be relations or result-sets from other queries), and authenticated structures for both A_i in R and A_j in S exist. The server can provide the proof for the join as follows: 1. Select the relation with the smaller size, say R, 2. Construct the VO for R (if R is an entire relation then $\mathcal{VO}$ contains only the signature of the root node from the index of R), 3. Construct the $\mathcal{VO}$s for each of the following selection queries: for each record r_k in R, $q_k =$"SELECT * FROM S WHERE $r.A_j = r_k.A_i$". The client can easily verify the join results. First, it authenticates that the relation R is complete and correct. Then, using the $\mathcal{VO}$ for each query q_k, it makes sure that it is complete for every k (even when the result of q_k is empty).

After this verification, the client can construct the results for the join query and be sure that they are complete and correct.

4. THE DYNAMIC CASE

Surprisingly, previous work has not dealt with dynamic scenarios where data gets updated at regular time intervals. In this section we analyze the performance of all approaches given dynamic updates between the owner and the servers. In particular we assume that either insertions or deletions can occur to the database, for simplicity. The performance of updates in the worst case can be considered as the cost of a deletion followed by an insertion. There are two contributing factors for the update cost: computation cost such as creating new signatures and computing hashes, and I/O cost.

4.1 Aggregated Signatures with B$^+$-trees

Suppose that a single database record r_i is inserted in or deleted from the database. Assuming that in the sorted order of attribute A the left neighbor of r_i is r_{i-1} and the right neighbor is r_{i+1}, for an insertion the owner has to compute signatures $\mathcal{S}(r_{i-1}|r_i)$ and $\mathcal{S}(r_i|r_{i+1})$, and for a deletion $\mathcal{S}(r_{i-1}|r_{i+1})$. For k consecutive updates in the best case a total of $k+2$ signature computations are required for insertions and 1 for deletions if the deleted tuples are consecutive. In the worst case a total of $2k$ signature computations are needed for insertions and k for deletions, if no two tuples are consecutive. Given k updates, suppose the expected number of signatures to be computed is represented by $E\{k\}$. Then the additional I/O incurred is equal to $\frac{E\{k\}\cdot|s|}{P}$, excluding the I/Os incurred for updating the B^+-tree structure. Since the cost of signature computations is larger than even the I/O cost of random disk accesses, a large number of updates is expected to have a very expensive updating cost. The experimental evaluation verifies this claim. The total update cost for the ASB-tree is:

$$\mathcal{C}_u^a = E\{k\}\cdot\mathcal{C}_s + \frac{E\{k\}\cdot|s|}{P}\cdot\mathcal{C}_{IO}. \tag{19}$$

4.2 The Merkle B-tree

The MB-tree can support efficient updates since only hash values are stored for the records in the tree and, first, hashing is orders of magnitude faster then signing, second, for each tuple only the path from the affected leaf to the root need to be updated. Hence, the cost of updating a single record is dominated by the cost of I/Os. Assuming that no reorganization to the tree occurs the cost of an insertion is $\mathcal{C}_u^m = \mathcal{H}_{|r|} + d_m(\mathcal{H}_{f_m|h|} + \mathcal{C}_{IO}) + \mathcal{S}_{|h|}$.

In realistic scenarios though one expects that a large number of updates will occur at the same time. In other cases the owner may decide to do a delayed batch processing of updates as soon as enough changes to the database have occurred. The naive approach for handling batch updates would be to do all updates to the MB-tree one by one and update the path from the leaves to the root once per update. Nevertheless, in case that a large number of updates affect a similar set of nodes (e.g., the same leaf) a per tuple updating policy performs an unnecessary number of hash function computations on the predecessor path. In such cases, the computation cost can be reduced significantly by recomputing the hashes of all affected nodes only once, after all the updates have been performed on the tree. A similar analysis holds for the incurred I/O as well.

Clearly, the total update cost for the per tuple update approach for k insertions is $k\cdot\mathcal{C}_u^m$ which is linear to the number of affected nodes $k\cdot d_m$. The expected cost of k updates using batch processing can be computed as follows. Given k updates to the MB-tree, assuming that all tuples are updated uniformly at random and using a standard balls and bins argument, the probability that leaf node X has been affected at least once is $P(X) = 1-(1-\frac{1}{f_m^{d_m-1}})^k$ and the expected number of leaf nodes that have been affected is $f_m^{d_m-1}\cdot P(X)$. Using the same argument, the expected number of nodes at level i (where $i=1$ is the leaf level and $1\le i\le d_m$) is $f_m^{d_m-i}\cdot P_i(X)$, where $P_i(X) = [1-(1-\frac{1}{f_m^{d_m-i}})^k]$. Hence, for a batch of k updates the total expected number of nodes that will be affected is:

$$E\{X\} = \sum_{i=0}^{d_m-1} f_m^i[1-(1-\frac{1}{f_m^i})^k]. \tag{20}$$

Hence, the expected MB-tree update cost for batch updates is

$$\mathcal{C}_u^m = k\cdot\mathcal{H}_{|r|} + E\{X\}\cdot(\mathcal{H}_{f_m|h|}+\mathcal{C}_{IO}) + \mathcal{S}_{|h|}. \tag{21}$$

In order to understand better the relationship between the per-update approach and the batch-update, we can find the closed form for $E\{X\}$ as follows:

$$\begin{aligned}
& \textstyle\sum_{i=0}^{d_m-1} f_m^i(1-(\frac{f_m^i-1}{f_m^i})^k) \\
= & \textstyle\sum_{i=0}^{d_m-1} f_m^i(1-(1-\frac{1}{f_m^i})^k) \\
= & \textstyle\sum_{i=0}^{d_m-1} f_m^i[1-\sum_{x=0}^{k}\binom{k}{x}(-\frac{1}{f_m^i})^x] \\
= & \textstyle\sum_{i=0}^{d_m-1} f_m^i - \sum_{i=0}^{d_m-1}\sum_{x=0}^{k}\binom{k}{x}(-1)^x(\frac{1}{f_m^i})^{x-1} \\
= & \textstyle kd_m - \sum_{x=2}^{k}\binom{k}{x}(-1)^x\sum_{i=0}^{d_m-1}(\frac{1}{f_m^i})^{x-1} \\
= & \textstyle kd_m - \sum_{x=2}^{k}\binom{k}{x}(-1)^x\frac{1-(\frac{1}{f_m^{d_m}})^{x-1}}{1-(\frac{1}{f_m})^{x-1}}
\end{aligned}$$

The second term quantifies the cost decrease afforded by the batch update operation, when compared to the per update cost.

For non-uniform updates to the database, the batch updating technique is expected to work well in practice given that in real settings updates exhibit a certain degree of locality. In such cases one can still derive a similar cost analysis by modeling the distribution of updates.

4.3 The Embedded MB-tree

The analysis for the EMB-tree is similar to the one for MB-trees. The update cost for per tuple updates is equal to $k\cdot\mathcal{C}_u^e$, where $\mathcal{C}_u^e = \mathcal{H}_{|r|} + d_e\log_{f_k} f_e\cdot(\mathcal{H}_{f_k|h|}+\mathcal{C}_{IO}) + \mathcal{S}_{|h|}$, once again assuming that no reorganizations to the tree occur. Similarly to the MB-tree case the expected cost for batch updates is equal to:

$$\mathcal{C}_u^e = k\cdot\mathcal{H}_{|r|} + E\{X\}\cdot\log_{f_k} f_e\cdot(\mathcal{H}_{f_k|h|}+\mathcal{C}_{IO}) + \mathcal{S}_{|h|}. \tag{22}$$

4.4 Discussion

For the ASB-tree, the communication cost for updates between owner and servers is bounded by $E\{K\}|s|$, and there is no possible way to reduce this cost as only the owner can compute signatures. However, for the hash based index structures, there are a number of options that can be used for transmitting the updates to the server. The first option is for the owner to transmit only a delta table with the updated nodes of the MB-tree (or EMB-tree) plus the signed root. The second option is to transmit only the signed root and the updates themselves and let the servers redo the necessary computations on the tree. The first approach minimizes the computation cost on the servers but increases the communication cost, while the second approach has the opposite effect.

5. QUERY FRESHNESS

The dynamic scenario reveals a third dimension of the query authentication problem that has not been sufficiently explored in previous work: namely, *query result freshness*. When the owner updates the database, a malicious or compromised server may still retain an older version. Because the old version was authenticated by the owner, the client will still accept it as authentic, unless it receives some extra information to indicate that this is no longer the case. In fact, a malicious server may choose any of the previous versions, and in some scenarios even a combination of them. If the client wishes to be sure that the version is not only authentic, but is also the most recent version available, some additional work is necessary.

This issue is similar to the problem of ensuring the freshness of signed documents, which has been studied extensively, particularly in the context of certificate validation and revocation. There are many solutions which we do not review here. The simplest is to publish a list of revoked signatures; more sophisticated ones are: including the time interval of validity as part of the signed message and reissuing the signature after the interval expires, and using hash chains to confirm validity of signatures at frequent intervals [19].

The advantage of all Merkle tree based solutions presented here is that any of these approaches can be applied directly to the single signature of the root of the tree, because this signature alone authenticates the whole database. Thus, whatever solution to the signature freshness problem one uses, it needs to be used only for one signature and freshness will be assured. Each update will require re-issuing one signature only.

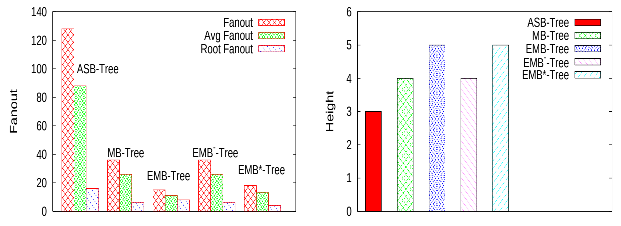
This is in contrast to the ASB-tree approach, which uses multiple signatures. It is unclear how to solve the freshness problem for the ASB-tree without applying the freshness techniques to each signature individually, which will be prohibitively expensive because of the sheer number of signatures.

6. EXPERIMENTAL EVALUATION

For our experimental evaluation we have implemented the aggregated signatures technique using a B^+-tree (ASB-tree), the MB-tree, the EMB-tree and its two variants, EMB$^-$-tree and EMB*-tree. To the best of our knowledge, this work is the first that performs simulations on a working prototype. Previous work has used only analytical techniques that did not take into account important parameters. A well designed experimental evaluation can reveal many interesting issues that are hidden when only theoretical aspects are considered. In addition, empirical evaluations help verify the correctness of the developed cost models. The prototype can be downloaded from [2].

6.1 Setup

We use a synthetic database that consists of one table with $100,000$ tuples. Each tuple contains multiple attributes, a primary key A, and is 500 bytes long. For simplicity, we assume that an authenticated index is build on A,



(a) Fanout (b) Height

Figure 6: Index parameters.

with page size equal to 1 KByte. All experiments are performed on a Linux machine with a 2.8GHz Intel Pentium 4 CPU.

The cost $\mathcal{C}_{IO}$ of one I/O operation on this machine using 1KByte pages for the indexes is on average 1 ms for a sequential read and 15 ms for random access. The costs $\mathcal{C}_{\mathcal{H}}$ of hashing a message with length 500 bytes is approximately equal to 3 μs. In comparison, the cost $\mathcal{C}_{\mathcal{S}}$ of signing a message with any length is approximately equal to 34 ms. The cost of one modular multiplication with 128 byte modulus is close to 200 μs. To quantify these costs we used the publicly available Crypto++ [7] and OpenSSL [25] libraries.

6.2 Performance Analysis

We run experiments to evaluate the proposed solutions under all metrics. First, we test the initial construction cost of each technique. Then, we measure their query and verification performance. Finally, we run simulations to analyze their performance for dynamic scenarios. For various embedded index structures, the fanout of their embedded trees is set to 2 by default, except if otherwise specified.

6.2.1 Construction Cost

First we evaluate the physical characteristics of each structure, namely the maximum and average fanout, and the height. The results are shown in Figure 6. As expected, the ASB-tree has the maximum fanout and hence the smallest height, while the opposite is true for the EMB-tree. However, the maximum height difference, which is an important measure for the number of additional I/Os per query when using deeper structures, is only 2. Of course this depends on the database size. In general, the logarithmic relation between the fanout and the database size limits the difference in height of the different indices.

Next, we measure the construction cost and the total size of each structure, which are useful indicators for the owner/server computation overhead, communication cost and storage demands. Figure 7(a) clearly shows the overhead imposed on the ASB-tree by the excessive signature computations. Notice on the enclosed detailed graph that the overhead of other authenticated index structures in the worst case is twice as large as the cost of building the B^+-tree of the ASB-tree approach. Figure 7(b) captures the total size of each structure. Undoubtedly, the ASB-tree has the biggest storage overhead. The EMB-tree is storage demanding as well since the addition of the embedded trees decreases the index fanout substantially. The MB-tree has the least storage overhead and the EMB^*-tree is a good compromise between the MB-tree and the EMB-tree for this metric. Notice that the proposed cost models capture very accurately the tree sizes.

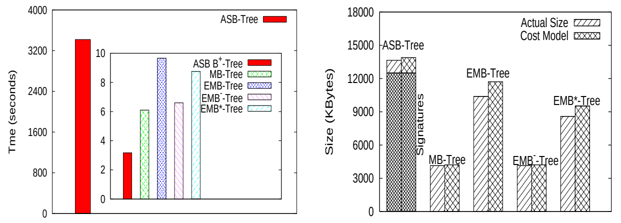

(a) Time (b) Storage

Figure 7: Index construction cost.

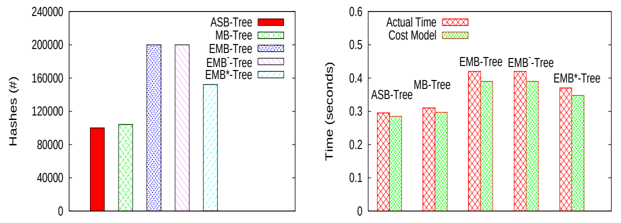

(a) Hash computations (b) Time

Figure 8: Hash computation overhead.

The client/server communication cost using the simplest possible strategy is directly related to the size of the authenticated structures. It should be stressed however that for the hash based approaches this cost can be reduced significantly by rebuilding the trees at the server side. In contrast, the ASB-tree is not equally flexible since all signatures have to be computed at the owner.

The construction cost of our new authenticated index structures has two components. The I/O cost for building the trees and the computational cost for computing hash values. Figure 8 shows the total number of hash computations executed per structure, and the total time required. Evidently, the EMB-tree approaches increase the number of hashes that need to be computed. However, the additional computation time increases by a small margin as hashing is cheap, especially when compared with the total construction overhead (see Figure 7). Thus, the dominating cost factor proves to be the I/O operations of the index.

6.2.2 *Query and Verification Cost*

In this section we study the performance of the structures for range queries. We generate a number of synthetic query workloads with range queries of given selectivity. Each workload contains 100 range queries generated uniformly at random over the domain of A. Results reflect the average case, where the cost associated with accessing the actual records in the database has not been included. A 100 page LRU buffer is used, unless otherwise specified. In the rest of the experiments we do not include the cost model analysis not to clutter the graphs.

The results are summarized in Figure 9. There are two contributing cost factors associated with answering range queries. The first is the total number of I/Os. The second is the computation cost for constructing the $\mathcal{VO}$. The number of I/Os can be further divided into query specific I/Os (i.e., index traversal I/Os for answering the range query) and $\mathcal{VO}$ construction related I/Os.

Figure 9(a) shows the query specific I/Os as a function of selectivity. Straightforwardly, the number of page

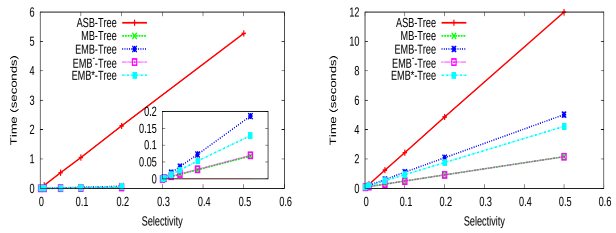

(a) Query runtime with LRU buffer. (b) Query runtime without LRU buffer.

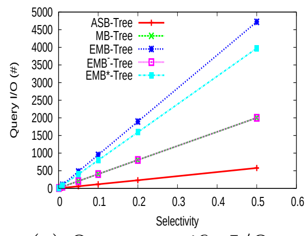
Figure 10: The effect of the LRU buffer.

access is directly related to the fanout of each tree. Notice that the majority of page access is sequential I/O at the leaf level of the trees. Figure 9(b) shows the additional I/O needed by each structure for completing the $\mathcal{VO}$. Evidently, the ASB-tree has to perform a very large number of sequential I/Os for retrieving the signatures of the results. Our authenticated index structures need to do only a few (upper bounded by the height of the index) extra random accesses for traversing the path that leads to the upper boundary tuple of the query result. Figure 9(c) shows the total I/O incurred by the structures. It is clear that the ASB-tree has the worst performance overall, even though its query specific performance is the best.

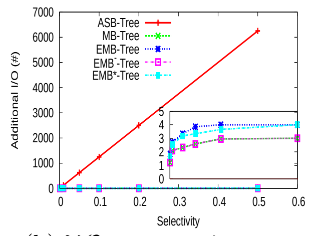
Figure 9(d) shows the runtime cost of additional computations that need to be performed for modular multiplications and hashing operations. The ASB-tree has an added multiplication cost for producing the aggregated signature. This cost is linear to the query result-set size and cannot be omitted when compared with the I/O cost. This observation is instructive since it shows that one cannot evaluate analytically or experimentally authenticated structures correctly only by examining I/O performance. Due to expensive cryptographic computations, I/O operations are not always a dominating factor. The EMB^--tree has a minor computation overhead, depending only on the fanout of the conceptual embedded tree. The rest of the structures have no computation overhead at all.

Interesting conclusions can be drawn by evaluating the effects of the main memory LRU buffer. Figure 10 shows the total query runtime of all structures with and without the LRU buffer. We can deduce that the LRU buffer reduces the query cost substantially for all techniques. We expect that when a buffer is available the computation cost is the dominant factor in query runtime, and the ASB-tree obviously has much worse performance, while without the buffer the I/O cost should prevail. However, since overall the ASB-tree has the largest I/O cost, the hash based structures still have better query performance.

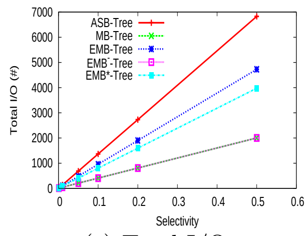
Finally, we measure the $\mathcal{VO}$ size and verification cost at the client side. The results are shown in Figure 11. The ASB-tree, as a result of using aggregated signatures always returns only one signature independent of the result-set size. The MB-tree has to return $f_m \log_{f_m} N_D$ number of hashes plus one signature. As $f_m \gg \log_{f_m} N_D$ the fanout is the dominating factor, and since the MB-tree has a relatively large fanout, the $\mathcal{VO}$ size is large. The EMB-tree and its variants, logically work as an MB-tree with fanout f_k and hence their $\mathcal{VO}$ sizes are significantly reduced, since $f_m \gg f_k$. Notice that the EMB^*-tree has the smallest $\mathcal{VO}$ among all embedded index structures, as the searches in its embedded multi-way search trees can stop at any level of the tree, reducing the total number of hashes.



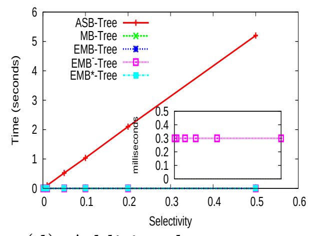
(a) Query specific I/Os.



(b) $\mathcal{VO}$ construction specific I/Os.

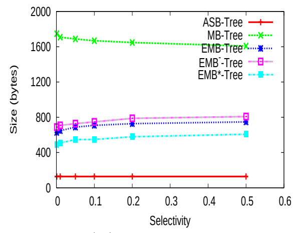


(c) Total I/O.

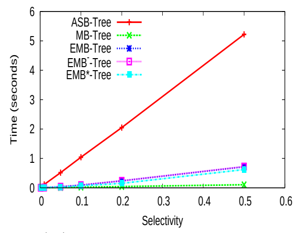


(d) Additional computation cost.

Figure 9: Performance analysis for range queries.



(a) $\mathcal{VO}$ size.



(b) Verification time.

Figure 11: Authentication cost.

The verification cost for the ASB-tree is linear to the size of the query result-set due to the modular multiplication operations, resulting in the worst performance. For the other approaches the total cost is determined by the total hashes that need to be computed. Interestingly, even though the MB-tree has the largest $\mathcal{VO}$ size, it has the fastest verification time. The reason is that for verification the number of hash computations is dominated by the height of the index, and the MB-tree has much smaller height compared to the other structures.

6.2.3 *Update Cost*

There are two types of update operations, insertions and deletions. To expose the behavior of each update type, we perform experiments on update workloads that contain either insertions or deletions. Each update workload contains 100 batch update operations, where each batch operation consists of a number of insertions or deletions, ranging from a ratio $\sigma = 1\%$ to 50% of the total database size before the update occurs. Each workload contains batch operations of equal ratio. We average the results on a per batch update operation basis. Two distributions of update operations are tested. Ones that are generated uniformly at random, and ones that exhibit a certain degree of locality. Due to lack of space we present here results only for uniform insertions. Deletions worked similarly. Skewed distributions exhibit a somewhat improved performance and have similar effects on all approaches. Finally, results shown here include only the cost of updating the structures and not the cost associated with updating the database.

Figure 12 summarizes the results for insertion operations. The ASB-tree requires computing between $\sigma N_D + 1$ and $2\sigma N_D$ signatures. Essentially, every newly inserted tuple requires two signature computations, unless if two new tuples are consecutive in order in which case one computation can be avoided. Since the update operations are uniformly distributed, only a few such pairs are expected on average. Figure 12(a) verifies this claim. The rest of the structures require only one signature re-computation.

The total number of pages affected is shown in Figure 12(b). The ASB-tree needs to update both the pages containing the affected signatures and the B^+-tree structure. Clearly, the signature updates dominate the cost as they are linear to the number of update operations. Other structures need to update only the nodes of the index. Trees with smaller fanout result in larger number of affected pages. Even though the EMB$^-$-tree and MB-tree have smaller fanout than the ASB-tree, they produce much smaller number of affected pages. The EMB-tree and EMB*-tree produce the largest number of affected pages. Part of the reason is because in our experiments all indexes are bulk-loaded with 70% utilization and the update workloads contain only insertions. This will quickly lead to many split operations, especially for indexes with small fanout, which creates a lot of new pages.

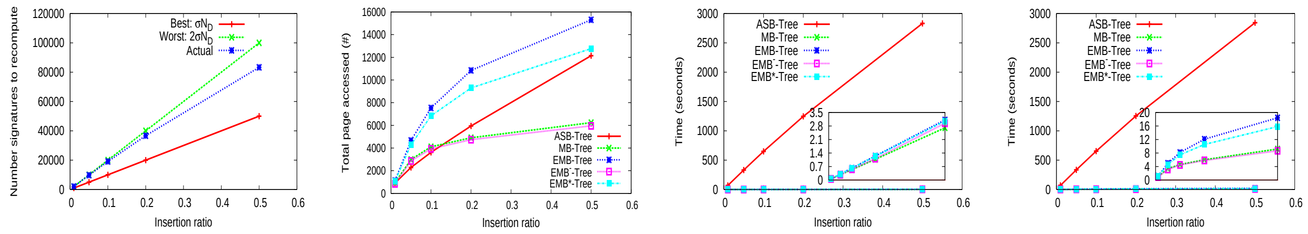
Another contributing factor to the update cost is the computation overhead. As we can see from Figure 12(c) the ASB-tree obviously has the worst performance and its cost is order of magnitudes larger than all other indexes, as it has to perform linear number of signature computations (w.r.t the number of update operations). For other indexes, the computation cost is mainly due to the cost of hashing operations and index maintenance. Finally, as Figure 12(d) shows, the total update cost is simply the page I/O cost plus the computation cost. Our proposed structures are the clear winners. Finally the communication cost incurred by update operations is equal to the number of pages affected.

6.2.4 *Discussion*

The experimental results clearly show that the authenticated structures proposed in this paper perform better than the state-of-the-art with respect to all metrics except the $\mathcal{VO}$ size. Still, our optimizations reduced the size to four times the size of the $\mathcal{VO}$ of the ASB-tree. Overall, the EMB^--tree gives the best trade-off between all performance metrics, and it should be the preferred technique in the general case. By adjusting the fanout of the embedded trees, we obtain a nice trade-off between query (VO) size, verification time, construction (update) time and storage overhead.

7. CONCLUSION

We presented a comprehensive evaluation of authenticated index structures based on a variety of cost metrics and taking into account the cost of cryptographic operations, as well as that of index maintenance. We proposed a novel structure that leverages good performance based on all metrics. We extended the work to dynamic environments, which has not been explored in the past. We also formulated the problem of query freshness, a direct outcome of the dynamic case. Finally, we presented a comprehensive ex-



(a) Number of signature re-computations. (b) Number of pages affected. (c) Computation cost. (d) Total update time.

Figure 12: Performance analysis for insertions.

perimental evaluation to verify our claims. For future work, we plan to extend our ideas for multidimensional structures, and explore more involved types of queries.

8. ACKNOWLEDGMENTS

This work was partially supported by NSF grants IIS-0133825, CCR-0311485 and CCF-0515100. The authors would like to thank the anonymous reviewers for their constructive comments.

9. REFERENCES

[1] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proc. of ACM Management of Data (SIGMOD)*, pages 439–450, 2000.

[2] Authenticated Index Structures Library. http://cs-people.bu.edu/lifeifei/aisl/.

[3] E. Bertino, B. Carminati, E. Ferrari, B. Thuraisingham, and A. Gupta. Selective and authentic third-party distribution of XML documents. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(10):1263–1278, 2004.

[4] L. Bouganim, C. Cremarenco, F. D. Ngoc, N. Dieu, and P. Pucheral. Safe data sharing and data dissemination on smart devices. In *Proc. of ACM Management of Data (SIGMOD)*, pages 888–890, 2005.

[5] L. Bouganim, F. D. Ngoc, P. Pucheral, and L. Wu. Chip-secured data access: Reconciling access rights with data encryption. In *Proc. of Very Large Data Bases (VLDB)*, pages 1133–1136, 2003.

[6] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[7] Crypto++ Library. http://www.eskimo.com/∼weidai/cryptlib.html.

[8] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic data publication over the internet. *Journal of Computer Security*, 11(3):291–314, 2003.

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

[10] A. Evfimievski, J. Gehrke, and R. Srikant. Limiting privacy breaches in privacy preserving data mining. In *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, pages 211–222, 2003.

[11] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):96–99, 1988.

[12] H. Hacigumus, B. R. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database service provider model. In *Proc. of ACM Management of Data (SIGMOD)*, pages 216–227, 2002.

[13] H. Hacigumus, B. R. Iyer, and S. Mehrotra. Providing database as a service. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 29–40, 2002.

[14] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *Proc. of Very Large Data Bases (VLDB)*, pages 720–731, 2004.

[15] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Authenticated Index Structures for Outsourced Database Systems. Technical Report BUCS-TR_2006-004, CS Department, Boston University, 2006.

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

[17] K. McCurley. The discrete logarithm problem. In *Proc. of the Symposium in Applied Mathematics*, pages 49–74. American Mathematical Society, 1990.

[18] R. C. Merkle. A certified digital signature. In *Proc. of Advances in Cryptology (CRYPTO)*, pages 218–238, 1989.

[19] S. Micali. Efficient certificate revocation. Technical Report MIT/LCS/TM-542b, Massachusetts Institute of Technology, Cambridge, MA, March 1996.

[20] G. Miklau and D. Suciu. Controlling access to published data using cryptography. In *Proc. of Very Large Data Bases (VLDB)*, pages 898–909, 2003.

[21] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In *Symposium on Network and Distributed Systems Security (NDSS)*, 2004.

[22] E. Mykletun, M. Narasimha, and G. Tsudik. Signature bouquets: Immutability for aggregated/condensed signatures. In *European Symposium on Research in Computer Security (ESORICS)*, pages 160–176, 2004.

[23] M. Narasimha and G. Tsudik. Dsac: Integrity of outsourced databases with signature aggregation and chaining. In *Proc. of Conference on Information and Knowledge Management (CIKM)*, pages 235–236, 2005.

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

[25] OpenSSL. http://www.openssl.org.

[26] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. In *Proc. of ACM Management of Data (SIGMOD)*, pages 407–418, 2005.

[27] H. Pang and K.-L. Tan. Authenticating query results in edge computing. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 560–571, 2004.

[28] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM (CACM)*, 21(2):120–126, 1978.

[29] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *Proc. of ACM Management of Data (SIGMOD)*, pages 551–562, 2004.

[30] R. Sion. Query execution assurance for outsourced databases. In *Proc. of Very Large Data Bases (VLDB)*, pages 601–612, 2005.

[31] R. Tamassia and N. Triandopoulos. Efficient Content Authentication over Distributed Hash Tables. Technical report, CS Department, Brown University, 2005.