Merkle Hash Tree based Techniques for Data Integrity of Outsourced Data

Muhammad Saqib Niaz
Dept. of Computer Science
Otto von Guericke University
Magdeburg, Germany
saqib@iti.cs.uni-magdeburg.de

Gunter Saake
Dept. of Computer Science
Otto von Guericke University
Magdeburg, Germany
gunter.saake@ovgu.de

ABSTRACT

One of the problems associated with outsourcing data to cloud service providers is the data integrity of outsourced data. In this paper we present data integrity techniques for the outsourced data. Data integrity encompasses the completeness, correctness and freshness of the data. This paper focuses on the Merkle Hash Tree based data integrity techniques. It also presents the techniques for storage and retrieval of Merkle Hash Tree based authentication data to and from cloud data service provider. Critical analysis of the Radix Path Identifiers, a technique for storage of Merkle Hash Trees in the databases, is presented in this paper.

General Terms

Cloud Databases, Security

Keywords

Database Security, Data Integrity, Outsourced Data

1. INTRODUCTION

Data outsourcing means to store your data on third party cloud data service providers. It is cheaper and easier to maintain the data on a cloud data service instead of maintaining it in data owner's own premises. Besides all the benefits, data outsourcing poses numerous security threats to the outsourced data. The list includes but is not limited to data integrity, access privacy and unauthorized access of data. The focus of this paper is data integrity that encompasses completeness, correctness and freshness.

There are three parties involved in these schemes. Data owner (DO), data clients and data service provider (DSP). A DSP provides all the data services and can be trusted with the server availability, timely backups, replications and disaster recovery. But the DSP cannot be trusted with the integrity of outsourced data. A DSP has unlimited access to the data to make it possible for the DSP to forge the data in anyway. It is assumed that the link between the DO and DSP and the links between clients and DSP are secure using some technique like SSL and the forgery of the data over these links can be easily detectable.

Following are some the features that need to be considered in designing data integrity techniques for outsourced data:

- Computation overhead for the DO
- Computation overhead of the DSP
- Storage overhead of DSP
- Computation overhead of the client
- Storage overhead of the client

The rest of the paper is organized as follows. A basic data integrity technique is presented in Section 2. The Merkle Hash Tree based data integrity scheme is presented in Section 3. Section 4 explains the storage and retrieval technique for Merkle Hash Tree based authentication data. Section 5 presents our analysis of the Radix Path Identifiers technique and the ongoing work on a new technique. Finally, the conclusions follow in Section 6.

2. BASIC TECHNIQUE

For the rest of the paper, we assume that there is a mechanism in place to securely share some data between DO and clients. This data could be the public key of the DO or some hash data. Only the DO can modify the data and the clients have read-only access of the data at the DSP.

The simplest data integrity technique could be to individually sign all the tuples in a data table and storing the signatures in a separate column in the same data table. Afterwards, on query of client, this signature can be sent to a client along with the tuple data. Clients can then check the integrity of the data by verifying the signature of the DO for the associated tuple. This scheme poses a huge computation overhead for the DO and the clients. Despite the computational overhead, it has a linear storage overhead as a distinct signature needs to be stored with each tuple. Still, attacks are possible on this scheme. DSP can delete some valid tuples from the data and the client would never be able to establish this fact. DSP can send an incomplete data set to the client and this forgery will also go undetected at client's end.

Data integrity schemes can be divided into two main categories, i.e., probabilistic and deterministic. Probabilistic approaches for data integrity have been suggested in [7, 11, 12]. The proposed techniques do not require any changes

27^{th} GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 26.05.2015 - 29.05.2015, Magdeburg, Germany.

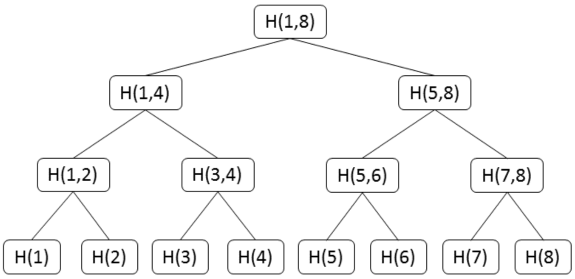

Figure 1: Merkle Hash Tree

at the DSP end, but sometimes the integrity results can be doubtful, as evident from the name.

The second category consists of deterministic approaches that generally base on Authenticated Data Structures (ADS) [6, 5, 10, 1, 2]. ADS based schemes will be the focus of the rest of the paper.

3. MERKLE HASH TREES

Authenticated Data Structures is a technique in which some kind of authentication data is stored on the DSP. On the client's query, a DSP returns the queried data along with some extra authentication data that is then used by the client to verify the authenticity of returned data.

Numerous techniques have been proposed that utilizes ADS for checking the data integrity. Signature aggregation based schemes have been proposed in [6, 5]. These approaches require to modify signatures of all the records, which renders it impractical considering the number of signatures [10]. The authenticated skip lists based approach has been proposed in [1]. A modified Merkle Hash Tree (MHT) based scheme has been proposed in [2] named super-efficient data integrity scheme. In this scheme the main MHT is divided into smaller MHTs and the root hashes of these sub-trees are signed. The purpose of the division in smaller MHTs is to avoid the unnecessary calculation up to the root of the main hash tree.

The Merkle Hash Tree based data integrity techniques for outsourced data are based on a signature scheme proposed by Merkle in [4]. This scheme eliminated the need of digital signatures for the data integrity purposes. MHT based data integrity techniques are based on two sub-components, i.e., Merkle's Signature Scheme and *B+* Trees.

3.1 Merkle's Signature Scheme

Merkle proposed a Signature Scheme based on a binary tree of hashes in [4]. Figure 1 shows a typical example of an MHT. Each leaf node holds the hash of a data block, e.g.,$H(1)$ holds the hash of the data block *1*. Internal nodes hold the hash of the concatenated hashes of their children e.g. $H(1,2) = H(H(1) \mid H(2))$ where '|' indicates concatenation. This scheme is based on the assumption that a safe/trusted way exists to share the root of the tree between the signer and the verifier. To verify the integrity of any data block, the whole tree of hashes does not need to be transmitted to the verifier. A signer transmits the hashes of only those nodes which are involved in the authentication path of the data block under consideration. For example, if the receiver needs to verify the integrity of data block *2* then only $H(1)$, $H(3,4)$ and $H(5,8)$ need to be transfered to the receiver. The receiver can calculate the $H(2)$ from data block *2*. $H(1,2)$ can then be calculated by using the received $H(1)$ and calculated $H(2)$. In the same way, $H(1,4)$ can be calculated and then $H(1,8)$. The receiver then can compare the calculated $H(1,8)$ with the already shared $H'(1,8)$ and if both the hashes match then the integrity of data block *2* is confirmed.

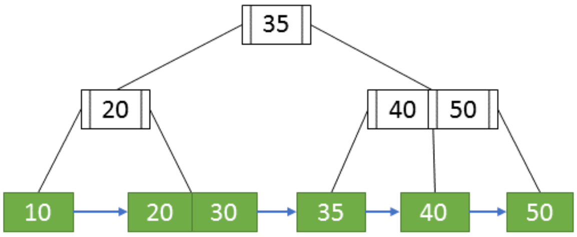

Figure 2: B+ Tree of order 3

Some important facts regarding Merkle's Signature Scheme are as follows:

- Security of this signature scheme depends on the security of the hash function.
- Only one hash needs to be maintained/shared securely.
- To authenticate any data block only $\log_2 n$ hashes need to be transfered, where n denotes total number of data blocks.
- In case of integrity checking of a continuous range of blocks, even less than $\log_2 n$ hashes need to be transfered.

3.2 B+ Trees

B+ trees are a special case of *B* trees as shown in Figure 2. They are n-ary trees. The root node can be a leaf node or it can be an internal node. Internal nodes only hold keys, they do not hold data. Data always stays in the leaf nodes. Leaf nodes are connected through pointers to form a kind of linked list. This linkage helps in sequential traversal of the data.

Let n be the order of a *B+* tree. The root node can hold *1* to *n-1* keys when root node is the only node in the tree. If root node is an internal node then it can have *2* to n child nodes. Internal nodes can have $\lceil n/2 \rceil$ to n child nodes. Leaf nodes can hold $\lceil n/2 \rceil$ to *n-1* keys [8].

3.3 Data Integrity based on Merkle Hash Tree

Data integrity schemes based on MHT have been designed by replacing binary trees with *B+* trees in original Merkle's Signature Scheme. The *B+* tree presented in Figure 2 is used with some modifications. Leaf nodes are linked with direct pointers. Besides keys, leaf nodes also hold the hashes of the data records pointed by corresponding keys. As an example, the leaf node *20, 30* also holds the hashes of the data records of the keys *20* and *30*. Internal nodes' pointers also hold the hashes of the concatenated hashes of its child nodes. Like right pointer of the internal node *20* holds the hash of the concatenated hashes of the data records pointed by keys *20* and *30* i.e. $H(20,30) = H(H(20) \mid H(30))$.

Security of Merkle Hash Tree based data integrity schemes depend on the security of the hash function as in original

Table 1: Employee Data Table

ID	Name	Salary
10	Alice	1000
20	Bob	2000
30	Cindy	3000
40	Dan	3000
50	Eva	2000
60	Felix	1000

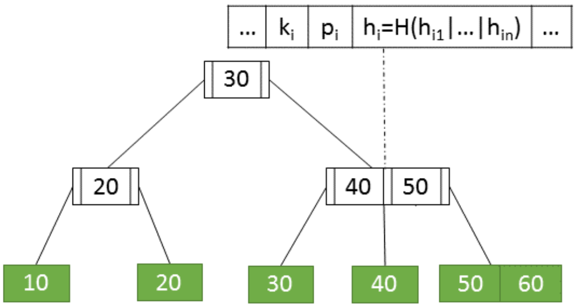

Figure 3: Merkle Hash Tree based on Table 1

Merkle's signature scheme. This scheme resolves the freshness issue of the query results, too. Each time a DO updates the data in the DSP, a new root hash is calculated based on the newly updated state of the data. By sharing the new root hash with the clients, freshness can be ensured.

3.4 Implementation Issues

A problem associated with the MHTs based data integrity schemes is the efficient storage and retrieval of MHTs in the DSP's database. Numerous approaches exist to store hierarchical or tree like data in a database, e.g., adjacency list, nested set, nested interval and closure table etc. Each approach has its pros and cons. For this specific problem of storing MHTs, a new technique named Radix Path Identifiers has been proposed in [10].

4. RADIX PATH IDENTIFIER

Consider a data table named *Employee* as shown in Table 1. A Merkle Hash Tree is created based on the data in *Employee* table as shown in Figure 3. Data is inserted in the MHT in an ascending order. Fanout of this *B+* tree is three that's why every node holds either one or two keys.

The basic idea is to assign numbers based on a radix to each pointer of the internal node and each key of the leaf node in order to uniquely identify them in a MHT. Radix could be any number equal to or greater than fanout of the MHT. We take *3* as the radix for the MHT created for *Employee* table.

Radix path identifiers have been added to the MHT shown in Figure 3 and the modified MHT is shown in 4. Radix Path Identifier of a pointer or key (in leaf node) depends upon its level in MHT and position in a node. Let l be the level of the MHT. The level of root node is *0* and the level of leaf nodes is the maximum. r_b is the radix base. f denotes the fanout of the MHT. i denotes the index of a pointer or a key in a node, ranging from *0* to *f*. Radix Path Identifier rpi can be computed using the following equation:

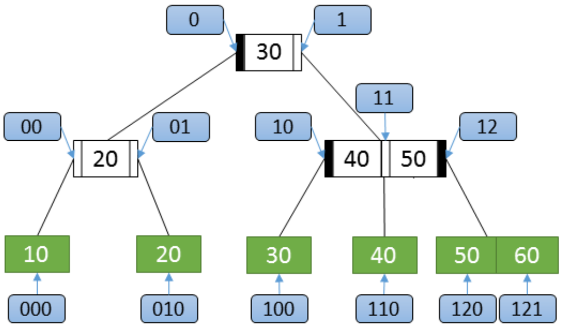

Figure 4: MHT with Radix Path Identifiers

$$rpi = \begin{cases} l & \text{if } l == 0 \\ rpi_{parent} * r_b + i & \text{if } l > 0 \end{cases} \tag{1}$$

For example, to calculate the *RPI* of the key *60* in the leaf node, the level of the key is determined. The level is not zero so the lower part of the equation is applicable and also note that all the calculations done are based on ternary number system. i in this case is *1* as *60* is the second key in the leaf node. *RPI* of the parent is *12* and the r_b is *3*. Multiplying rpi_{parent} with r_b gives *120* and adding i into it gives *121*, so the *RPI* of the key *60* in leaf node is *121*.

The proposed Radix Path Identifier scheme has several important properties:

1. *RPIs* are continuous in nodes, but not continuous among two consecutive nodes. For example, the base-3 numbers *10, 11, 12* are continuous but *110* and *120* are not continuous as shown in Figure 4.

2. From an *RPI*, we can easily find the *RPI* of its parent pointer based on the fact that rpi_{parent} equals to $\lfloor rpi/r_b \rfloor$.

3. From the *RPI* in a node, we can easily calculate the min and max *RPIs* in the node, which are $(\lfloor rpi/r_b \rfloor) * r_b$ and $(\lfloor rpi/r_b \rfloor) * r_b + (r_b - 1)$.

4. From an *RPI* in a node, we can easily compute the index i of the pointer or key in the node, which is rpi mod r_b.

4.1 MHT Storage in the Database

Two models have been suggested in [10] for storage of Radix Path Identifiers based MHTs in the database. The first method is to store the whole data in one authentication table called Single Authentication Table (SAT). The second method is to store each level of MHT in an individual table called Level Based Authentication Table (LBAT).

4.1.1 Single Authentication Table

In this technique, one table holds the entire authentication data as shown in Table 2. A tuple in this table represents either a pointer in an internal node or a key in a leaf node of the MHT. The authentication table has four columns named as *ID*, *RPI*, *Hash* and *Level*. The *ID* column in authentication table corresponds to the values in *ID* column in *Employee* table. In case of leaf nodes, each key corresponds to a tuple in the *Employee* table so the mapping is straight forward. However in case of internal nodes,

Table 2: Single Authentication Table (SAT)

ID	RPI	Hash	Level
-1	0	hash	2
30	1	hash	2
-1	0	hash	1
20	1	hash	1
-1	3	hash	1
40	4	hash	1
50	5	hash	1
10	0	hash	0
20	3	hash	0
30	9	hash	0
40	12	hash	0
50	15	hash	0
60	16	hash	0

the number of pointers is always *1* more than the number of keys. Because of that one pointer is stored in the table with *-1* as the *ID*. Rest of the pointers are saved with the *IDs* of the corresponding keys in the node. Considering the left most pointer in each internal node as an extra pointer, *-1* is assigned in the *ID* column of these pointers. The *Hash* column holds the hashes associated with the pointers in internal nodes and keys in leaf nodes.

The *RPI* holds the Radix Path Identifier of the pointer in internal node or key in leaf node. *RPIs* shown in Figure 4 are unique because these numbers are written in base *3* with their preceding zeros. However, for storing *RPIs* in authentication table, preceding zeros are ignored and the *RPIs* are converted into base *10* numbers. This results in mapping of different base *3* numbers to the same base *10* numbers. For example, *011* in third level and *11* in second level transform to the same base *10* number i.e. *4*. Consequently, the transformed *RPIs* are unique in a level but they can be repeated among different levels of the tree. In order to distinguish between the same *RPIs*, a *Level* column is added to the authentication table.

4.1.2 Level Based Authentication Table

In the LBAT technique, authentication data for each level of an MHT is stored in an individual table. Besides this, an extra table is created that holds the data about the authentication tables i.e. name of the table and its associated level. LBATs for the MHT shown in Figure 4 are shown in Table 3. As every LBAT table represents one level in the tree, there is no need to have a column *Level* in the authentication tables. Level *0* authentication table has exactly the same number of records as the *Employee* table so both tables can be merged to form one. *RPI* and *Hash* columns have been added at the end of *Employee* table to hold the authentication data for Level *0*.

4.1.3 Performance comparison between both schemes

Considering the table level locks during updates and inserts, it is easier/faster to update authentication data in LBAT than SAT. In the LBAT, as authentication data is stored along with the data record, it makes it straight forward to retrieve authentication data for the leaf level along with the required table data.

4.2 Authentication Data Extraction

Table 3: Level Based Authentication Tables (LBAT)

Emp_2 (Root)		
ID	**RPI**	**Hash**
-1	0	hash
30	1	hash

Emp_1		
ID	**RPI**	**Hash**
-1	0	hash
20	1	hash
-1	3	hash
40	4	hash
50	5	hash

Employee (Leaf Nodes)				
ID	**Name**	**Salary**	**RPI**	**Hash**
10	Alice	1000	0	hash
20	Bob	2000	3	hash
30	Cindy	3000	9	hash
40	Dan	3000	12	hash
50	Eva	2000	15	hash
60	Felix	1000	16	hash

Authentication data extracted from LBAT is used to compute the root hash of the MHT. For data extraction from LBAT table, four different ways have been presented i.e. Multi-Join, Single-Join, Zero-Join and Range-Condition in [9]. In all the following methods, data will be extracted from LBATs to verify the authenticity of data record with *ID 40*. All the required data involved in the authentication path of *ID 40* also needs to be extracted from the LBAT and the pointers involved in authentication path are marked with black color in Figure 4.

4.2.1 Multi-Join

In the Multi-Join approach, all the authentication data from respective LBATs are retrieved in a single query. Following SQL statement retrieves the authentication data of record with *ID 40*. In order to fetch all the authentication data in one query, multiple left outer joins have been used which introduces redundancy in the result.

```
select a0.RPI as RPI0, a0.hash as hash0,
a1.RPI as RPI1, a1.hash as hash1,
a2.RPI as RPI2, a2.hash as hash2
from Employee emp
left join Employee a0 on a0.RPI/3 = emp.RPI/3
left join Emp_1 a1 on a1.RPI/3 = emp.RPI/(3*3)
left join Emp_2 a2 on a2.RPI/3 = emp.RPI/(3*3*3)
where emp.ID = 40;
```

4.2.2 Single-Join

In the Single-Join approach, data from each authentication table is retrieved separately. As *ID* column does not exist in authentication tables that's why in each query, the authentication table has been joined with the *Employee* table on column *RPI*.

```
select e0.RPI, e0.hash
from Employee emp
left outer join Employee e0 on e0.RPI/3 = emp.RPI/3
where ID = 40;

select e1.RPI, e1.hash
from Employee emp
left outer join Emp_1 e1 on e1.RPI/3 = emp.RPI/(3*3)
```

```
where emp.ID = 40;

select e2.RPI, e2.hash
from Employee emp
left outer join Emp_2 e2 on e2.RPI/3 = emp.RPI/(3*3*3)
where emp.ID = 40;
```

4.2.3 *Zero-Join*

As evident from the name, no tables are joined for querying authentication data. Each table is queried individually. To query each table without any joins, the *RPI* of the record under consideration have to be extracted first and stored in some variable. Afterwards this stored *RPI* is used to extract authentication data from the LBATs.

```
declare @rpid as int;
select @rpid = RPI from Employee where ID = 40;

select RPI, hash from Employee where RPI/3=@rpid/3;

select RPI, hash from Emp_1 where RPI/3=@rpid/(3*3);

select RPI, hash from Emp_2 where RPI/3=@rpid/(3*3*3);
```

4.2.4 *Range-Condition*

Execution of queries presented in Zero-Join section scans *RPI* for each query. This scan can be replaced with index seek by creating an index on *RPI* and replacing the Zero-Join queries with Range-Conditions. Following queries show how to utilize an index created on *RPI* and efficiently query data from LBAT for authentication of data record *ID 40.*

```
declare @rpid as int;
select @rpid = RPI from Employee where ID = 40;

select RPI, hash from Employee
where RPI >= (@rpid/3)*3
and RPI < (@rpid/3)*3+3;

select RPI, hash from Emp_1
where RPI >= (@rpid/(3*3))*3
and RPI < (@rpid/(3*3))*3+3;

select RPI, hash from Emp_2
where RPI >= (@rpid/(3*3*3))*3
and RPI < (@rpid/(3*3*3))*3+3;
```

Each of the above given queries retrieve authentication data from a specific level of the MHT. The range condition specified in the above queries encompasses all the *RPIs* of the elements present in a node. For example, at level three, a node can have following *RPIs* i.e. *120*, *121* and *122.* In the *RPIs*, the last digit always stays less than the fanout of the tree that is why *3* is mentioned as the upper bound in the query.

4.3 Data Operations

Four data operations, i.e. select, insert, update and delete, will be discussed.

4.3.1 *Select*

Selection of a single record along with its associated authentication data has already been discussed in detail. Authentication of a continuous range of records is a bit different than the authentication of a single record. Let suppose, our range query returns a set of records from *ID 20* to *40*. User needs to find the two bounding values of the range under consideration. In this case, the two bounding values would be *10* and *50*. The user just needs to fetch the range of records from *ID 20* to *40* without their authentication data. In order to authenticate this range, only the authentication data of the two bounding values is required, i.e., *10* and *50.* By verifying the data integrity of the bounding values, user can be assured of the data integrity of the whole range.

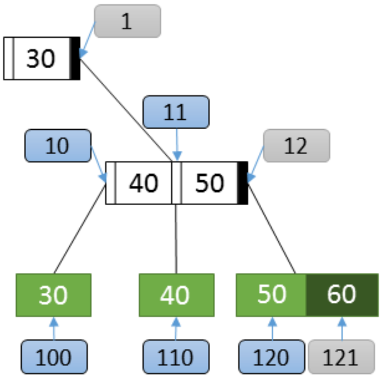

Figure 5: Updating a Record

4.3.2 *Update*

Update is a more complicated operation than Select. In updating a record, along with updating the data table, the user also needs to update the hashes in all the records involved in the authentication path of the updated record. For example, if user updates the data record with *ID 60*, hash of this record will be updated in the *Employee* table along with the data update. In addition user needs to update the hash data in the pointers marked as *12* and *1* as shown in Figure 5.

4.3.3 *Insert & Delete*

Insert & Delete are more complicated operations than updating a record. Insert & Delete could affect the associated MHT in three different ways. Simplest case could be that the insertion or deletion of a record effects only a single leaf i.e. a key can be added or deleted from a leaf node, in this case only the data involved in the authentication path of the affected leaf node needs to be updated. A little more complicated case could be the change in a subtree of the MHT, in this case all the authentication records of that subtree needs to be updated. In addition the authentication path of the root of the updated subtree also needs to be updated. The most complex case could be the addition or deletion of a new level in the MHT. In case of addition of a new level following needs to be done:

1. Addition of a new LBAT table in the database for the newly inserted level in the MHT.

2. Information regarding the new LBAT table needs to be inserted in the table that holds the data about the LBATs.

3. Update of data in all the LBATs.

5. ANALYSIS & ONGOING WORK

5.1 Analysis

In this section, we analyze the properties of Radix Path Identifiers and identify our next steps based on it. Merkle Hash Tree based data integrity technique guarantees all three aspects of data integrity, i.e., completeness, correctness and freshness [3]. And in doing so, it completely avoids digital signatures which pose a lot of computation overhead. Analysis of the Radix Path Identifier technique is as follows:

- It is assumed that only the DO can change the data.
- All the tests are performed on traditional DBMS i.e. SQL Server [10]. NoSQL databases may perform differently.
- Cached technique for Update results in the lowest overhead. Without caching of data at DO's end, the overhead can go up to 100%.
- Insert at the end of the table gives better result than the Insert at the beginning of the table. Both cases poses significant overhead. No results have been published for Delete.
- In order to modify the data, DO either has to download the whole copy of the table along with authentication data or has to keep the whole data cached at DO's premises.

5.2 Ongoing Work

We are currently starting to work on a Data Integrity Technique that is based on Merkle Hash Trees and Radix Path Identifiers. We want to achieve following goals in this new technique:

- Multiple users should be able to manipulate data.
- A user should not be required to keep a copy of the data in order to modify the data because keeping a copy of data eliminates the purpose of data outsourcing.
- Communication overhead should be minimized to a level near to performing operations in a normal database. For instance, to insert a row of data, a single insert statement should be sent to the database.
- The technique is being designed keeping in view the NoSQL database concepts, too.

6. CONCLUSIONS

Numerous techniques have been proposed in the literature to check the data integrity of the outsourced data to untrusted clouds. Focus of our paper was on Merkle Hash Tree based techniques. Despite a lot of research on MHT based techniques, still insertion and deletion pose a lot of communication and computation overhead. We have also discussed a technique named Radix Path Identifiers to store and retrieve authentication data in the DSP's database. We plan to design a new technique to eliminate the shortcomings of the current data integrity techniques. The main purpose of our technique is to avoid keeping or fetching the copy of whole data in order to run an insert or update statement. We are also going to simplify the process of inserting and deleting the data the way we are used to do in traditional DBMSs.

7. REFERENCES

[1] G. Di Battista and B. Palazzi. Authenticated relational tables and authenticated skip lists. In *Proceedings of the 21st Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 31–46, Berlin, Heidelberg, 2007. Springer-Verlag.

[2] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Super-efficient verification of dynamic outsourced databases. In *Proceedings of the 2008 The Cryptopgraphers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'08, pages 407–424, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 121–132, New York, NY, USA, 2006. ACM.

[4] R. C. Merkle. A certified digital signature. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, pages 218–238, 1989.

[5] M. Narasimha and G. Tsudik. Authentication of outsourced databases using signature aggregation and chaining. In *In International Conference on Database Systems for Advanced Applications (DASFAA*, pages 420–436. DASFAA, 2006.

[6] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 407–418, New York, NY, USA, 2005. ACM.

[7] R. Sion. Query execution assurance for outsourced databases. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 601–612. VLDB Endowment, 2005.

[8] A. L. Tharp. *File Organization and Processing.* John Wiley & Sons, Inc., New York, NY, USA, 1988.

[9] W. Wei and T. Yu. Practical integrity assurance for big data processing deployed over open cloud, 2013.

[10] W. Wei and T. Yu. Integrity assurance for outsourced databases without DBMS modification. In *Data and Applications Security and Privacy XXVIII - 28th Annual IFIP WG 11.3 Working Conference, DBSec 2014, Vienna, Austria, July 14-16, 2014. Proceedings*, pages 1–16, 2014.

[11] M. Xie, H. Wang, J. Yin, and X. Meng. Integrity auditing of outsourced data. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 782–793. VLDB Endowment, 2007.

[12] M. Xie, H. Wang, J. Yin, and X. Meng. Providing freshness guarantees for outsourced databases. In *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '08, pages 323–332, New York, NY, USA, 2008. ACM.