Authenticated Relational Tables and Authenticated Skip Lists *

Giuseppe Di Battista[1] and Bernardo Palazzi[1,2,3]

[1] Roma TRE University, Rome Italy, {gdb,palazzi}@dia.uniroma3.it,
[2] ISCOM Italian Ministry of Communication, Rome, Italy,
[3] Brown University, Department of Computer Science, Providence, RI USA

Abstract. We present a general method, based on the usage of typical DBMS primitives, for maintaining authenticated relational tables. The authentication process is managed by an application external to the DBMS, that stores just one hash information of the authentication structure. The method exploits techniques to represent hierarchical data structures into relational tables and queries that allow an efficient selection of the elements needed for authentication.

Key words: Authenticated Relational Table, Authenticated Skip List, Authenticated query

1 Introduction

We consider the following scenario. A user needs to store data in a relational database, where the Data Base Management System (DBMS) is shared with other users. For example, the DBMS is available on-line through the Web, and anybody in the Internet can store and access data on it. Nowadays, there are many sites providing services of this type [1, 22, 25, 30] and the literature refers to such facilities as to *outsourced databases* [13, 19, 27].

When the database is accessed, the user wants to be sure on the integrity of her/his data, and wants to have the proof that nobody altered them.

Of course, accessing the DBMS is subject to authentication restrictions, and the users must provide credentials to enter. However, the user might not trust the DBMS manager, or the site that provides the service, or even the DBMS software. Extending the argument, the same problem can be formulated even in terms of a traditional database. Also in this case, with the current technologies, although DBMSes put at disposal logs of the performed transactions and other security features, for a user it is somehow impossible to be completely sure that nobody altered the data.

A first attempt for the user to be sure of the authenticity of the data is to put a signature on each t-uple of each relational table of the database. Unfortunately, this technique does not provide enough security. In fact, adversaries could

* This work was supported in part by grant IIS–0324846 from the National Science Foundation and a gift from IAM Technology, Inc.

remove some t-uples and the user would not have any evidence of this. Another straightforward possibility would be to sign each table as a whole. However, this does not scale-up, and even mid-size tables would be impossible to authenticate.

We propose a method and a prototype for solving the above mentioned problem. Namely, for each relational table R of the user we propose to store in an extra relational table $S(R)$ (in the following *security table*) of the DBMS a special version of authenticated data structure that allows to verify the authenticity of R.

With this approach, if the user wants to have the proof of authenticity of R, it is sufficient to check the values of a few elements stored in $S(R)$. On the other hand, if the user updates R, only a few variations on $S(R)$ are needed to preserve the proof of authenticity. We also propose efficient techniques to manage and to query $S(R)$ and show the practical feasibility of the approach.

Observe that the proposed approach is completely independent on the specific adopted DBMS and can be implemented into an extra software layer or either a plug-in, under the sole responsibility of the user. The authentication process is managed by an application external to the DBMS that stores just a constant size ($O(1)$ wrt the size of R) secret. The method does not require trust in the DB manager or DBMS.

The paper is organized as follows. Section 2 provides basic terminology and summarizes the state of the art. Section 3 describes the adopted model. Sections 4 and 5 provide technical insights. Section 6 presents experiments that show the feasibility of the approach. Section 7 concludes the paper analyzing the security of the approach and proposing future work.

2 Background and State of the Art

Authenticated data structures (ADS) [29] have been devised to be used in a computational model where untrusted responders answer queries on a data structure on behalf of a trusted source and provide to the user a proof of the validity of the answer. Early work on ADS was originated by the certificate revocation problem in $PKIs$ and focused the attention on authenticated dictionaries, on which membership queries are performed.

The Merkle hash tree MHT scheme [16] can be used to implement a static authenticated dictionary. An MHT of a set stores cryptographic hashes of the value of elements belonging to the set at the leaves of the MHT and an authentication value at each internal node, which is the result of computing a cryptographic hash function on the values of its children. The MHT uses linear space and has $O(\log n)$ proof size, query time and verification time.

A dynamic authenticated dictionary that uses a hierarchical hashing technique over skip lists, a data structure introduced by Pugh [26], is presented in [8, 9]. Such ADS obtains $O(\log n)$ proof size, query time, update time and verification time. Other schemes based on variations of MHT have been proposed in [2, 4, 12, 20]. A detailed analysis of the efficiency of authenticated dictionary schemes based on hierarchical cryptographic hashing is conducted in [28], where

precise measures of the computational overhead due to the authentication are introduced. Lower bounds on the authentication cost are given, existing authentication schemes are analyzed, and a new authentication scheme is presented that achieve performance very close to the theoretical optimal.

The notion of a two parties model in ADS is introduced in [10], where only the client needs to maintain the proof of validity for his data.

A first step towards the design of more general ADS (beyond dictionaries) is done in [7, 14, 21] with a first approach on the authentication of relational database operations and multidimensional orthogonal range queries.

Buldas, in a more recent paper [3], studies how to extend ADS to perform more complex queries and uses optimizations on interval queries. In [23, 24] the authors propose a method to authenticate projection queries using different cryptographic techniques for verifying the completeness of relational queries. While the papers are quite promising in terms of theoretical bounds and analysis, the practical efficiency is not demonstrated.

In [17], Miklau and Suciu proposed to embed into a relational table an MHT, with a model that is similar to the one adopted in this paper. However, the technique is described only partially and seems to have some drawbacks. Namely, validating the result of a query seems to require several distinct queries on the DBMS. This is in contrast with the typical atomicity requirements of concurrency. Also, the $MHTs$ require frequent rebalancing for supporting updates and it is unclear how to match this requirement with the need to have a few updates in the relational table. Further, the time performance illustrated in the paper are not supported by a clear description of the experimental platform and show some inconsistency. For example in one of the tests the time requested for authentication decreases with the growth of the table.

For the purposes of this paper we need to provide a description of the skip lists. The skip list data structure [26] is an efficient tool for storing an ordered set of *elements*. It supports the following operations on a set of elements.

- **find**(x): Determine whether element x is in the set.
- **insert**(x): Insert element x into the set.
- **delete**(x): Remove element x from the set.

A skip list S stores a set of elements in a sequence of linked lists $S_0, S_1, \ldots, S_t$ called *levels*. The members of the lists are called *nodes*. The base list, S_0, stores in its nodes all the elements of S in order, as well as *sentinels* associated with the special elements $-\infty$ and $+\infty$. Each list S_{i+1} stores a subset of the elements of S_i. The method used to define the subset from one level to the next determines the type of skip list. The default method is simply to choose the elements of S_{i+1} at random among the elements of S_i with probability $\frac{1}{2}$. One could also define a deterministic skip list [18], which uses simple rules to guarantee that between any two elements in S_i there are at least 1 and at most 3 elements of S_{i+1}. In either case, the sentinel elements $-\infty$ and $+\infty$ are always included in the next level up, and the top level, is maintained to be $O(\log n)$. We therefore distinguish the node of the top list S_t storing $-\infty$ as the start node s.

An element that is in S_{i-1} but not in S_i is said to be a *plateau element* of S_{i-1}. An element that is in both S_{i-1} and S_i is said to be a *tower element* in S_{i-1}. Thus, between any two tower elements, there are some plateau elements. In randomized skip lists, the expected number of plateau elements between two tower elements is one. The skip list of Fig. 1 has 7 elements (including sentinels). The element 6 is stored in 3 nodes with different level. The overall number of nodes is 17.

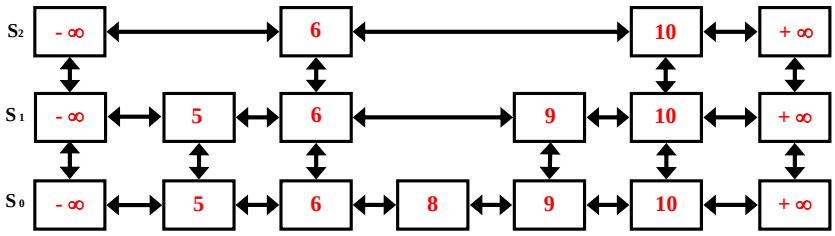

Fig. 1. Skip List

To perform a search for element x in a skip list, we begin at the start node s. Let v denote the current node in our search (initially, $v = s$). The search proceeds using two actions, *hop forward* and *drop down*, which are repeated one after the other until we terminate the search. See Fig. 2.

- $Hopforward$: We move right along the current list until we find the node of the current list with largest element less than or equal to x. That is, while $elem(right(v)) < x$, we perform $v = right(v)$.
- $Dropdown$: If $down(v) = null$, then we are done with our search: node v stores the largest element in the skip list less than or equal to x. Otherwise, we update $v = down(v)$.

In a deterministic skip list, the above searching process is guaranteed to take $O(\log n)$ time. Even in a randomized skip list, it is fairly straightforward to show (e.g., see [11]) that the above searching process runs in expected $O(\log n)$ time, for, with high probability, the height t of the randomized skip list is $O(\log n)$ and the expected number of nodes visited on any level is 3.

To insert a new element x, we determine which lists should contain the new element x by a sequence of simulated random coin flips. Starting with $i = 0$, while the coin comes up heads, we use the stack A to trace our way back to the position of list S_{i+1} where element x should go, add a new node storing x to this list, and set $i = i + 1$. We continue this insertion process until the coin comes up tails. If we reach the top level with this insertion process, we add a new top level on top of the current one. The time taken by the above insertion method is $O(\log n)$ with high probability. To delete an existing element x, we remove

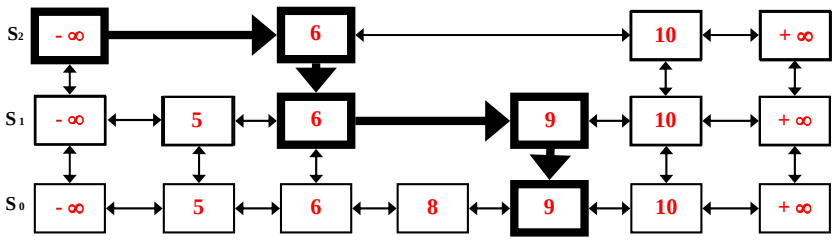

Fig. 2. A value searching in a Skip List: search for element 9 in the skip list of Figure 1. The nodes visited and the links traversed are drawn with thick lines and arrows.

all the nodes that contain the element x. This takes time is $O(\log n)$ with high probability.

To introduce the *Authenticated Skip Lists* we need to use the commutative hash technique [9] developed by Gooodrich and Tamassia. A hash function h is commutative if $h(x;y) = h(y;x)$, for all x and y. Given a cryptographic hash function h that is collision resistant in the usual sense, we construct a candidate commutative cryptographic hash function, h_0, as follows [9] :

$h_0(x,y) = h(min(x,y), max(x,y))$

It can be shown that h_0 is commutatively collision resistant [9].

The authenticated skip list introduced in [9] consists of a skip list where each node v stores a label computed accumulating the elements of the set with a commutatively cryptographic hash function h. For completeness, let us review how hashing occurs. See [9] for details. For each node v we define label $f(v)$ in terms of the respective values at nodes $w = right(v)$ and $u = down(v)$. If $right(v) = null$, then we define $f(v) = 0$. The definition of $f(v)$ in the general case depends on whether u exists or not for this node v.

- $u = null$, i.e., v is on the base level:
  - If w is a tower node, then
    $f(v) = h(elem(v), elem(w))$
  - If w is a plateau node, then
    $f(v) = h(elem(v), f(w))$.
- $u \neq null$, i.e., v is not on the base level:
  - If w is a tower node, then
    $f(v) = f(u)$.
  - If w is a plateau node, then
    $f(v) = h(f(u), f(w))$.

We illustrate the flow of the computation of the hash values labeling the nodes of a skip list in See Fig. 3. Note that the computation flow defines a directed acyclic graph DAG, not a tree. After performing the update in the skip list, the hash values must be updated to reflect the change that has occurred. The additional computational expense needed to update all these values is expected with high probability to be $O(\log n)$. The verification of the answer to a query

is simple, thanks to the use of a commutative hash function. Recall that the goal is to produce a verification that some element x is or is not contained in the skip list. In the case when the answer is "yes", we verify the presence of the element itself. Otherwise, we verify the presence of two elements x_a and x_b stored at consecutive nodes on the bottom level S_0 such that $x_a < x < x_b$. In either case, the answer authentication information is a single sequence of values, together with the signed, timestamped, label $f(s)$ of the start node s.

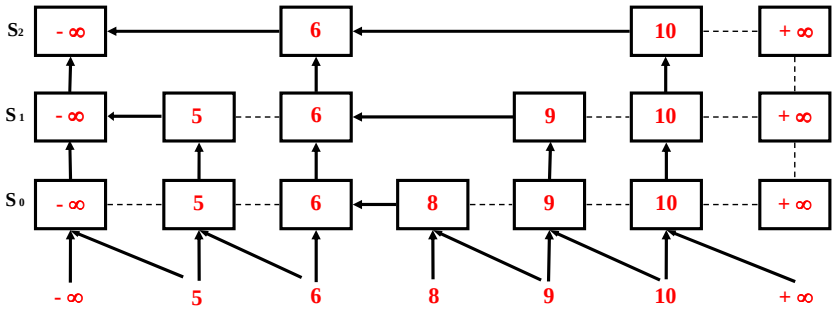

Fig. 3. Authenticated Skip List: Flow of the computation of the hash values labeling the nodes of the skip list of Fig. 2. Nodes where hash functions are computed are drawn with thick lines. The arrows denote the flow of information, not links in the data structure.

Let $P(x) = (v_1; ...; v_m)$ be the sequence of nodes that are visited when searching for element x, in reverse order. In the example of Fig. 4, we have $P(9)$ that needs not only the nodes $(9, 6, -\infty)$ with the thick line but also all the siblings with the stroke dash-dot-dash-dot. Note that by the properties of a skip list, the size m of sequence $P(x)$ is $O(\log n)$ with high probability. We construct from the node sequence $P(x)$ a sequence $Q(x) = (y_1; ...; y_m)$ of values such that:

- $y_m = f(s)$, the label of the start node;
- $y_m = h(y_{m-1}; h(y_{m-2}; h(...; y_1)...)))$

The user verifies the answer for element x by simply hashing the values of the sequence $P(x)$ in the given order, and comparing the result with the signed value $f(s)$, where s is the start node of the skip list. If the two values agree, then the user is assured of the validity of the answer at the time given by the timestamp.

3 The Reference Model

A user stores a relational table R into a DBMS. The user would like to perform the usual relational operations on R, namely, would like to select a set of t-uples, to insert elements, and to delete elements. The user wants to verify that a query result is authentic. The amount of information that the user has to maintain in

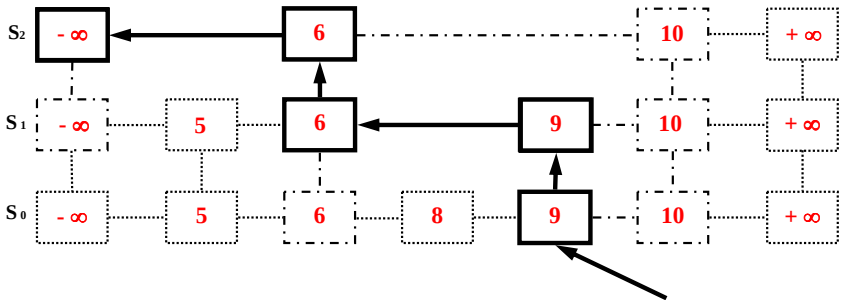

Fig. 4. Values needed to authenticate the result of a query.

a secure environment to be certain of the authenticity of the answer should be kept small (ideally constant size) with respect to the size of R.

We propose to equip R with an authenticated skip list A to guarantee its integrity. Of course, there are at least two approaches for implementing A. Either A is stored in main memory within an application controlled by the user, or A is stored into the same DBMS storing R. We follow the second approach. Namely, we investigate how to efficiently store A into a further relational table $S(R)$, called *security table*, used only for that purpose. Fig. 5 shows a relational table, an authenticated skip list for its elements, and the implementation of the skip-list into a second relational table.

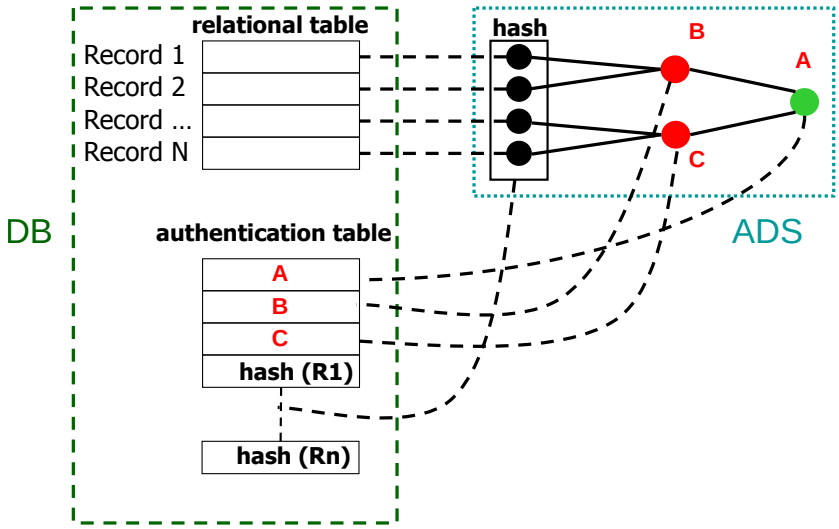

Fig. 5. A relational table and its security table.

There are two options. We call them the *coarse-grained* and the *fine-grained* approach.

What we call coarse-grained approach is probably the most natural way to represent an authenticated skip list S inside a relational table $S(R)$. Namely, it consists of storing each element of S inside a specific record of $S(R)$. On the other hand, the fine-grained approach shifts the attention on a smaller element of S. It consists of storing each level of an element of S inside a record of $S(R)$.

In order to visualize the coarse-grained approach, it is effective to think at S in terms of a "quarter clockwise rotation". As an example, Table 1 is a coarse-grained representation of the authenticated skip list of Fig. 6.

More precisely, the fields of Table 1 have the following meaning.

- **Key**: The value of an element of S. It can be any type of value, not only a number, but on such a type a total ordered must be defined.
- **Prv** n - **Nxt** n: Pointers to the previous and to the next element in S, for each level n.
- **Hash** n: Information needed to authenticate S, stored at each level n.

Each element of S has a height, that is, the number of nodes with the same value of key that constitute an element of S, that is randomly determined. This is the main trade-off of this technique, because on one hand this kind of representation has the property to maintain the identity between the number of records in $S(R)$ and the elements present in S, but on the other hand it has an overhead in the size of the table, because each record has a number of fields equal to the highest S in A. This is necessary because we do not know the height of a new S and then we have to arrange $S(R)$ for worst cases, when an S is at the highest level. So, we must pad with "null" values the fields that do not reach the highest level.

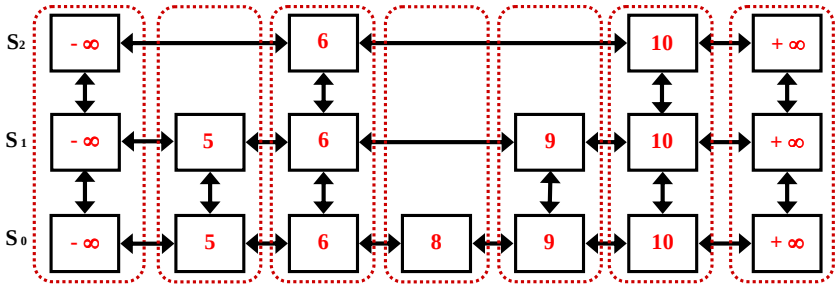

Fig. 6. Storing a Skip List inside a Relational Table.

Once stated how to represent S inside the security table $S(R)$, we developed methods to perform in S a set of authenticated relational operations, without the need to load in main memory the whole $S(R)$. Performing authenticated

Key	Hash 0	Prv 0	Nxt 0	Hash 1	Prv 1	Nxt 1	Hash 2	Prv 2	Nxt 2
- ∞	f($-\infty$, 5)	*null*	5	**f(f($-\infty$), f(5))**	*null*	5	**f(f($-\infty$),f(6))**	*null*	**6**
5	f(5, 6)	$-\infty$	6	f(f(5), f(6))	$-\infty$	6	*null*	*null*	*null*
6	**f(6, f(8))**	5	8	**f(f(6), f(9))**	5	**9**	$f(f(6), f(10))$	10	$-\infty$
8	f(9,6)	9	6	*null*	*null*	*null*	*null*	*null*	*null*
9	**f(9, 10)**	8	10	**f(9, 10)**	**6**	**10**	*null*	*null*	*null*
10	f(10, f(+∞))	9	+∞	**f(f(10), f(+∞))**	9	+∞	**f(f(10), f(+∞))**	6	+∞

Table 1. A coarse-grain representation of an authenticated skip list into a relational table. In bold face the elements necessary to authenticate element 9.

operations on R requires the usage of queries that retrieve all the elements that are needed to compute the authentication path. Such elements are spread on all $S(R)$. The main requirements in devising such queries are:

- The need to build queries that retrieve only the authentication elements that are strictly necessary, to reduce, as much as possible, the amount of required memory.
- The need of fast queries that allow to authenticate a result with a small time overhead. In this respect it is meaningful to minimize the number of used queries.

It is important to perform such queries using only standard SQL. In fact, our model does not allow any modification of the DBMS engine. Also, thinking in terms of SQL allows the identification of a precise interface between an authentication tool based on our techniques and the DBMS, allowing its implementation in terms of a plug-in. The main idea here is to use an algorithm that retrieves the authentication elements, starting from the knowledge of the value K to authenticate:

1. We perform a query that loads in memory all the records that are not *null* at top level and that have a value smaller than K.
2. We select the greatest element in the query result (that is the predecessor of K at the top level).
3. We perform an interval query on the elements (that are not *null*) at the immediately lower level, with the following range: from the element retrieved in the previous step to the element stored in its field next to the top level.
4. We repeat the steps $2-3$ until we reach level 0.

In order to understand which elements are loaded in main memory by queries of the algorithm, it is effective to think at a shape like a "funnel" that has its stem on K. See Fig. 7. The loaded elements are those that "touch" the funnel.

Note that the number of queries that is needed to retrieve the authentication root path is proportional to the number of levels in S, that is logarithmic in the number of elements that are currently present in S.

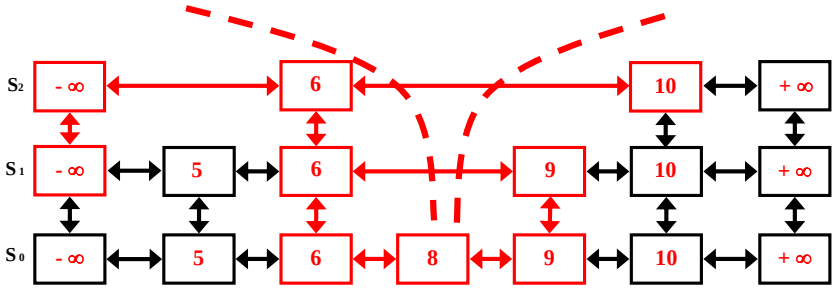

Fig. 7. Loaded elements in an authentication query.

4 A Fine Grained Approach

This approach stores inside each record a node instead of an element of S. A node is an invariant-size component in S. Hence, it can be stored in a record with a fixed number of fields, independently on the number of elements stored in S. More precisely, in this case the fields of $S(R)$ have the following meaning:

- **Key**: value of an element of S;
- **Level**: height of an element of S, that is the number of the lists that the element belongs to;
- **prvKey-nxtKey**: pointers to the previous and to the next element of S at the same level;
- **parentLvl-parentKey**: pointer to the parent element in the path of authentication; it is needed to allow the retrieval of the root path;
- **Hash**: information needed for the authentication, performed with the method used in S [9].

The direct storage of S nodes significantly reduces the space overhead, that it is typical of the coarse grain approach. In fact, in this case there is no need to store *null* values.

This approach allows the usage of very efficient techniques to manage $S(R)$ dynamically and securely. The method we adopt is based on the *nested set* method for storing hierarchical data structures inside adjacency lists, that in turn fit well into relational tables [5] See Fig. 8 and Tab. 2.

5 Exploiting Nested Sets

The problem of storing hierarchical data structures inside relational tables has been already studied in database theory [6, 15]. The solution that we exploit is due to Celko [5], that shows a method to store a tree inside a relational table. Such a method is based on augmenting the table with two extra fields.

In order to understand what is a nested set, it is effective to think at the nodes of the tree as circles and to imagine that the circles of the children are

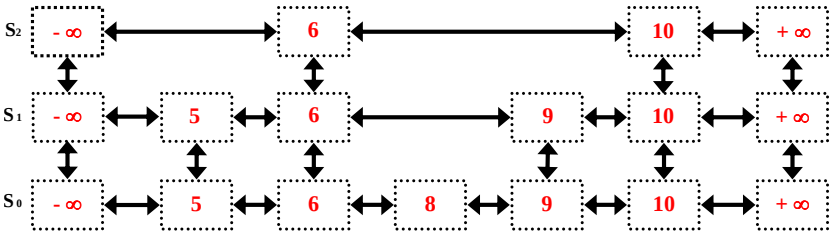

Fig. 8. Storing a Skip List inside a Relational Table. A Fine Grained Approach

Key	Level	prvKey	nxtKey	parentLvl	parentKey	Hash
$-\infty$	**2**	*null*	6	*null*	*null*	**f(f($-\infty$), f(6))**
$-\infty$	**1**	*null*	5	2	$-\infty$	**f(f($-\infty$), f(5))**
$-\infty$	**0**	*null*	5	1	$-\infty$	**f($-\infty$, 5)**
5	1	$-\infty$	6	1	$-\infty$	f(5,6)
5	0	$-\infty$	6	1	5	f(5, 6)
6	**2**	$-\infty$	**10**	**2**	$-\infty$	f(**f(6),f(10)**)
6	**1**	5	**9**	**2**	**6**	**f(f(6),f(9))**
6	**0**	5	8	1	6	**f(6, f(8))**
8	0	6	9	0	6	f(8, 10)
9	**1**	**6**	**10**	**1**	**6**	**f(9, 10)**
9	**0**	8	10	1	9	**f(9, 10)**
10	**2**	6	$+\infty$	2	6	**f(10,f($+\infty$))**
10	**1**	9	$+\infty$	2	10	**f(10,f($+\infty$))**
10	0	9	$+\infty$	2	10	f(10,f($+\infty$))

Table 2. A fine grain representation of an authenticated skip list into a relational table. In bold the elements necessary to authenticate element 9

nested inside their parent. The root of the tree is the largest circle and contains all the other nodes. The leaf nodes are the innermost circles, with nothing else inside them. The nesting shows the hierarchical relationship.

The two extra fields have the role of left and right boundaries of the circle and allow to represent the nesting of the hierarchy.

Unfortunately, skip lists are not *trees* but a directed acyclic graph. Hence, we have to extend the nested set method to this different setting. Table 3 illustrates how the fine-grained approach can be equipped with nested-sets features. Observe the Left and Right fields that represent the boundaries of the "circles". Fig. 9 shows the correspondence between boundaries and nodes of the skip-list. The figure shows also a root path.

Now we show one of the features of the proposed approach. Namely, we argue that, in order to authenticate an element of a relational table R, we need just one query on $S(R)$. Such a query is used to retrieve the complete *root-path* and all its *sibling* elements. Observe that, authenticating an element in an ADS requires a number of steps that is logarithmic (worst case or average case) in the number

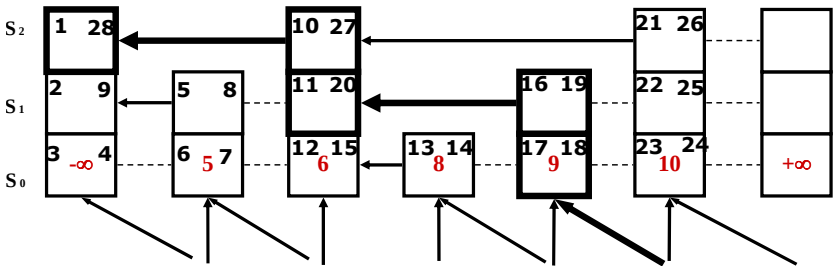

Fig. 9. An ADS and its Nested Set. Thick lines show the authentication root path for element 9.

of the elements while this logarithmic dependence does not yield a logarithmic number of queries in our case but a constant number of queries. We make the

Key	Level	prvKey	nxtKey	parentLvl	parentKey	Left	Right
$-\infty$	**2**	*null*	6	*null*	*null*	**1**	**28**
$-\infty$	**1**	*null*	5	2	$-\infty$	**2**	**9**
$-\infty$	**0**	*null*	5	1	$-\infty$	**3**	**4**
5	**1**	$-\infty$	6	1	$-\infty$	**5**	**8**
5	**0**	$-\infty$	6	1	5	**6**	**7**
6	**2**	$-\infty$	10	2	$-\infty$	**10**	**27**
6	**1**	5	9	2	6	**11**	**20**
6	**0**	5	8	1	6	**12**	**15**
8	**0**	6	9	0	6	**13**	**14**
9	**1**	6	10	1	6	**16**	**19**
9	**0**	8	10	1	9	**17**	**18**
10	**2**	6	$+\infty$	2	6	**21**	**26**
10	**1**	9	$+\infty$	2	10	**22**	**25**
10	**0**	9	$+\infty$	2	10	**23**	**24**

Table 3. A representation of an authenticated skip list into a relational table using nested set. In bold the key value and the *left* and *right* fields. The 2 extra fields added are needed for fast queries.

argument using an example. The following query uses directly the value of the element to authenticate. The example is for the authentication of element 9.

```
SELECT    *
FROM          skiplist
WHERE  Left <= (SELECT     Left
                FROM      skiplist
                WHERE      key =  9 AND level = 0)
       AND  Right >= (
```

```
SELECT     Right
FROM          skiplist
WHERE      key =  9 AND level = 0);
```

The above query retrieves only the authentication root-path starting from 9. To validate 9 we have to retrieve also all sibling nodes of the root-path. This is possible by using two subqueries that retrieve all elements that are:

- in the fields nxtKey of the root-path;
- on the level below and with the same key of the root-path.

Using this method we built a quick algorithm to get the complete authentication path needed to validate a table interrogation, using only one query, that is that all concurrency problems related to selection queries will be managed by the DBMS. Also, it is possible to modify the query in order to retrieve all the information needed to authenticate all the t-uples obtained by a Select with just one query.

6 Experimental Evaluation

This section shows the experimental results obtained using a prototype implementation of the techniques presented in the previous sections. The Hardware architecture where tests have been performed consists of quite common laptop with following features:

- cpu intel©centrino^{TM}duo T2300 (1.66 GHz, 667 FSB);
- RAM 1.5 Gb DDR2
- HDD 5,400 rpm Serial ATA

The Software architecture consists of following elements:

- Microsoft©Windows^{TM}XP Tablet edition 2005;
- Java^{TM}version 1.5
- MySql JDBC Connector Java-bean 5.03
- MySql DBMS version 4.1

The data sets for tests have been chosen with a scale from $10,000$ to $1,000,000$ of elements. Such elements were sampled at random from a set 10 times larger. All values presented in this section have been computed as average of the results of 5 different tests. The elements in each test are a sample, randomly selected, composed of $\frac{1}{1000}$ of the entire set. All times are in *milliseconds*. All tests show the *clock-wall* time.

The first test is about the authentication of a single value inside a relational table. Table 4 shows the results of the authentication of a single element inside different size authenticated tables, stressing the differences between coarse grain and fine grain approaches. Tests are about the following measures:

- **RAM**: the time to validate a value in main memory;

– **DB → RAM**: the time to load in main memory from a secondary memory storage system (e.g., a hard disk), the elements necessary to validation;
– **NODES**: the numbers of elements loaded from the database in main memory;
– **STEPS**: the numbers of elements actually used in the authentication process, the difference between NODES value and this value shows the overhead of the elements loaded in main memory.

	10,000		100,000		1,000,000	
CHECK	Coarse	Fine	Coarse	Fine	Coarse	Fine
RAM	0	0	0	0	0	0
DB → RAM	36	11	252	42	2680	377
NODES	35	27	44	31	57	43
STEPS	25	27	33	30	39	41

Table 4. Test results for validation of an element inside different size tables. All the results are in ms. Times for fine- and coarse-grained approaches.

The results showed above are very similar to those obtained from the authentication of an element not-present in the table. In fact it is sufficient to check the previous and the next element of the value that is not present to proof the element lack.

The second test is about the insertion of a single value inside an authenticated relational table. The table 5 shows the results of the insertion of a single element inside different size authenticated tables using only coarse grain approach. Tests concern the following measures:

– **RAM**: the time to insert in main memory a value;
– **DB → RAM**: the time to load in main memory from a secondary memory storage system (e.g., a hard disk), the elements necessary to insertion;
– **RAM → DB**: the time to store in secondary memory the elements updated in main memory;

INSERT	10,000	100,000	1,000,000
RAM	0	0	0
DB → RAM	32	260	2605
RAM → DB	14	26	26
Tot. Time	46	286	2631

Table 5. Test results for insertion of an element inside a different size tables. Using coarse grain approach. All results are in ms.

Methods that allow to delete and modify an element inside an authenticated table are similar to times showed for insertion operation.

The obtained experimental results put in evidence the feasibility of the approach. In fact, the time for answering a query is comparable to the one obtained in a non authenticated setting. The fine-grained approach, based on Celko techniques, shows much better performance wrt the coarse-grained one.

7 Conclusions and Future Work

We have described methods that allow a user to verify the authenticity and completeness of simple queries results, even if the database system is not trusted. The overhead for the user is limited at storing only a single hash value. Our work is the first to design and evaluate techniques for authenticated skip list that are appropriate to a relational database, and the first to prove the feasibility of authenticated skip list for integrity of databases.

The security of the presented method is based on the reliability of *ADSes*. There are many works [3, 9, 12, 16] in the literature that demonstrate that the security of *ADS* is based on the difficulty to find useful collisions in a cryptographic hash function. So all the security relies on the effectiveness of hash functions. The prototype used for the experiments uses commutative hashing. In [9] it is demonstrated that commutative hashing does not augment the possibility to find a collision in the used hash function.

In the future we would like to investigate how to authenticate more complex queries making use of a larger set of relational operations. Further, we would like to study models to build integrity verification services in peer to peer systems.

References

1. Web based Database Software Solutions On-Demand. `http://www.teamdesk.net`.
2. A. Buldas, P. Laud, and H. Lipmaa. Accountable certificate management using undeniable attestations. In *ACM Conference on Computer and Communications Security*, pages 9–17, 2000.
3. A. Buldas, M. Roos, and J. Willemson. Undeniable replies for database queries. In *In Proceedings of the Fifth International Baltic Conference on DB and IS, volume 2, pages 215-226, 2002.*, 2002.
4. Barbara Carminati. Selective and authentic third-party distribution of xml documents. *IEEE Transactions on Knowledge and Data Engineering*, 16(10):1263–1278, 2004. Fellow-Elisa Bertino and Member-Elena Ferrari and Fellow-Bhavani Thuraisingham and Senior Member-Amar Gupta.
5. Joe Celko. *Joe Celko's Trees and hierarchiesin SQL for smarties.* Morgan-Kaufmann, 2004.
6. C.J. Date. Why is it so difficult to provide a relational interface to ims. In *Relational Database– Selected Writings*, pages 241–257. Addison-Wesley, 1986.
7. P. T. Devanbu, M. Gertz, C. U. Martel, and S. G. Stubblebine. Authentic third-party data publication. In *Proceedings of the IFIP TC11/ WG11.3 Fourteenth Annual Working Conference on Database Security*, pages 101–112, Deventer, The Netherlands, 2001. Kluwer, B.V.

8. M. Goodrich, A. Schwerin, and R. Tamassia. An efficient dynamic and distributed cryptographic accumulator. Technical report, Johns Hopkins Information, 2000.
9. M. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. Technical report, Johns Hopkins Information, 2000.
10. M. T. Goodrich, M. Shin, R. Tamassia, and W. H. Winsborough. Authenticated dictionaries for fresh attribute credentials. *iTrust 2003*, 2003.
11. Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, Inc., New York, NY, USA, 2000.
12. Paul C. Kocher. On certificate revocation and validation. In *FC '98: Proceedings of the Second International Conference on Financial Cryptography*, pages 172–177, London, UK, 1998. Springer-Verlag.
13. F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 121–132, New York, NY, USA, 2006. ACM Press.
14. C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.
15. A. Meier, R. Dippold, J. Mercerat, A. Muriset, J. Untersinger, R. Eckerlin, and F. Ferrara. Hierarchical to relational database migration. *IEEE Softw.*, 11(3):21–27, 1994.
16. R. C. Merkle. A certified digital signature. *Advances in Cryptology-Crypto'89*, 435:218–238, 1989.
17. Gerome Miklau and Dan Suciu. Implementing a tamper-evident database system. In *ASIAN: 10th Asian Computing Science Conference*, pages 28–48, 2005.
18. J. Ian Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *SODA '92: Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 367–375, Philadelphia, PA, USA, 1992.
19. Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Authentication and integrity in outsourced databases. *Trans. Storage*, 2(2):107–138, 2006.
20. Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. In *Proceedings 7th USENIX Security Symposium*, Jan 1998.
21. G. Nuckolls, C. Martel, and S. Stubblebine. Certifying data from multiple sources. In *EC '03: Proceedings of the 4th ACM conference on Electronic commerce*, pages 210–211, New York, NY, USA, 2003. ACM Press.
22. Caspio Bridge online database. `http://www.caspio.com`.
23. H. Pang, A. Jain, K. Ramamritham, and K. Tan. Verifying completeness of relational query results in data publishing. In *SIGMOD Conference*, pages 407–418, 2005.
24. H. Pang and K. Tan. Authenticating query results in edge computing. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 560, Washington, DC, USA, 2004. IEEE Computer Society.
25. Livebase project Blog on Web-based db. `http://livebase.blog.com/1142527/`.
26. William Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449, 1989.
27. R. Sion. Query execution assurance for outsourced databases. In *VLDB '05*, pages 601–612. VLDB Endowment, 2005.
28. R. Tamassia and N. Triandopoulos. On the cost of authenticated data structures. Technical report, Brown University, 2003.
29. Roberto Tamassia. Authenticated data structures. In *ESA 2003, LNCS*, Sep 2003.
30. online database Zoho Creator. `http://creator.zoho.com`.