Integrity Assurance for Outsourced Databases without DBMS Modification

Wei Wei[1] and Ting Yu[1,2]

[1] North Carolina State University, Raleigh NC 27606, USA
wwei5@ncsu.edu,yu@csc.ncsu.edu
[2] Qatar Computing Research Institute, Tornado Tower, 18th floor, Doha, Qatar

Abstract. Database outsourcing has become increasingly popular as a cost-effective solution to provide database services to clients. Previous work proposed different approaches to ensuring data integrity, one of the most important security concerns in database outsourcing. However, to the best of our knowledge, existing approaches require modification of DBMSs to facilitate data authentication, which greatly hampers their adoption in practice. In this paper, we present the design and implementation of an efficient and practical integrity assurance scheme *without requiring any modification to the DBMS at the server side*. We develop novel schemes to serialize Merkle B-tree based authentication structures into a relational database that allows efficient data retrieval for integrity verification. We design efficient algorithms to accelerate query processing with integrity protection. We further build a proof-of-concept prototype and conduct extensive experiments to evaluate the performance overhead of the proposed schemes. The experimental results show that our scheme imposes a low overhead for queries and a reasonable overhead for updates while ensuring integrity of an outsourced database without special support from server-side DBMSs.

Keywords: Data Integrity, Database Outsourcing, Radix-Path Identifier.

1 Introduction

Database outsourcing has become increasingly popular as a cost-effective solution to provide database services to clients. In this model, a *data owner* (DO) outsources data to a third-party *database service provider* (DSP), which maintains the data in a DBMS and answers queries from *clients* on behalf of the data owner. However, it introduces one of the most important security concerns, data integrity. Usually, DSPs are not fully trusted by data owners. Thus, data owners have to protect the integrity of their own data when outsourcing data to DSPs. Specifically, when clients retrieve data from a DSP, they should be able to verify that the returned data is what should be returned for their requests on behalf of data owners, i.e., no data is maliciously modified by DSPs and DSPs return all data clients request.

There are many techniques proposed to address integrity issues, including correctness, completeness and freshness. These techniques can be divided into two categories. Approaches belonging to the first category are based on *authenticated data structures* (ADSs) such as Merkle hash tree (MHT) [4, 9, 12] and Signature Aggregation

V. Atluri and G. Pernul (Eds.): DBSec 2014, LNCS 8566, pp. 1–16, 2014.

[9, 14, 16, 18]. Existing ADS-based approaches require modifying a DBMS so that it can generate a *verification object* (VO) when executing a query and return the VO along with the actual result to clients, so that clients can verify the integrity of the query result. Such modification is usually costly and hard to be deployed in a third-party service provider, which hampers the adoption of database outsourcing [24]. The second category uses a probabilistic approach [20, 24, 25], which injects some fake data into outsourced databases. Although the probabilistic approach does not require the modification of DBMSs, its integrity guarantee is significantly weaker than that of those based on ADSs.

In this paper, we explore the feasibility of utilizing approaches of the first category to provide integrity assurance *without requiring any modification of DBMSs*. In existing approaches, DBMSs are modified to be ADS-aware. That is, they are enhanced with special modules that efficiently manage ADSs and facilitate the generation of VOs. Unfortunately, it is often hard to convince database service providers to make such modifications to their DBMSs. In fact, up to today, to the best of our knowledge, no existing cloud database services support integrity checking [19]. Thus, for clients who care about query integrity, it is desirable to have integrity assurance techniques over "vanilla" DBMSs (i.e., without any special features for outsourced data integrity). The general approach is straightforward: the data owner would have to store authenticated data structures along with their own data in relations, and retrieve appropriate integrity verification data besides issuing queries. And all these have to be done through the generic query interface (usually SQL) of the DBMS. Though the basic idea is simple, the challenge is to make it practical: we need to design appropriate schemes to convert ADSs into relations and form efficient queries to retrieve and update authentication information, *without imposing significant overhead*.

In this paper, we present an efficient and practical scheme based on Merkle B-tree, which provides strong integrity assurance without requiring special support from database service providers. Our scheme serializes a Merkle B-tree based ADS into relations in a way, such that the data in the ADS can be retrieved and updated directly and efficiently using existing functionality provided by DBMSs, that is, SQL statements. Our major contributions are summarized as follows:

- We propose a novel scheme called Radix-Path Identifier to identify each piece of authentication data in a Merkle B-tree based ADS so that the MBT can be serialized into and de-serialized from a database, and design an efficient and practical mechanism to store all authentication data of a Merkle B-tree in a database, where the authentication data in the MBT can be retrieved and updated efficiently.
- We explore the efficiency of different methods such as Multi-Join, Single-Join, Zero-Join and Range-Condition, to retrieve authentication data from a serialized MBT stored in a database, create appropriate indexes to accelerate the retrieval of authentication data, and optimize the update process for authentication data.
- We build a proof-of-concept prototype and conduct extensive experiments to evaluate the performance overhead and efficiency of our proposed scheme. The results show that our scheme imposes a low overhead for queries and a reasonable overhead for updates while providing integrity assurance. Note that although we describe our

scheme based on relational DBMSs, it is not hard to see that our scheme can also be applied to Non-SQL databases such as Bigtable [3], Hbase [1].

We note that many modern relational databases also have built-in support for XML. One seemingly promising approach is to represent Merkle B-tree as XML, store the XML representation into the DBMSs, and utilize their built-in XML support to retrieve authentication data for integrity verification. However, as can be seen from the performance result presented in Section 6, the XML-based solutions do not provide a good performance compared with our scheme, which is mainly because the XML features are not targeting at providing efficient operations of MHT-based integrity verification.

The rest of the paper is organized as follows. We discuss related work in Section 2. In Section 3, we describe the data outsourcing model we target, state assumptions, attack models and our goals. Section 4 explains the major design of our scheme in details, and section 5 illustrate how our scheme provides integrity assurance for different data operations such as *select, update, insert and delete*. Section 6 discusses the experimental results. Finally, the paper concludes in Section 7.

2 Related Work

Researchers have investigated on data integrity issues for years in the area of database outsourcing [5, 8, 9, 13, 16, 17, 20, 24, 25]. Pang et. al. [16] proposed a signature aggregation based scheme that enables a user to verify the completeness of a query result by assuming an order of the records according to one attribute. Devanbu et. al. [5] uses Merkle hash tree based methods to verify the completeness of query results. But they do not consider the freshness aspect of data integrity. Xie et al. [24] proposed a probabilistic approach by inserting a small amount of fake records into outsourced databases so that integrity can be effectively audited by analyzing the inserted records in the query results, which only protects integrity probabilistically.

Li et. al. [9] first brought forward the freshness issue as an aspect of data integrity. It verifies if data updates are correctly executed by DSPs so that queries will be executed over the up-to-date data instead of old data. Xie et al. [25] analyzed different approaches to ensuring query freshness. The aggregated signature based approaches [14, 16] require to modify signatures of all the records, which renders it impractical considering the number of signatures.

Miklau et. al. [11] designed a scheme based on interval tree to guarantee data integrity when interacting with a vulnerable or untrusted database server. However, several disadvantages are mentioned in Di's work [6], which dealt with a similar issue based on authenticated skip list [7]. Di Battista's work [6] does not explain clearly how authentication data is retrieved. It claims that only one query is required for integrity verification while it also mentions that multiple queries are necessary to retrieve all authentication data. Palazzi et. al. [15] proposed approaches to support range queries based on multiple attributes with integrity guarantee, complementary to our work.

Compared with previous work, our scheme is able to provide integrity assurance for database outsourcing, including all three aspects: correctness, completeness and freshness. More importantly, one significant advantage of our scheme is that existing

approaches need to modify the implementation of DBMSs in order to maintain an appropriate authenticated data structure and generate VOs. This requirement often make these approaches hard to be deployed in real-world applications [24]. Our work provides a strong integrity guarantee (instead of probabilistic guarantee [24]) without requiring DBMSs to be modified to perform any special function beyond query processing.

3 System Model

3.1 Database Outsourcing Model

Figure 1 shows our database outsourcing model with integrity protection. There are three types of entities: *data owner*, *database service provider* (DSP) and *clients*. A data owner uploads a database with data and authentication data to a DSP, which provides database functionality on behalf of the data owner. Clients send to the DSP queries to retrieve data and a *verification object* (VO).

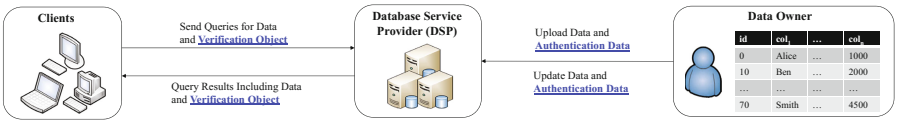

Fig. 1. Non-Intrusive Database Outsourcing Model

In our outsourcing model, we assume that the DSP is oblivious to integrity protection. In fact, the DSP does not even know where and how to store authentication data and when and how to return authentication data to clients for integrity verification. Everything related to data integrity verification is done at the client side through an integrity-aware DBMS driver and is transparent to applications running in the client side, and data and authentication data updates are done by the data owner. In this way, data owners can provide integrity assurance for their outsourced databases without any special support from DSPs. Therefore, the adoption of database outsourcing with integrity assurance is completely decided by data owners themselves.

3.2 Assumptions and Attack Models

First, we assume that data owners and clients do not fully trust the services provided by DSPs. Second, since our scheme relies on digital signatures to provide integrity protection, we assume that the data owner has a pair of private and public keys for signature generation and verification. The public key is known to all clients. Moreover, like in many existing work [8, 9, 14, 18], we assume that the data owner is the only entity who can update its data. In addition, we assume that communications between DSPs and clients are through a secure channel (e.g., through SSL). Thus, DSPs and clients can detect any tampered communication.

Regarding attack models, we focus ourselves on the malicious behavior from a DSP since it is the only untrusted party in our target database outsourcing model. We do not have any assumption about what kind of attacks or malicious behavior a DSP may take. A DSP can behave arbitrarily to compromise data integrity. Typical malicious behaviors include, but not limited to, modifying a data owner's data without the data owner's authorization, returning partial data queried to clients and reporting non-existence of data even if data does exist. Further, it could return stale data to clients instead of executing queries over the latest data updated by the data owner [23].

3.3 Security Goals

We aim at providing integrity protection for all three aspects in data integrity: correctness, completeness, and freshness. First, the correctness checks if all records returned in a query result come from the original data set without being maliciously modified, which is usually achieved using digital signatures that authenticate the authenticity of records. Second, the completeness checks if all records satisfying conditions in a query are completely returned to clients. Third, the freshness checks if all records in a query result are the up-to-date data instead of some stale data.

Regarding freshness, we propose mechanisms for data owners to efficiently compute signatures of updated data and guarantee the correctness of the signatures, which is the key to provide freshness guarantee. The security guarantee we provide is as strong as Merkle B-tree (MBT) [9]. In the paper, we do not focus on how the latest signatures are propagated to clients for integrity verification purpose, as it can be easily achieved by applying existing techniques [10, 25].

4 System Design

4.1 Running Example

We first present an example that will be referred to throughout the paper to illustrate our schemes. Without loss of generality, we assume that a data owner has a database with a table called "data", as shown in the left side of Figure 2. The table has several columns. The *id* column is a unique key or indexed. Besides, there are n columns $\{col_1, ..., col_n\}$ containing arbitrary data.

4.2 Authenticated Data Structure

Regarding Authenticated Data Structure (ADS), there are two options: signature aggregation based ADS and Merkle hash tree based ADS. We observe that there are several disadvantages of developing a scheme based on signature aggregation based ADS. First, to minimize communication cost, signature aggregation operation needs to be done dynamically in DBMSs, which unfortunately is not supported. Moreover, it is unknown how to efficiently guarantee freshness using signature aggregation based approaches [9]. Additionally, techniques based on signature aggregation incur significant computation cost in client side and much larger storage cost in the server side.

Thus, we choose to adapt MHT-based ADS, in particular, Merkle B-tree (MBT) [9]. MHT-based ADS can not only guarantee correctness and completeness, but also provide efficient freshness protection since only one root hash needs to be maintained correctly. Figure 2 shows a Merkle B-tree created based on the table introduced in Section 4.1. The values in the *id* column are used as keys in the MBT. A hash h_i is associated with a pointer in an internal node or a record in a leaf node. For simplicity, the hashes associated with pointers and records in nodes of the MBT are not shown in the figure. The hash of a record in a leaf node is the hash value of the data record in the data table. The hash associated with a pointer in an internal node is the hash of concatenating all hashes in the node pointed by the pointer.

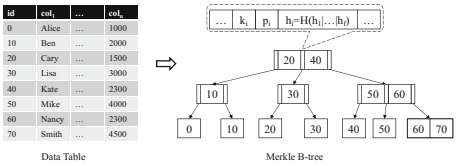

Fig. 2. Data table to Merkle B-tree

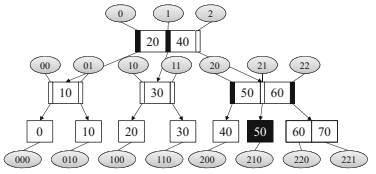

Fig. 3. Radix-path identifier

4.3 Identify Authentication Data

The first thing is to identify pointers in internal nodes and records in leaf nodes of a MBT since each pointer or record is associated with a piece of authentication data, that is, a hash. And also we need to capture their parent-child and sibling relationships. Besides, we need to preserve the ordering of pointers or records in a node of a MBT.

Existing Approaches. There are a few widely-used models such as adjacency list, path enumeration, nested set and closure table to store tree-like hierarchical data into a database [2, 21]. Each of them has its own advantages and disadvantages. For example, with an adjacency list, it is easy to find the parent of a pointer or a record since it captures the parent-child relationship directly. But to find its ancestor, we have to traverse the parent-child relationship step by step, which could make the process of retrieving VO inefficient. The path enumeration model uses a string to store the path of each pointer or record, which is used to track the parent-child relationship. Unlike the adjacency list model, it is easy to find an ancestor of a pointer or record in a node. But same as the adjacency list, path enumeration does not capture the order of pointers or records in a node.

Radix-Path Identifier. To address the disadvantages of existing approaches, we propose a novel and efficient scheme called *Radix-Path Identifier*. The basic idea is to use numbers based on a certain radix to identify each pointer or record in a MBT. Figure 3 shows all identifiers as base-4 numbers for pointers or records in the tree based on a radix equal to 4. Given a MBT, the *Radix-Path Identifier* of a pointer or record depends on its level and position in the MBT. To illustrate this scheme, suppose that the fanout of a MBT is f. The radix base r_b could be any number larger than or equal to f. l denotes

the level where a node resides in the MBT. The level of the root node is 0. i denotes the index of a pointer or record in a node, ranging from 0 to f. The *Radix-Path Identifier* $rpid$ of a pointer or record can be computed using the following equation:

$$rpid = \begin{cases} i & \text{if } l == 0, \\ rpid_{parent} * r_b + i & \text{if } l > 0. \end{cases} \quad (1)$$

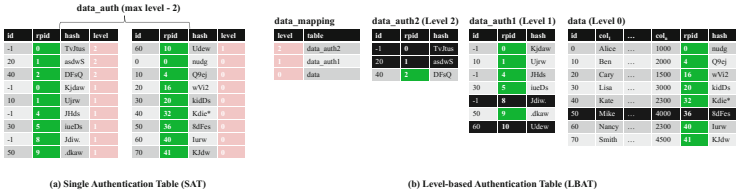

Fig. 4. Authentication Data Organization

Note that $rpid_{parent}$ is the *Radix-Path Identifier* of its parent pointer in the tree. Equation 1 captures not only the ordering among pointers or records in one node, but also the parent-child and sibling relationships among nodes. The identifier of each pointer or record in the root node is i. With identifiers in the root node, we can use the second part of Equation 1 to compute identifiers of pointers or records in their child nodes. In this way, all identifiers can be computed starting from the root node to the leaf nodes. The proposed *Radix-Path Identifier* scheme has several important properties: 1) Identifiers of pointers or records in a node are continuous, but not continuous between that of those in two sibling nodes. For example, the base-4 numbers $20, 21, 22$ are continuous and $200, 210$ are not continuous, shown in Figure 3; 2) From an identifier of a pointer or record in a node, we can easily find the identifier of its parent pointer based on the fact that $rpid_{parent}$ equals to $\lfloor rpid/r_b \rfloor$; 3) From the identifier of a pointer or record in a node, we can easily calculate the min and max identifiers in the node, which are $(\lfloor rpid/r_b \rfloor) * r_b$ and $(\lfloor rpid/r_b \rfloor) * r_b + (r_b - 1)$; 4) From an identifier of a pointer or record in a node, we can easily compute the index i of the pointer or record in the node, which is $rpid$ mod r_b. These properties will be utilized for efficient VO retrieval and authentication data updates.

4.4 Store Authentication Data

Once we identify each pointer or record in nodes of a MBT, the next step is how we can store the authentication data associated with them into a database. In the following, we propose two different designs - Single Authentication Table (SAT) and Level-based Authentication Table (LBAT), and discuss their advantages and disadvantages.

SAT: Single Authentication Table. A straightforward way is to store all authentication data as data records called *Authentication Data Record* (ADR) into one table in a database, where its corresponding data table is stored. Figure 4(a) shows all authentication data records in a single table for the data table described in the running example.

The name of the authentication table adds a suffix "_auth" to the original table name "data". The authentication table has 4 columns: *id*, *rpid*, *hash* and *level*. *id* column stores values from *id* column of the data table, which are keys in the B+ tree except "-1". Note that since the number of keys is less than the number of pointers in the internal nodes in a B+ tree node, we use "-1" as the *id* for the left-most pointers in the internal nodes. *rpid* records identifiers for pointers or records in the B+ tree. *hash* column stores the hash values of pointers or records in the B+ tree, which is essential for integrity verification. *level* stores values indicating the level of a pointer or record in the B+ tree. The *level* value is necessary for searching the *rpid* for a data record given an *id* of the data record because the *rpid* values could be the same in different levels. The level of a leaf node is 0, and the level of the root node is the maximum level.

Although SAT is simple and straightforward, it has several disadvantages, which makes it an inefficient scheme. First, updates could be inefficient since one data record update usually requires updating ADRs in different levels. With table level locks, it is not allowed to concurrently execute ADR updates since all ADR updates have to be executed over the only one table. Although concurrent updates can be enabled with row level locks, it may consume much more database server resources, which may not be desired. Second, it may require join queries to find the *rpid* of a data record since the data table is separated from the authentication data table. Third, updates to a data record and its ADR in the leaf level cannot be merged into a single query to improve the performance since they go to different tables.

LBAT: Level-Based Authentication Table. To resolve the above issues, we propose a Level-based Authentication Table (LBAT). In this scheme, instead of storing all ADRs into one table, we store ADRs in different levels to different tables. We create one table per level for an MBT except the leaf level (for reasons given below) along with a mapping table to indicate which table corresponds to which level. For nodes in the leaf level of the MBT, since each data record corresponds to an ADR in leaf nodes, we extend the data table by adding two columns - *rpid* and *hash* to store ADRs instead of creating a new table, which reduces the redundancy of *id* values as well as the update cost to some extent. Figure 4(b) shows all tables created or extended to store ADRs and the mapping table for the data table described in the running example. Tables for different levels have different number of records. For the root level, it may only contain a few records. Also, the number of records in the mapping table is equal to the number of levels in the MBT. We name those tables by adding a suffix such as "_mapping", "_auth0", etc, based on table types and levels.

The proposed LBAT scheme presents several advantages. First, since ADRs in different levels are stored in different authentication tables, it makes concurrent updates possible with table level lock, which also allows to design efficient concurrent update mechanisms. Second, since we store ADRs in the leaf level along with data, it makes it straightforward to retrieve the *rpid* of a data record. Third, due to the same advantage, it is easy to merge updates for a data records and its ADR in the leaf level for performance improvement.

4.5 Extract Authentication Data

To extract the ADRs for the record based on LBAT, we make the best use of the properties of our *Radix-Path Identifier*. Once we receive all related ADRs, we can compute the root hash since we can infer the tree structure from the *rpid* values, which conveniently captures the relationship among pointers, records and nodes in the MBT.

Since the DSP is only assumed to provide standard DBMS functionalities, all the above operations have to be realized by SQL queries issued by the client. We explore four different ways - Multi-Join, Single-Join, Zero-Join and Range-Condition, to find the authentication data records based on LBAT. We use specific examples to show how they work. All examples are based on the data presented in the running example. Suppose that we want to verify the integrity of the data record with the *id* 50. The ADRs needs to be returned shown as the black parts in Figure 3, which is also highlighted with a black background in Figure 4(b). Multi-join uses one query joining all related tables to retrieve authentication data records, which returns a lot of redundant data, and Single-Join uses multiple queries, each of which joins two tables to avoid returning redundant data. Due to space limit, we only illustrate Zero-Join and Range-Condition in details below. More details about Multi-join and Single-Join can be found at [22].

Zero-Join. In this scheme, we aim at minimizing the redundant data returned in Multi-Join and avoid multiple join queries in Single-Join. In fact, what we actually need is the *rpid* of the record 50. If we know its *rpid*, we can eliminate the "join" completely from the SQL statements. The following shows the SQL statements we use to retrieve the authentication data without joining any table.

```
-- find the rpid of the data record with the id 50
declare @rowrpid AS int;
set @rowrpid=(select top 1 rpid from data where id=50);
-- level 2, 1, 0 (from root level to leaf level)
select rpid,hash from data where rpid/4=@rowrpid/(4);
select rpid,hash from data_auth1 where rpid/4=@rowrpid/(4*4);
select rpid,hash from data_auth2 where rpid/4=@rowrpid/(4*4*4);
```

Compared with Single-Join, the major difference is that we declare a "rowrpid" variable to store the *rpid* of the record, which is retrieved from the first query. After that, we use the "rowrpid" for other queries to retrieve the authentication data for nodes in different levels. Although it needs to execute one more query, it eliminates the "join" clause completely.

Range-Condition. We observe that the execution of the above queries does not utilize the indexes created on the *rpid* field in the authentication tables. Instead of doing an index seek, each of them actually does an index scan, which is inefficient and incurs a high computation cost in the server side. To utilize indexes, we propose a new method called Range-Condition to retrieve authentication data for records. The following shows the SQL statements we use to retrieve the authentication data for the record 50 using Range-Condition.

```
-- find the rpid of the data record with the id 50
declare @rowrpid AS int;
set @rowrpid=(select top 1 rpid from data where id=50);
-- level 2, 1, 0 (from leaf level to root level)
select rpid,hash from data
where rpid>=(@rowrpid/(4))*4 and rpid<(@rowrpid/(4))*4+4;
```

```
select rpid,hash from data_auth1
where rpid>=(@rowrpid/(4*4))*4 and rpid<(@rowrpid/(4*4))*4+4;
select rpid,hash from data_auth2
where rpid>=(@rowrpid/(4*4*4))*4 and rpid<(@rowrpid/(4*4*4))*4+4;
```

As can be seen from the figure, the major difference from Zero-Join is the *where* condition. Instead of using equality, the Range-Condition uses a range query selection based on the *rpid* column. The range query retrieves the same set of ADRs as the equality condition used in Zero-Join. Thus, they both return the same set of authentication data records, and Single-Join does that too. However, with the range query on the *rpid* field, it can utilize indexes built on the *rpid* column, which minimizes the computation cost in the server side.

5 Data Operations

In this section, we illustrate the details of handling basic queries such as *select*, *update*, *insert* and *delete* with integrity protection efficiently based on our design using the running example. Without loss of generality, we assume that clients always have the latest root hash of the table for integrity verification, and we focus on how to retrieve authentication data from DSPs. Due to space limit, we do not discuss *insert* and *delete*. Please refer to [22] for implementation details and experimental results.

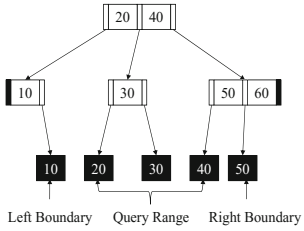

Fig. 5. Range Query with Integrity Protection

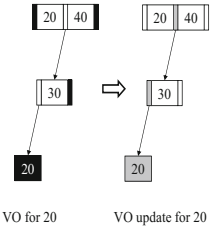

Fig. 6. Update with Integrity Protection

5.1 Select

As discussed in Section 4.5, we can retrieve authentication data for a *Unique Select* query, which returns only one data record based on a unique key selection. Thus, we focus on how to handle a *Range Select* query with integrity protection, which retrieves records within a range.

The verification process for *Range Select* queries is different from *Unique Select* queries. First, we need to find the two boundary keys for a range query. For example, for a range query with a range from 15 to 45, we need to identify its two boundaries, which are 10 and 50 in this case. Although DBMSs do not provide a function to return the boundary records directly, we can use the following two queries to figure out what the left and right boundaries are for a query range:

```
select top 1 id from data where id < 15 order by id desc
select top 1 id from data where id > 45 order by id asc
```

Then, to retrieve the authentication data for the range query, we only need to retrieve the authentication data for both boundaries, which is similar to the way we use to retrieve authentication data object for a data record since the authentication data for records within the range are not necessary and they will be computed by using the returned records. Figure 5 shows the authentication data records and the data records that need to be retrieved for the range query from 15 to 45.

To execute the range query with integrity protection, we need to rewrite the range query by adding SQL statements of retrieving authentication data records. Then, we execute all SQL statements in one database transaction. Once the result with authentication data is returned, we verify the integrity of the query result using the authentication data. If the verification succeeds, the data result is returned to the client as before; otherwise, an integrity violation exception could be thrown to warn the client of the integrity verification failure.

The overhead to provide data integrity for range queries consists of both computation and communication cost. The computation cost in the client side includes two parts: rewriting range query and verifying data integrity. The computation cost in the server side is the execution of additional queries for authentication data retrieval. The communication cost between them includes the text data of additional queries and the authentication data returned along with the data result.

This process can also handle *Unique Select* queries. However, it requires to retrieve authentication data for both left boundary and right boundary, which may not be necessary. If the key does not exist, we have to resort to the process of handling range queries, where we can check left boundary and right boundary to make sure the record with the key does not exist.

5.2 Update

Single Record Update. When a data record is updated, we need to update its authentication data (mainly hash values) accordingly. For updating a record, we assume that the record to be updated already exists in the client side and the VO for the updated record is cached in the client too. Otherwise, we retrieve the data record and its VO first, then update it and its authentication data.

Figure 6 shows the VO in black for the record 20 in the left side and the hash values in gray to be updated once the record is updated. Each data update requires an update on all authentication data tables. It means if the MBT tree's height is h, then the total Number of update queries is $h + 1$. In this case, we need to actually update 4 records. One of them is to update the data record and three of them is to update the authentication data records. The generation of update queries for authentication data is simple since we know the *rpid* of the data record to be updated, and then we can easily compute its parent *rpid* and generate update queries.

Since the authentication data table for the leaf level of a MBT is combined with the data table, we can combine two update queries into one to improve the performance. Thus, in this case we only need 3 update queries instead of 4. All update queries are

executed within one transaction. So, consistency of data records and authentication data is guaranteed by the ACID properties of DBMSs, and data integrity is also guaranteed since the verification and the root hash update are done directly by the data owner.

Batch Update and Optimization. Suppose that we want to update x records at one time. As the number of records to be updated increases, the total number of update queries we need to generate to update both data and authentication data increases linearly. In this case, the total number of update queries is $x * h$. We observe from those update queries that several update queries try to update the same authentication data record again and again due to the hierarchical structure of a B+ tree. We also notice that each update SQL statement only updates the same authentication record in one table. In fact, we just need to get the latest hash of the authentication data record, and do one update. To do that, we need to track all update queries for each table, find the set of queries to update one authentication data record in an authentication table, and remove all of them except the latest one. In this way, the number of necessary update queries could be much less than the number of update queries we generate before. The process, called *MergeUpdate*, improves the performance of batch update to a great extent.

6 Experimental Evaluation

System Implementation. We have implemented the Merkle B-tree and the query rewrite algorithms for clients, which is the core of generating select, update and insert SQL statements to operate authentication data. We also built a tool to create authentication tables and generate authentication data based on a data table in a database. Data owners can run this tool on all data tables in a database before outsourcing the database to a DSP. Once the authentication data is created for the database, they can upload the database to the DSP. We have also implemented all four different ways - *MultiJoin*, *SingleJoin*, *ZeroJoin* and *RangeCondition* - to retrieve authentication data for performance overhead evaluation. Our implementation is based on .NET and SQL Server 2008. In addition, we implemented two XML-based schemes: OPEN-XML and DT-XML, which utilize built-in XML functionality of SQL Server, for efficiency analysis and comparison. In both OPEN-XML and DT-XML schemes, we use a hierarchical XML structure to represent the authentication data of a Merkle B-tree and store the XML string into a database. The OPEN-XML scheme uses OPENXML function provided in SQL Server to retrieve VO data from the XML string, and the DT-XML uses XPath and nodes() methods to retrieve VO data from an indexed XML data field, where the XML string is stored.

Experiment Setup. We use a synthetic database that consists of one table with $100,000$ records. Each record contains multiple columns, a primary key *id*, and is about 1KB long. For simplicity, we assume that an authenticated index is built on *id* column. We upload the database with authentication data to a third-party cloud service provider, which deploys the SQL Server 2008 R2 as a database service, and run experiments from a client through a home network with 30Mbps download and 4Mbps upload. To evaluate the performance overhead of integrity verification and the efficiency of the proposed mechanisms, we design a set of experiments using the synthetic database.

6.1 Performance Analysis

VO Size. Figure 7 shows how the VO size changes as the fanout of a MBT changes for *Unique Select* and *Range Select*. The results clearly show that as the fanout increases, the VO size increases, and the VO size of *Range Select* is almost twice of that of *Unique Select* since the VO of *Range Select* includes the VO of two boundaries of the range. Note that for *Range Select*, its VO size almost stays the same no matter how many records are returned in a *Range Select*.

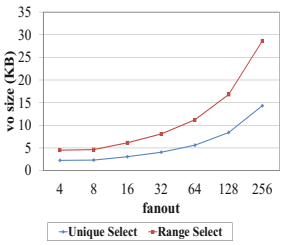

Fig. 7. VO size vs fanout

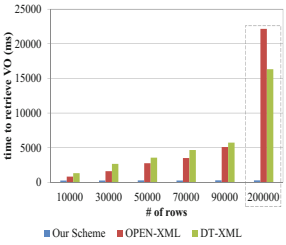

Fig. 8. VO retrieval time

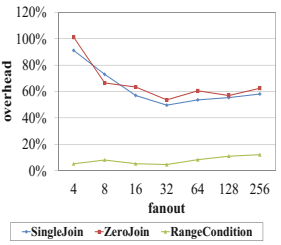

Fig. 9. Unique select overhead

VO Retrieval. Figure 8 shows the time to retrieve a VO for our scheme using RangeCondition and two XML-based schemes when the number of rows in the data set changes. As can be seen from the figure, when the data size is small, three schemes show a similar time to retrieve the VO. However, as the data size increases, two XML-based schemes show linear increases in terms of the VO retrieval time. When the data size goes up to $200,000$ records, the XML-based schemes take more than 15 seconds to retrieve a VO for one single record. In this case, our scheme is about 100 times faster than the two XML-based schemes. The result indicates that a well-design scheme could be much more efficient than a scheme using built-in XML functionality in DBMSs.

Unique Select. We conduct experiments to see how different fanouts of a MBT and different methods of retrieving VO could affect the performance of *Unique Select* queries, where we vary the fanout of a MBT and compare the performance overhead caused by different VO retrieval methods, shown in Figure 9. The results show that the overhead of SingleJoin and ZeroJoin is much higher than that of RangeCondition. When the fanout is 32, the overhead of SingJoin or ZeroJoin is about 50%, but the overhead of RangeCondition is 4.6%. The communication cost for the three different methods is almost same, and the major performance difference is caused by the computation cost in the server side. As we can see from this figure, when the fanout increases from 4 to 32, the overhead of both SingleJoin and ZeroJoin drops, and when the fanout is larger than 32, their overhead increases. It is because in general the VO size increases and the number of queries to be executed to retrieve authentication data decreases as the fanout increases, and when the fanout is less than 32 the computation cost dominates the overhead and when the fanout is larger than 32 the communication cost dominates the overhead. Based on the current experiment environment, the 32 fantout shows a better performance compared with other fanouts. In the following experiments we use 32 as the default fanout unless specified otherwise.

Range Select. We also run experiments to explore how the overhead changes when the number of records retrieved increases. Figure 10 shows the response time of retrieving different number of records in range queries, where NoVeri denotes range queries without integrity verification support, ZeroJoin and RangeCondition denote rang queries with integrity verification but using VO retrieval method ZeroJoin and RangeCondition respectively. The results show two points: 1) the RangeCondition is much better than ZeroJoin when the number of rows to be retrieved is small, which is because the computation cost dominates the overhead caused by different VO retrieval methods; 2) once the number of records to be retrieved is larger than a certain number, the response time of all three is almost the same. In our algorithm, the overhead caused by different VO retrieval methods does not change as the number of retrieved records increases. Thus, as the number of retrieved records increases, the overhead becomes relatively smaller and smaller. We also conduct experiments to show how the overhead changes as the database size increases, where we run range queries to retrieve 512 rows from databases with different number of data records. As shown in Figure 11, the overhead is about 3% even if the number of data records goes up to 1.6 million.

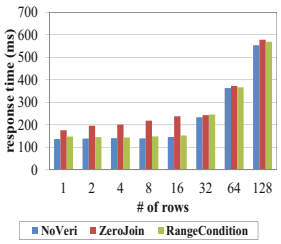

Fig. 10. Range select response time

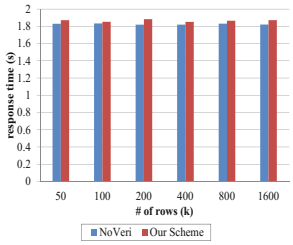

Fig. 11. Scalability of Range Select

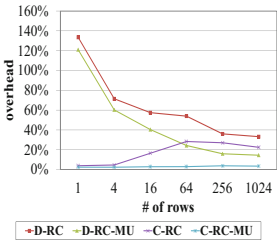

Fig. 12. Direct and cached update overhead comparison

Update. We evaluate the performance overhead caused by two different update cases - Direct Update and Cached Update. For Direct Update, we first retrieve the data to be updated and verify its data integrity, and then we generate update queries for both data and authentication data and send them to the sever for execution. For Cached Update, we assume that the data to be updated is already cached in the memory, we just need to generate update queries and send them to the server for execution. Figure 12 shows the overhead versus the number of rows to be updated. In the figure, 'D' denotes Direct Update, C denotes Cached Update, "RC" denotes RangeCondition, and "MU" denotes MergeUpdate, which indicates if a MergeUpdate process is used to reduce the number of SQL statements generated for updating authentication data records. The results show that when we directly update only a few records with integrity protection, the overhead could go above 100%, but if we update cached records, the overhead is about 2.5%. In this case, the additional round-trip time in Direct Update dominates the response time of the whole update process. As the number of updated rows increases, the overhead percentage of Direct Update decreases because the response time is dominated by the

update time in the server side. The major overhead for Cached Update comes from the execution of update statements to update authentication data in the server side. The results also show that the performance of C-RC-MU is comparable to the performance of NoVeri without integrity protection, but without the MergeUpdate optimization, the overhead of C-RC ranges from 3% to 30% shown in the figure.

7 Conclusion

In the paper, we present an efficient and practical Merkle B-tree based scheme that provides integrity assurance without modifying the implementation of existing DBMSs. We have proposed a novel approach called Radix-Path Identifier, which makes it possible to serializes a Merkle B-tree into a database while enabling highly efficient authentication data retrieval and updates. We have explored the efficiency of different methods such as MultiJoin, SingleJoin, ZeroJoin and RangeCondition, to retrieve authentication data from a serialized MBT stored in a database, implemented a proof-of-concept prototype, and conducted extensive experimental evaluation. Our experimental results show that our scheme imposes a small overhead for *Select*, *Update* and *Append* and a reasonable overhead for *Insert* and *Delete*.

Acknowledgments. The authors would like to thank the anonymous reviewers for their helpful suggestions. This work is partially supported by the U.S. Army Research Office under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI), by the NSF under grants CNS-0747247 and CCF-0914946, by NSFC under Grants No. 61170280, and SPRPCAS under Grant No. XDA06010701, and by K.C. Wong Education Foundation. The contents of this paper do not necessarily reflect the position or the policies of the U.S. Government.

References

1. Hbase, `http://hbase.apache.org/`
2. Celko, J.: Joe Celko's Trees and Hierarchies in SQL for Smarties. Morgan Kaufmann (2004)
3. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst. 26, 4:1–4:26 (2008)
4. Devanbu, P., Gertz, M., Martel, C., Stubblebine, S.G.: Authentic data publication over the internet. J. Comput. Secur. 11, 291–314 (2003)
5. Devanbu, P.T., Gertz, M., Martel, C.U., Stubblebine, S.G.: Authentic third-party data publication. In: Thuraisingham, B., van de Riet, R., Dittrich, K.R., Tari, Z. (eds.) Data and Application Security. IFIP, vol. 78, pp. 101–112. Springer, Heidelberg (2001)
6. Di Battista, G., Palazzi, B.: Authenticated relational tables and authenticated skip lists. In: Barker, S., Ahn, G.-J. (eds.) Data and Applications Security 2007. LNCS, vol. 4602, pp. 31–46. Springer, Heidelberg (2007)
7. Goodrich, M.T., Tamassia, R.: Efficient authenticated dictionaries with skip lists and commutative hashing. Technical report. Johns Hopkins Information Security Institute (2001)
8. Goodrich, M.T., Tamassia, R., Triandopoulos, N.: Super-efficient verification of dynamic outsourced databases. In: Malkin, T. (ed.) CT-RSA 2008. LNCS, vol. 4964, pp. 407–424. Springer, Heidelberg (2008)

9. Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L.: Dynamic authenticated index structures for outsourced databases. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD 2006, pp. 121–132. ACM, New York (2006)
10. Micali, S.: Efficient certificate revocation. Technical report, Cambridge, MA, USA (1996)
11. Miklau, G., Suciu, D.: Implementing a tamper-evident database system. In: Grumbach, S., Sui, L., Vianu, V. (eds.) ASIAN 2005. LNCS, vol. 3818, pp. 28–48. Springer, Heidelberg (2005)
12. Mouratidis, K., Sacharidis, D., Pang, H.: Partially materialized digest scheme: an efficient verification method for outsourced databases. The VLDB Journal 18, 363–381 (2009)
13. Mykletun, E., Narasimha, M., Tsudik, G.: Authentication and integrity in outsourced databases. Trans. Storage 2, 107–138 (2006)
14. Narasimha, M., Tsudik, G.: Authentication of outsourced databases using signature aggregation and chaining. In: Li Lee, M., Tan, K.-L., Wuwongse, V. (eds.) DASFAA 2006. LNCS, vol. 3882, pp. 420–436. Springer, Heidelberg (2006)
15. Palazzi, B., Pizzonia, M., Pucacco, S.: Query racing: fast completeness certification of query results. In: Foresti, S., Jajodia, S. (eds.) Data and Applications Security and Privacy XXIV. LNCS, vol. 6166, pp. 177–192. Springer, Heidelberg (2010)
16. Pang, H., Jain, A., Ramamritham, K., Tan, K.-L.: Verifying completeness of relational query results in data publishing. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD 2005, pp. 407–418. ACM, New York (2005)
17. Pang, H., Tan, K.-L.: Authenticating query results in edge computing. In: Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, pp. 560–571. IEEE Computer Society, Washington, DC (2004)
18. Pang, H., Zhang, J., Mouratidis, K.: Scalable verification for outsourced dynamic databases. Proc. VLDB Endow. 2, 802–813 (2009)
19. Pizzette, L., Cabot, T.: Database as a service: A marketplace assessment (2012)
20. Sion, R.: Query execution assurance for outsourced databases. In: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB 2005, pp. 601–612. VLDB Endowment (2005)
21. Tropashko, V.: Nested intervals tree encoding in sql. SIGMOD Rec. 34(2), 47–52 (2005)
22. Wei, W., Yu, T.: Practical Integrity Assurance for Big Data Processing Deployed over Open Cloud. PhD thesis, North Carolina State University (2013)
23. Wei, W., Yu, T., Xue, R.: ibigtable: Practical data integrity for bigtable in public cloud. In: Proceedings of the Second ACM Conference on Data and Application Security and Privacy, CODASPY 2013. ACM (2013)
24. Xie, M., Wang, H., Yin, J., Meng, X.: Integrity auditing of outsourced data. In: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB 2007, pp. 782–793. VLDB Endowment (2007)
25. Xie, M., Wang, H., Yin, J., Meng, X.: Providing freshness guarantees for outsourced databases. In: Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology, EDBT 2008, pp. 323–332. ACM, New York (2008)