Integrity Auditing of Outsourced Data[*]

Min Xie[†]   Haixun Wang[‡]   Jian Yin[‡]   Xiaofeng Meng[†]

[†]Renmin University of China
Beijing 100872, China
{xiemin,xfmeng}@ruc.edu.cn

[‡]IBM T. J. Watson Research Center
Hawthorne, NY 10532, USA
{haixun,jianyin}@us.ibm.com

ABSTRACT

An increasing number of enterprises outsource their IT services to third parties who can offer these services for a much lower cost due to economy of scale. Quality of service is a major concern in outsourcing. In particular, query integrity, which means that query results returned by the service provider are both correct and complete, must be assured. Previous work requires clients to manage data locally to audit the results sent back by the server, or database engine to be modified for generating authenticated results. In this paper, we introduce a novel integrity audit mechanism that eliminating these costly requirements. In our approach, we insert a small amount of records into an outsourced database so that the integrity of the system can be effectively audited by analyzing the inserted records in the query results. We study both randomized and deterministic approaches for generating the inserted records, as how these records are generated has significant implications for storage and performance. Furthermore, we show that our method is provable secure, which means it can withstand any attacks by an adversary whose computation power is bounded. Our analytical and empirical results demonstrate the effectiveness of our method.

1. INTRODUCTION

Recent advance of network technology allows an increasing number of enterprises to outsource their IT functions or business processes to third parties that can provide these services for a much lower cost due to economy of scale. According to a recent survey, IT outsourcing has grown by a staggering 79% as companies seek to reduce costs and focus on their core competencies. Data processing service outsourcing is a major component as most of IT functions evolve around data processing.

Security is essential for outsourced data processing services. Because a third party service provider may not be trusted or may not be securely administrated, security properties must be assured at the infrastructure level. In this paper, we focus on mechanisms that provide security assurance for database services offered by third parties.

Problem Setting. In the database outsourcing scenario, the database owner stores data at a service provider, and the clients send queries to the service provider (Figure 1). We assume data and communication are encrypted, the database system at the service provider supports query processing over encrypted data, and the problem of data privacy has been taken care of [5, 16].
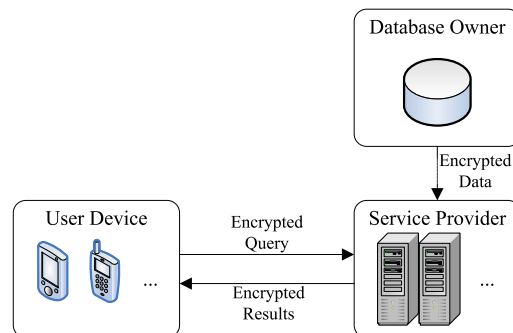

Figure 1: System Model

In addition to data privacy, an important security concern in the database outsourcing paradigm is integrity [11, 12, 15, 9]. When a client receives a query result from the service provider, he wants to be assured that the result is both *correct* and *complete*, where *correct* means that the result must originate in the owner's data and has not been tampered with, and *complete* means that the result includes all records satisfying the query. The goal of this paper is to provide a simple and elegant protocol to monitor the integrity of outsourced database services.

Providing integrity assurance is a new and challenging task. Traditional DBMSs do not have this issue because in-house data processing is always trusted. Current approaches for this problem require either changes to be made in DBMS kernels, or a significant subset of the data to be stored locally at the client site. Both of the approaches are costly, hard to implement, and ineffective at least in some scenarios. Particularly, a severe challenge is triggered by a rising trend in mobile computing – more and more clients are accessing database services from such devices as PDAs and cell phones, which have limited storage capacity and processing power. Thus, a protocol for integrity assurance needs to impose little storage or computation overhead in the client side.

Overview of our approach. In this paper, we propose a probabilistic integrity audit method. We insert a small number of tuples into the outsourced database. For a query issued against this augmented database, there is certain probability that a small amount of the inserted tuples are returned with the original data. The integrity of the system can be effectively monitored by analyzing the inserted tuples in a reply.

To perform the analysis, the client must know what tuples have been inserted into the outsourced database. If an inserted tuple that satisfies the query is absent from the reply, then we know the integrity is breached; if all the inserted tuples that satisfies the query does appear in the reply, we can deliver a probabilistic assurance on query integrity.

We address several challenges to this task. First, in order to know the set of the inserted tuples to be returned, the client must keep a copy of the inserted tuples. This requires local data storage and local query processing. In our approach, we use a deterministic function to "describe" the inserted data. As the data generated by the function is encrypted, it is impossible for a service provider to differentiate the inserted data from any other data in the encrypted database. As a result, we only need to store the definition of the function at the client side instead of all the inserted data.

A second challenge is to ensure that our integrity monitoring scheme is secure. If the adversaries or the untrusted service provider can tell inserted tuples from original tuples, then our scheme will fail. We must ensure that the query processing process does not provide the adversaries any information that may lead to security breach. To this end, we show that our scheme is provable secure, which means it achieves the highest level of security when the adversaries are assumed to be computationally bounded.

A third challenge is generality. Previous work has largely focused on simple selection queries. We show that our scheme provides integrity check for joins and updates.

Overall, our technique is applicable to a wide range of data processing services including search engines, storage systems, backup systems, etc. In the paper, we use database as an example as it illustrates most of features in our approach.

Paper Organization. In Section 2, we review the previous work related to security in outsourced databases. Section 3 introduces the background information about query integrity assurance. Section 4 and 5 introduce our scheme for integrity monitoring, and Section 6 shows that it is provable secure. Section 7 extends our scheme to support advanced queries, and Section 8 studies how to optimize our scheme. We show empirical results of our approach in Section 9, and conclude in Section 10.

2. RELATED WORK

When we outsource database operations to an untrusted service provider, we face two challenges: *data privacy* and *query integrity*. Much work has been done to protect data privacy; this paper focuses on protecting query integrity.

Hacigümüs et. al. [8] first brought up security issues in the scenario of database outsourcing. It focuses on the *privacy* aspect of the outsourced database, in particular, efficiency of various encrypting schemes using both hardware and software encryption. That work does not consider the problem of data integrity.

The pioneering work on the problem of integrity [6, 11] focuses on the authentication of the data records, that is, the *correctness* aspect of the integrity. Devanbu et. al. [6] authenticates data records using the Merkle hash tree [10], which is based on the idea of using a signature on the root of the Merkle hash tree to generate a proof of *correctness*. Mykletun et. al. [11] discussed and compared several

signature methods which can be utilized in data authentication, and they identified the problem of *completeness*, but did not provide a solution.

Some recent work [12, 9, 15] studied the problem of auditing the *completeness* aspect of the integrity. By explicitly assuming an order of the records according to one attribute, Pang et. al.[12] used an aggregated signature to sign each record with the information from two neighboring records in the ordered sequence, which ensures the result of a *simple selection query* is continuous by checking the aggregated signature. But it has difficulties in handling *multipoint selection query* of which the result tuples occupy a non-continuous region of the ordered sequence. Besides, it can only handle a subclass of join operations efficiently, the *primary key/foreign key* join, because that the result of the join forms a continuous region of original ordered data can only be assured in this case. Other work [6, 9] uses Merkle hash tree based methods to audit the *completeness* of query results, but since the Merkle hash tree also uses the signature of the root Merkle tree node, a similar difficulty exists. The network and CPU overhead on the client side can be prohibitively high for some types of queries. In some extreme cases, the overhead could be as high as processing these queries locally, which can undermine the benefits of database outsourcing. Moreover, to ensure freshness, an additional system is needed to deliver the most up-to-date root signature to all the clients in a reliable and timely manner. It is unclear where such a system can be placed in an outsourced database environment, while the freshness problem could be solved naturally in our approach without additional requirements. In our approach, we just need to vary the inserted tuples according to the current time. We will prototype this functionality in our future work.

Sion [15] introduces a mechanism called the *challenge token* and uses it as a probabilistic proof that the server has executed the query over the entire database. It can handle arbitrary types of queries including joins and does not assume the underlying data is ordered. But their scheme cannot detect all malicious attacks, for instance, when the service provider computes the complete result but returns part of it for sake of business profit from a competition rival.

Finally, one significant advantage of our scheme over previous methods is that all previous methods must modify the DBMS kernel in order to provide proof of integrity (e.g., the aggregated signature methods [6, 12, 9, 15], and the challenge token mechanism in Sion's work [15]). This requirement often renders these methods impractical to deploy in real life. Our work audits query integrity without requiring the database engine to perform any special function beyond query processing, as our integrity check relies on nothing but the query results returned by the DBMS.

3. PRELIMINARIES

We audit query integrity by inserting a small number of *fake tuples* into the outsourced database and then analyzing the fake tuples that show up in the query result. Our approach has three basic needs: data encryption, data authentication, and tuple authentication (i.e., telling fake tuples from real tuples). We describe these preliminaries below.

Data Encryption. Data is encrypted to guarantee privacy. But, in the scenario of database outsourcing, we have an additional requirement for the encryption scheme, that is, it must be able to support queries directly over encrypted data. This topic has been studied by much recent work [7, 1]. The ultimate goal of the research in this field is to prevent the adversary from learning anything about the encrypted data and still preserving the ability to processing encrypted queries efficiently. The research in this area

complements our work described in this paper and our solution can leverage progress in this area.

Data Authentication. For data authentication, we add a special che- cksum or *header* column a_h to the dataset. We compute a_h using Eq 3.1, where $\oplus$ denotes string concatenation and H is a one-way hash function.

$$a_h = H(tid \oplus a_1 \oplus a_2 \oplus \ldots \oplus a_n) \qquad (3.1)$$

The one-way hash function H has the following property [2]. It takes a variable length input string x and converts it into a fixed-length (e.g., 128 bits) binary sequence $H(x)$. However, it is difficult to reverse the process, in other words, given a value x', it is computationally infeasible for an attacker to find an x such that $H(x) = x'$.

It follows that it is computationally infeasible for the attacker to compute a valid *header* from the encrypted data, because the encryption prevents the attacker from knowing the plain text a_i, which is the required input to the one-way hash function. Thus, any modification to the original record or insertion of foreign records will not pass header verification. Furthermore, the unique tid will prevent adversary from inserting duplicate tuples into the dataset.

It is clear that the *correctness* aspect of query integrity can be ensured by data authentication. Given a query q, its result R_q is correct as long as each tuple $t \in R_q$ is valid, and t satisfies q. The major focus of our work is to ensure R_q is *complete*.

Tuple Authentication. We audit query integrity by inserting a small number of *fake tuples* into the outsourced database. The client authenticates a tuple to tell whether it is a fake tuple or a real tuple. To do this, we use Eq 3.2 to derive the checksum or the *header* of a tuple $t = (tid, a_1, \ldots, a_n)$.

$$a_h = \begin{cases} H(tid \oplus a_1 \oplus \ldots \oplus a_n) & t \text{ is real} \\ H(tid \oplus a_1 \oplus \ldots \oplus a_n) + 1 & t \text{ is fake} \end{cases} \qquad (3.2)$$

On receiving a result tuple from the service provider, the client knows it is a real tuple if its header equals to $H(tid \oplus a_1 \oplus \ldots \oplus a_n)$, or a fake tuple if its header equals to $H(tid \oplus a_1 \oplus \ldots \oplus a_n) + 1$. Otherwise it is an invalid tuple. The server, on the other hand, cannot tell a fake tuple from a real tuple because of the encryption and the use of the one-way hash function. Tuple authentication will not cause any storage overhead because the header is of fixed length (128 bits), and the computation overhead of Eq 3.2 over Eq 3.1 is negligible.

4. RANDOMIZED APPROACHES

We introduce an integrity auditing approach based on inserting fake tuples. In this naïve approach, fake tuples are randomly generated, and they are stored at the client side. In the next section, we discuss an advanced approach which avoids storing fake tuples by using a deterministic function to generate fake tuples. We start our discussion with queries containing range predicates only, and we will focus on joins and updates in later sections. The queries we are concerned with have the following form:

> **SELECT** $\star$ **FROM** T
> **WHERE** $T.A$ BETWEEN a_1 AND a_2 AND
> $\quad\quad\quad T.B$ BETWEEN b_1 AND b_2 AND $\ldots$

4.1 Method

Assume that the server returns R_Q for a query Q and the client knows in advance that R_Q should include certain tuples. Then, if any of them is absent, we know immediately that there is an attack. Moreover, if there is an attack, we should be able to catch it with a high probability. To be effective, we should have the ability to catch any attack with a high probability. One naïve approach is the following: we randomly generate a set of fake tuples, insert them into the outsourced dataset at the server side, and maintain a copy of them at the client side. When the client obtains result R_Q for query Q from the server, it queries Q against its own copy of fake tuples, and finds out what are the tuples that should appear in R_Q.

To audit whether all fake tuples covered by Q appear in R_Q can be a costly process, for the client needs to join R_Q with its own copy of fake tuples to get the result. To alleviate the cost, we use the *header* column information for each tuple t to easily find out the total number of fake tuples returned by the server for query Q.

Let $C_s(Q)$ be the set of fake tuples in R_Q, and let $C_c(Q)$ be the tuples among the client's copy of the fake tuples that satisfy Q. We have the following conclusion:

THEOREM 1. *If $|C_s(Q)| = |C_c(Q)|$, then $C_s(Q) = C_c(Q)$.*

PROOF. Assume to the contrary $C_s(Q) \neq C_c(Q)$. As $|C_s(Q)| = |C_c(Q)|$, $\exists t \in C_s(Q)$ such that $t \notin C_c(Q)$. But $t \in C_s(Q)$ means t is a fake tuple, whose authenticity is guaranteed by the encryption and the one-way hash function, and since t satisfies Q, t must appear in $C_c(Q)$. $\square$

Theorem 1 enables the client to audit the *completeness* of R_Q by counting the tuples, which avoids the join operation. Now, if $|C_s(Q)| \neq |C_c(Q)|$, we know immediately there is a problem. But if $|C_s(Q)| = |C_c(Q)|$, how likely the server is problem-free?

4.2 Probabilistic guarantee

The randomized approach of *completeness* auditing is a probabilistic approach. We analyze the probability of it being attacked.

As we have mentioned in Section 3, modifying tuple content or adding new tuples are easily detected through data authentication. Thus, the only attack is to delete tuples from the outsourced data or from the query result. Assume we randomly insert K fake tuples into the original data of N tuples. If an attacker deletes one tuple from the original database, the probability that it is not a fake tuple is $N/(N+K)$. Thus, with probability $N/(N+K)$, a deletion is not caught. If an attacker deletes m tuples from the database, he can avoid being caught with probability

$$\prod_{i=0}^{m-1} \frac{N-i}{K+(N-i)} \qquad (4.3)$$

Figure 2 shows that for a dataset of $N = 1,000,000$ tuples, with the number of fake tuples tuple ranges from 5% to 50% of N, the probability of escaping detection when 1 to 100 tuples are deleted from the database. It shows that the probability of escape decreases sharply when the number of fake tuples or deletion increases. In particular, when the fake tuples are more than 10% of the original data, and more than 50 tuples are deleted, it is close to impossible for the attacker to escape from being caught by the randomized approach.

5. DETERMINISTIC APPROACHES

Randomized approaches provide good protection against attacks. However, for many applications, it has one significant drawback: in order to audit query *completeness*, it has to store all randomly generated fake tuples at the client side, and actually evaluate queries on them. In this section, we introduce a novel approach to eliminate this drawback.
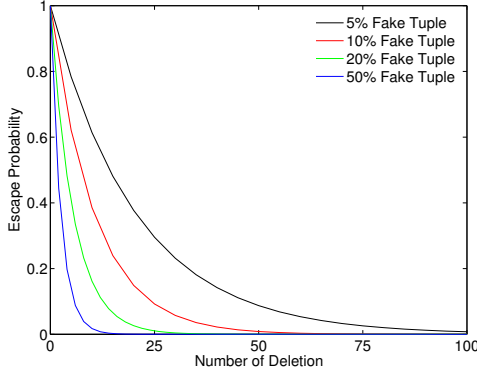
Figure 2: Escape analysis (N=1,000,000)

Figure 3: Grids Covered by a Query

5.1 Overview

Instead of generating fake tuples randomly, we use some predefined, deterministic functions to generate the fake tuples. Assume a dataset T has n attributes. We consider function $\mathcal{F}: D_1 \times D_2 \times \cdots D_{n-1} \to D_n$ where D_i is the domain of the i-th attribute. In other words, we first choose $n-1$ attribute values, and then use the function to derive the value of the remaining attribute. We then form a fake tuple with the n attribute values, and add it into the outsourced database. Figure 4 shows two possible functions for generating fake tuples.

During query processing, we audit if the fake tuples returned by the service provider are all the fake tuples that satisfy the query. To make the auditing more efficient, we divide the feature space into grids by discretizing each attribute (if the attribute is numerical). We can regard the result of a range query as a set of fully covered or partially covered grids in the feature space. Then, integrity auditing boils down to counting how many fake tuples each fully covered or partially covered grid contains.

The benefit of the deterministic method is obvious: instead of storing a large set of fake tuples, we store a deterministic function and a small grid data structure; instead of querying the fake tuples, we ask how many tuples the deterministic function will produce given a set of query predicates. In the following, we first discuss how to audit query integrity using the deterministic method, and then we describe the requirement of the deterministic function.

5.2 Deterministic completeness auditing

In randomized approaches, auditing query *completeness* for a query Q takes 3 steps. First, we find $C_s(Q)$, the fake tuples in the query result of Q's; Second, we evaluate query Q against the client's copy of fake tuples to derive the result $C_c(Q)$; Third, we check if $|C_s(Q)| = |C_c(Q)|$. Deterministic *completeness* auditing only differs in the second step. Because the fake tuples are not stored, instead of evaluating Q against the local data, we evaluate it against the deterministic function $\mathcal{F}$, to find out how many tuples would have satisfied query Q if they had been generated, stored, and queried. We use an example to illustrate the auditing process and motivate the use of grids in auditing.

EXAMPLE 1. *Assume a dataset has two attributes: Price and Quantity, the range of each attribute is from 0 to 100, and each attribute has been discretized into four subranges $\{[0, 25), [25, 50), [50, 75), [75, 100]\}$ to form a 4×4 grid, which is shown in Figure 3.*

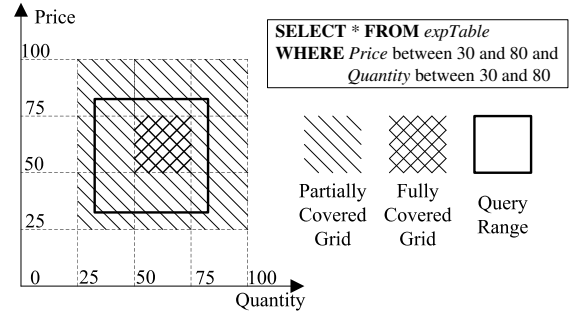The query Q in Figure 3 searches for tuples whose Price, Quan-

tity values are both in the range of $[30, 80]$. In order to find out how many fake tuples are in this range, one approach is to generate all the tuples in the range. However, this is rather inefficient. As we can see from the figure, the query result involves 9 of the 16 grids. Of these 9 grids, 8 are partially covered by Q, and 1 is fully covered by Q. The total number of fake tuples that satisfy the query is:

$$n = \sum_{g \in B_F} n_g + \sum_{g \in B_P} partial(g, Q) \tag{5.1}$$

where B_F is the set of fully covered grids, B_P is the set of partially covered grids, n_g is the total number of fake tuples generated in grid g, and $partial(g, Q)$ is a function that counts the number of fake tuples in g that satisfy query Q. It is clear that since n_g is irrelevant to query Q, and its value can be stored in the grid structure, all we need is an efficient way to compute $partial(g, Q)$ for each grid g in the partially covered area.

5.3 Deterministic generating functions

The core of our deterministic approach is the function $\mathcal{F}$. We want to use a function $\mathcal{F}$ so that *completeness* auditing can be performed efficiently. More specifically, we explore necessary properties of $\mathcal{F}$ to speed up the evaluation of $partial(g, Q)$.

First of all, any arbitrary function $\mathcal{F}$ such as $sine$, $cosine$, or any *polynomial* functions can be used to generate the required set of fake tuples for each grid. But in order to carry out integrity auditing, we may have to regenerate *all the fake tuples* using $\mathcal{F}$, which is not efficient.

We show that if the fake tuples generated by $\mathcal{F}$ in a grid g are *continuously covered* by a range query Q, then we can evaluate $partial(g, Q)$ efficiently. We use an example to demonstrate what it means by *continuously covered*. In Figure 4(a), there are two points outside the query range, but their neighboring points on both sides are inside the range. This is a case where points are not continuously covered by a range range. In contrast, the points generated by a monotonic function in Figure 4(b) are continuously covered.

It is easy to see that if fake tuples generated by $\mathcal{F}$ are always continuously covered by a range query, then we do not have to re-generate all fake tuples in *partially covered grids* for integrity audit. Thus, an intuitive improvement to the method is that instead of using arbitrary functions, we put a constrain on the function so that the tuples it generated are continuously covered.

Let a and b be two vectors in a k-dimensional space. We say $a \prec b$ if and only if $a[i] \leq b[i]$ for each dimension i. In our case, we regard $\mathcal{F}$ as a function that maps an (n-1)-dimensional vector to a single value. We say $\mathcal{F}$ is monotonic increasing (or decreasing) if $\mathcal{F}(a) \leq \mathcal{F}(b)$ for any a and b where $a \prec b$ (or $b \prec a$).
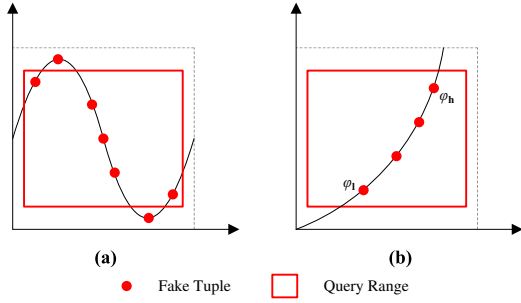
Figure 4: Function Property Analyze

THEOREM 2. *Let R be the range predicates in a query. Let S be a set of tuples generated by a monotonic function $\mathcal{F}$ where each tuple $t_x \in S$ is in the form of $t_x = (x, \mathcal{F}(x))$. Then, for any $a \prec b \prec c$, if t_a and t_c satisfy R, it must be true that t_b satisfies R.*

PROOF. Assume to the contrary that t_b does not satisfy R, which means there is at least one range $[begin, end]$ on one dimension i where $begin \leq t_b[i] \leq end$ does not hold. However, if $1 \leq i \leq n-1$, it contradicts the fact that $a \prec b \prec c$; if $i = n$, it contradicts the fact that $\mathcal{F}(a) \leq \mathcal{F}(b) \leq \mathcal{F}(c)$. $\square$

Based on Theorem 2, we can guarantee that if two tuples are covered by a query Q, then all tuples in-between the first and the last covered tuple mush be covered by Q. Figure 4(b) is a simple example that shows our intuition. As a result, to count the *fake tuples* covered by Q, we only need to find the two intersection points between the function $\mathcal{F}$ and the range of Q. We can use efficient binary search algorithm to find the two intersection points and get the count. We analyze an example below using linear generating functions.

5.4 Linear generating function

In this section, we analyze an example where $\mathcal{F}$ specifies a line segment in the feature space. We divide the whole n-dimension feature space into grids. For each grid g, two points $\vec{e}$ and $\vec{s}$ in g's feature space defines a line segment, and hence defines $\mathcal{F}$.

$$L = \{\vec{s} + (\vec{e} - \vec{s}) \cdot t \mid t \in [0,1]\} \qquad (5.2)$$

Assume we generate k fake tuples in the grid. We generate the tuples by *uniformly* selecting k points on the line segment from $\vec{s}$ to $\vec{e}$. Thus, the set of fake tuples are:

$$S = \{\vec{s} + (\vec{e} - \vec{s}) \cdot \frac{j}{k} \mid j \in \{0, 1, \cdots, k-1\}\} \qquad (5.3)$$

We associate each grid with a 3-tuple $(k, \vec{s}, \vec{e})$ to indicate how fake tuples are generated in the grid. Although $\vec{s}$ and $\vec{e}$ can be two arbitrary points in the feature space of the grid, we choose the points so that they are as far apart as possible, so that the grid is better "covered" by the line segment[1]. We can simply set $\vec{s} = (l_1, \ldots, l_n)$ and $\vec{e} = (u_1, \ldots, u_n)$, where (l_i, u_i) is the range of dimension i for the grid.

Note that although the fake tuples may exhibit a clear pattern in the plain text data (e.g., here for a linear $\mathcal{F}$, they are evenly distributed on a line), neither the service provider nor the attackers will be able to see the pattern, or utilize the pattern to break our scheme. To show this, we conduct a detailed security analysis of

---

[1]We study the distribution of the fake points and its relationship to the integrity assurance in detail in Section 8.

our scheme in Section 6, which shows that our scheme is provable secure.

Using a deterministic function enables us to audit a query without storing the faking tuples. For range queries, we do not even have to re-generate the fake tuples in auditing. More specifically, given a query Q, a client can figure out easily how many fake tuples exist in a partially covered grid. This process is outlined by algorithm $partial_{Linear}$ shown below. It first finds the two intersection points $\vec{\varphi}_l$ and $\vec{\varphi}_h$ between the query rectangle and the line segment using binary search (line 6-7), and then it computes the number of fake tuples that lie between the two points (line 8-11).

Function $partial_{Linear}(g, Q)$

Input: g : a grid partially covered by Q
Input: Q : a query
Output: n_g : number of fake tuples in g that satisfy Q

1 **begin**
2     $(\vec{s}, \vec{e}, k) \leftarrow$ The parameters associated with g
3     $\vec{t}_s \leftarrow \vec{s}$ // First Fake Tuple in g
4     $\vec{t}_e \leftarrow \vec{s} + (\vec{e} - \vec{s}) \cdot \frac{k-1}{k}$ // Last Fake Tuple in g
5     $\vec{\zeta} \leftarrow \frac{\vec{e} - \vec{s}}{k}$ // Gap Between Two Neighbor Fake Tuples in g
6     $\vec{\varphi}_l = BSearchL(\vec{t}_s, \vec{t}_e, \vec{\zeta}, Q)$
7     $\vec{\varphi}_h = BSearchH(\vec{t}_s, \vec{t}_e, \vec{\zeta}, Q)$
8   **if** $\vec{s} \leq \vec{\varphi}_l \leq \vec{e} \, AND \, \vec{s} \leq \vec{\varphi}_h \leq \vec{e} \, AND \, \vec{\varphi}_l \leq \vec{\varphi}_h$ **then**
9       $n_g = (\vec{\varphi}_l == \vec{\varphi}_h) \, ? \, 1 : (\frac{k \cdot (\vec{\varphi}_h - \vec{\varphi}_l)}{\vec{e} - \vec{s}})$
10   **else**
11       $n_g = 0$
12   **return** n_g
13 **end**

5.5 Cost analysis

Let Q be an arbitrary query. Assume the outsourced data base contains N fake tuples, out of which n satisfy Q. We analyze the cost incurred by auditing the integrity of Q.

- COMMUNICATION COST: The extra cost of communication comes from sending the n fake tuples from the service provider to the client. We denote this cost as $n \cdot \psi_N$, where ψ_N is the average cost of sending one tuple.

- COST ON THE SERVER SIDE: The extra cost comes from evaluating Q on the fake tuples. We denote this cost as $N \cdot \psi_Q$, where ψ_Q is the tuple average cost of processing Q.

- COST ON THE CLIENT SIDE: We decompose the cost into: (i) ψ_D: per tuple cost of decryption; (ii) ψ_C: per tuple cost of *correctness* auditing (the cost of analyzing the checksum or the header); (iii) ψ_P: cost of *completeness* auditing for the query. The extra cost at the client side is: $n \cdot \psi_D + n \cdot \psi_C + \psi_P$.

The *completeness* auditing consists of two tasks: (i) count the number of fake tuples in the result of Q and (ii) derive from $\mathcal{F}$ the number of fake tuples that satisfy Q. The first task can be incorporated into tuple validation with virtually no extra cost. For the second task, the cost is simply $\sum_{g \in B_P} 2 \log n_g$ for linear $\mathcal{F}$, where B_P is the set of partially covered grids and n_g is the number of fake tuples in a grid. Since $\log n_g \ll \log N$, the cost is dominated by $n \cdot \psi_D + n \cdot \psi_C$.

Thus, the total extra cost comes to:

$$N \cdot \psi_Q + n \cdot (\psi_N + \psi_D + \psi_C) \qquad (5.4)$$

It is clear that the extra cost for using fake tuples for integrity audit is a linear function of N and n, the total number of fake tuples in the outsourced database, and the number of fake tuples in each query result. Thus, a criterion for our scheme is how many fake tuples we need in order to reach certain accuracy in integrity audit. We show in Section 9 that our scheme has good performance in this regard.

6. SECURITY STATEMENT

In this section, we prove the security our system in an ideal setting. Our data outsourcing scheme is a provably secure scheme assuming that the underlying encryption function is a secure pseudorandom permutation. We follow the standard notations as in the practice-oriented provable security literature[4, 3].

We first introduce some necessary definitions for developing our theorem and proof.

DEFINITION 1. *Function family*
A function family F is a finite collection of functions together with a probability distribution on them. All functions in the collection are assumed to have the same domain and the same range.

There is a set of "keys" and each key names a function in F. We use F_k to denote the function selected by key k in the function family F.

DEFINITION 2. *ϵ-distinguisher*
Suppose that F_0 and F_1 are two function families. Let $\epsilon > 0$ and let f_0 and f_1 be two functions selected from F_0 and F_1 uniformly randomly. A distinguisher $\mathcal{A}$ is an algorithm; given a function, $\mathcal{A}$ outputs 0 or 1 as it determines whether the function is from F_0 or F_1. We use $Adv_{\mathcal{A}}$ to denote $\mathcal{A}$'s advantage in distinguishing F_0 from F_1.

$$Adv_{\mathcal{A}} = |Pr[\mathcal{A}(f_0) = 1] - Pr[\mathcal{A}(f_1) = 1]|$$

We say algorithm $\mathcal{A}$ is an ϵ-distinguisher of F_0 and F_1 if $Adv_{\mathcal{A}} > \epsilon$.

DEFINITION 3. *(q, t, ϵ)-pseudorandom*
A function family $F : U \to V$ is (q, t, ϵ)-pseudorandom if there does not exist an algorithm $\mathcal{A}$ that can ϵ-distinguish a pseudorandom function from a truely random function. Here A is allowed to use F_k as an oracle for q queries, and use no more than t computation time.

Given a dataset T, we generate a dataset S. We encrypt $S \cup T$ by applying F_k, where F_k is a (q, t, ϵ)-pseudorandom permutation. We then store the result, $X = F_k(S \cup T)$, at the service provider. The highest level of security is achieved if any subset from $F(T)$ is indistinguishable from a random subset of X to attackers. Here we use m to denote $|T|$ and n to denote $|S|$.

THEOREM 3. *There does not exist an adversary algorithm that can succeed in selecting l tuples from X such that all the l tuples are in T with a possibility bigger than $(\frac{m}{n+m})^l + \epsilon$ with $t - c$ computation and $q - m - n$ queries.*

PROOF. (Sketch) We prove this by contradiction. We assume there exists an algorithm $\mathcal{G}$ that can successfully choose l tuples from X such that the tuples are in T with a probability significantly higher than $(\frac{m}{n+m})^l + \epsilon$. We then construct an algorithm A

that breaks F_k, that is, we show F_k is not a (t, q, ϵ)-pseudorandom function.

We construct an algorithm A that works as follows. Algorithm A passes $T \cup S$ to its oracle, which generates an encrypted X_e. Algorithm A then passes X_e to algorithm $\mathcal{G}$, and $\mathcal{G}$ selects l tuples. Algorithm A then checks the l tuples to see whether they are all in T. If it is the case, it outputs 1, otherwise it outputs 0.

Clearly, if the underlying encryption is a random permutation R, then we have $Pr[A(R) = 1] = \frac{\binom{m}{l}}{\binom{n+m}{l}}$. However, if the underlying encryption is F_k, then algorithm $\mathcal{G}$ has advantage larger than ϵ over a random algorithm in selecting l tuples from T, in other words, we have $Pr[A(F_k) = 1] = (\frac{m}{n+m})^l + E$, where $E > \epsilon$. Let c be the amount of computation taken outside of $\mathcal{G}$. The total amount of computation is $(t - c) + c = t$ and the number of queries is $(q - m - n) + m + n = q$.

Thus, we have

$$
\begin{aligned}
Adv_A &= Pr[A(F_k) = 1] - Pr[A(R) = 1] \\
&= (\frac{m}{n+m})^l + E - \frac{\binom{m}{l}}{\binom{n+m}{l}} \\
&> \frac{\binom{m}{l}}{\binom{n+m}{l}} + E - \frac{\binom{m}{l}}{\binom{n+m}{l}} \\
&= E > \epsilon,
\end{aligned}
$$

which contradicts the fact that F_k is a (t, q, ϵ)-pseudorandom function. $\square$

7. BEYOND SIMPLE SELECTION QUERY

As in previous work [15], we have mainly focused on simple selection queries. In this section, we study join operations and update queries.

7.1 Integrity audit of join

Join is a very important operator in relational algebra. Previous work either cannot handle arbitrary join queries [12, 9] or need an explicit proof which may make the scheme impractical to deploy [15] for a service provider (see Section 2 for a detailed discussion). In this section, we discuss how our scheme supports join operations without any additional requirement on the service provider.

Without much loss of generality, we assume the join query has the following form:

```
SELECT *
FROM T₁, T₂
WHERE T₁.A op T₂.A AND T₁.B op T₂.B AND ...
    AND pred(T₁) AND pred(T₂) AND ...
```

where *op* is $=, >, \geq, <, \leq$ or $\neq$, and *pred(T_i)* is a predicate on table T_i.

Join Decomposition

As in auditing simple selection queries, we use the special *header* value in the result tuple (Note here we have two *header*s, one for T_1 and the other for T_2) to check whether the results are valid and correct. Here, we focus on the *completeness* aspect of the integrity.

Let us consider a join operation between two tables T_1 and T_2. Let $T_1 = T_{1o} \cup T_{1c}$ and $T_2 = T_{2o} \cup T_{2c}$, where T_{1o} and T_{2o} are the real tuples, and T_{1c} and T_{2c} are the fake tuples in T_1 and T_2. Thus, the result of $T_1 \bowtie T_2$ can be divided into four cases:

1. $T_{1c} \bowtie T_{2c}$

2. $T_{1c} \bowtie T_{2o}$

3. $T_{1o} \bowtie T_{2c}$

4. $T_{1o} \bowtie T_{2o}$

In Figure 5, we assume that each of T_1 and T_2 contains 2 fake and 2 real tuples, and their join results cover the above 4 cases. In the figure, we use **C** in the header to indicate a fake tuple, and **O** a real tuple.

Figure 5: Decomposing a natural join between $T_1(A, B)$ and $T_2(B, C)$ on column B into 4 parts.

Given the join query, the client can derive the fake tuples in T_1 and T_2 that satisfy the join condition. In other words, the client derives T_{1c} and T_{2c}. Thus, if any tuple in $T_{1c} \bowtie T_{2c}$ is missing from the result of the join, we know that the service provider is problematic.

However, for case 2, case 3, and case 4, we do not have such guarantee, because the client does not know the content of T_{1o} and T_{2o}. In other words it cannot derive the results of the join when the join involves real tuples.

We show that, for case 2, case 3, and case 4, useful information can be inferred from the join results to audit the join query. Consider one of the tuples in the join result $t = \{1, 2, 2\}$, which falls into case 4 (joined by two real tuples). It gives us the following *hint*: *At least one real tuple from T_1 has value 2 in join column B.* (We can infer the same thing for T_2). From this, we can further infer that: *All fake tuples in T_2 whose column B value is 2 must be in the join result (provided they satisfy all other predicates in the query)*. Similar hints can be obtained from result tuples for case 2 and case 3.

In summary, we can use *the value of join attributes* in the result of a join to audit the join query. Next, we formalize our analysis, and develop an efficient algorithm to audit join queries using the hints we described above.

Join Audit Algorithm

To audit join integrity, we use two pieces of information: the fake tuples that satisfy the join condition (we obtain fake tuples through the deterministic function $\mathcal{F}_T$) and information derived from the join result.

Indeed, using deterministic functions $\mathcal{F}_{T_1}$ and $\mathcal{F}_{T_2}$, we can easily obtain tuples in case 1: $T_{1c} \bowtie T_{2c}$. Then, we check if $T_{1c} \bowtie T_{2c}$ derived by the client really appear in the result. According to Theorem 1, we only need to check whether their count match, which simplifies the operation.

To audit case 2, 3, and 4, we rely on the hints obtained from the *join attribute values* in the join result returned by the service provider. For each of the three cases, we find those fake tuples in

T_{1c} and T_{2c} that must join with at least one real tuple and hence appear in the result of case 2 or case 3.

Let $\mathcal{A}$ be the set of join attributes. Let $r.\mathcal{A}\ op\ t.\mathcal{A}$ denote the join predicates. Let C_2 and C_3 denote the join results in case 2 and 3 respectively. We first obtain the set of fake tuples R and use the corresponding auditing rules below to check whether integrity condition is satisfied:

Rule A (Result tuple t is from case 2) :
$R = \{r | r \in T_{1c} \wedge r.\mathcal{A}\ op\ t.\mathcal{A}\}$
Condition: $\forall r(r \in R) \rightarrow \exists o(o \in T_{2o} \wedge r \bowtie o \in C_2)$

Rule B (Result tuple t is from case 3) :
$R = \{r | r \in T_{2c} \wedge t.\mathcal{A}\ op\ r.\mathcal{A}\}$
Condition: $\forall r(r \in R) \rightarrow \exists o(o \in T_{1o} \wedge o \bowtie r \in C_3)$

Rule C (Result tuple t is from case 4) :
$R_1 = \{r | r \in T_{1c} \wedge r.\mathcal{A}\ op\ t.\mathcal{A}\}$
$R_2 = \{r | r \in T_{2c} \wedge t.\mathcal{A}\ op\ r.\mathcal{A}\}$
Condition 1: $\forall r(r \in R_1) \rightarrow \exists o(o \in T_{2o} \wedge r \bowtie o \in C_2)$
Condition 2: $\forall r(r \in R_2) \rightarrow \exists o(o \in T_{1o} \wedge o \bowtie r \in C_3)$

For instance, case 4 corresponds to $T_{1o} \bowtie T_{2o}$. For any tuple $t \in T_{1o} \bowtie T_{2o}$, the values of the join column $\mathcal{A}$ come from real tuples in T_{1o} and T_{2o}. We then obtain R_1 and R_2, which are fake tuples in T_{1c} and T_{2c} that will join with T_{2o} and T_{1o} respectively. The two conditions in Rule C above basically checks if such join results appear in C_2 and C_3. The correctness of the above rules can be easily proved. We omit the proof here due to lack of space.

An efficient method to implement this auditing scheme for join is to scan the join result once. For each result tuple, according to which case it belongs to, we either count the tuples (Case 1) or identify whether there are some result tuples inferred by this tuple using the 3 rules above (Case 2, 3 and 4). The process can be carried out by simply using the deterministic functions $\mathcal{F}_{T_1}$ and $\mathcal{F}_{T_2}$ at runtime without any need to pre-store the fake tuples.

We use algorithm $JoinAudit$ to audit a join operation. In the algorithm, to audit the results from case 1, we count the number of tuples in $T_{1c} \bowtie T_{2c}$ both at the client side and in the result obtained from the service provider (line 5-6, line 18-19). If the two count not match we can alarm an attack (line 20-21). In order to use the hints from the join results based on the above rules, we initiate a hash table H which acts as a cache to temporarily memorize which fake tuples should appear in the result, and every time we see a result tuple t from case 2, 3 or 4, using one of the three rules, we generate fake tuples needed to be checked, and insert them into the hash table and check whether the integrity conditions are satisfied (line 7-17) (Given a result tuple t, let $t.T_1$ be the tuple from T_1 which forms t, the same for $t.T_2$). When we have processed all the result tuples, we check all the entries of the hash table, if one entry of hashtable is not marked with `Checked`, we know that some fake tuples are missing in the result, so we alarm an attack (line 20-21).

7.2 Integrity audit of updates

Auditing the integrity of database updates is a challenging task. The goal is to ensure that every update operation is really executed at the server side. In previous works, there either lacks consideration of the update operation or the auditing scheme has large overhead [12, 15]. In this work, we focus on auditing INSERT and DELETE operations[2] in outsourced databases.

---

[2] UPDATE is considered as a combination of INSERT and DELETE.

Function *JoinAudit* (*Result*)

Input: *Result* is result set from Server-Side

```
1  begin
2  |  n ← 0
3  |  H ← A new hash table
4  |  foreach tuple t ∈ Result do
5  |  |  if t belongs to case 1 then
6  |  |  |  n ← n + 1
7  |  |  if t belongs to case 2, 3, or 4 then
8  |  |  |  /*Use F_{T_1}, F_{T_2} to generate fake tuples*/
9  |  |  |  R ← generated fake tuples
10 |  |  |  foreach r ∈ R do
11 |  |  |  |  if r not in H then
12 |  |  |  |  |  Insert r into H, Initiate Marker
13 |  |  |  /*Using Infer Rules Condition to Check*/
14 |  |  |  foreach i ∈ {1, 2} do
15 |  |  |  |  if t.header_i is C then
16 |  |  |  |  |  if t.T_i in HT but Not Marked then
17 |  |  |  |  |  |  Mark the Entry Satisfied
18 |  /*Count tuples in case 1 at the client side*/
19 |  n' ← |T_{1c} ⋈ T_{2c}|
20 |  if n ≠ n' OR ∃ one entry in H not Satisfy then
21 |  |  ALARM ATTACK
22 end
```

Insertion

To ensure that tuples are really inserted into the database, an intuitive auditing scheme is to insert some fake tuples along with the real insertions. As a malicious attacker cannot distinguish a fake tuple from a real tuple being inserted, we can provide probabilistic integrity assurance for insert operations.

In our approach, the client maintains the number of points in each cell of the grid. For each cell, if the increase of its density reaches a certain amount, insertion of fake tuples is triggered. Additional fake tuples are generated by the same deterministic approach. Without loss of generality, we use linear functions as an example to illustrate our approach.

As described in Section 5.4, using a linear function, fake tuples generated for each cell uniformly distribute on a line segment defined by the two points $\vec{s}$ and $\vec{e}$ in the cell. Assuming the cell currently has k fake tuples, we know that the gap between two neighboring fake tuples on the line segment is $|\vec{\zeta}|$, where $\vec{\zeta} = \frac{\vec{e}-\vec{s}}{k}$. In other words, each fake tuple in the cell can be represented by $\vec{s}+\vec{\zeta}\cdot i$, where $0 \leq i < k$.

Our goal is to add new fake tuples without affecting the current fake tuples. To increase the number of fake tuples on the line segment, we shrink the gap $\vec{\zeta}$ into $\vec{\zeta}'$, and at the same time, we ensure the set of fake tuples defined by the new gap $\vec{\zeta}'$ contains those generated by the old gap $\vec{\zeta}$. That is, we want to ensure $(\forall i)(\exists i')(\vec{s} + \vec{\zeta} \cdot i = \vec{s} + \vec{\zeta}' \cdot i')$, which leads to $i' = \frac{\vec{\zeta}}{\vec{\zeta}'} \cdot i$. Since i' is an integer, it must be true that $\vec{\zeta} \equiv 0 (mod\ \vec{\zeta}')$. In other words, if we choose $\vec{\zeta}'$ that can divide $\vec{\zeta}$ exactly, we can guarantee the old fake tuples can remain in the cell when new *gap* can also regenerate the set of previous generated fake tuples. Thus, our insertion auditing scheme imposes only neglectable overhead.

We simply choose $\vec{\zeta}' = \frac{1}{2} \cdot \vec{\zeta}$. Assume we have k fake tuples

previously in a grid cell g. We generate the additional k fake tuples, and insert them into the outsourced database, and modify the current number of fake tuples of g from k to $2k$.

If there are more than one client, we must ensure the other clients know about the change. Once the inserted fake tuples are in the database (at first, only the client that initiates the insertion can audit the insertion), they will show up in queries issued by other clients. To inform other clients of the change, we extended the header of the newly inserted fake tuples to include some new information, $E(2k \oplus t)$, which means the number of fake tuples in the grid cell that contains the current fake tuple has increased to $2k$ since time t. With the propagation of the information, any client will be auditing with the up-to-date information about the fake tuples.

Deletion

There are two issues with deletions. First, a delete operation may remove fake tuples in the outsourced database if fake tuples are in the range of the delete operation. The problem has a straightforward solution because fake tuples are generated by deterministic functions and the client has full knowledge of their generating mechanism. Before sending out delete operations to the service provider, the client can modify the delete statements to exclude fake tuples.

Second, we need to audit whether delete statements are truthfuly executed by the service provider. In the same spirit of auditing insertion queries, we can first remove some fake tuples from a cell, and then check whether they are indeed deleted in the database. We describe the details of the audit below.

Removing fake tuples in a grid cell is equivalent to *stretching the gap* between neighboring fake tuples. Moreover, in order not to make the set of fake tuples defined by the new gap totally different from the old fake tuples (which means we have to delete all the old fake tuples in the grid and insert the new fake tuples generated under the new gap), we ensure that fake tuples defined by the new gap $\vec{\zeta}'$ form a subset of the fake tuples defined by the old gap $\vec{\zeta}$. That is, we want to ensure $(\forall i')(\exists i)(\vec{s}+\vec{\zeta}\cdot i = \vec{s}+\vec{\zeta}'\cdot i')$. Similar in shrinking the gap, it leads to $\vec{\zeta}' \equiv 0(mod\ \vec{\zeta})$. In other words, the new gap $\vec{\zeta}'$ should be divided by $\vec{\zeta}$ exactly. We simply choose $\vec{\zeta}' = 2 \cdot \vec{\zeta}$, which indicates that we have a half of fake tuples now in the corresponding grid cell.

After the stretching, we can form the additional delete operations to remove the deleted fake tuples from the outsourced database, and then modify the current number of fake tuples in the corresponding grid from k to $\lfloor \frac{k}{2} \rfloor$.

8. DISTRIBUTION-GUIDED APPROACH

The feature space is partitioned into a grid of cells. In high dimensional feature space, it may introduce two problems. First, most cells will be empty and hardly queried. Fake tuples generated into these cells are "wasted". Second, maintaining a high dimensional grid at a client may prove to be impractical. We address these problems in this section.

8.1 Histogram-Based method

We provide integrity assurance by analyzing the fake tuples in the query result. The more fake tuples show up in the query result, the better assurance we can provide. This result is revealed by Eq 4.3. On the other hand, the more fake tuples we add into the outsourced database, the higher the cost of storage and query processing. The question is then the following: where should we put the fake tuples in the feature space so that they have higher probability to be queried?

In our approach, we generate fake tuples that distribute uniformly in the feature space. If queries also distribute uniformly in the feature space, then every fake tuple has equal probability to be queried. Thus, the uniform distribution of the fake tuples maximizes the overall quality of integrity assurance.

However, queries may not distribute uniformly in the feature space. For example, in many real applications the distribution of the query follows that of the data, which means more queries are asked in denser regions. Assume the probability that a certain low density region being accessed by a query is close to 0, then fake tuples generated into this region are useless for providing integrity assurance, because they will not show up in query results.

Our grid-based tuple generation scheme can easily be adjusted in accordance with any query distribution. Without loss of generality, let us assume the query distribution follows the distribution of the data. We show how our scheme makes use of the distribution information.

For each grid cell, we record its density, that is, the number of tuples in the grid. The density information reflects the data distribution, and in our case, the query distribution as well. If the number of fake tuples we generate for each grid is proportional to its density, then we guarantee distribution match at the grid level. In Figure 6, a 2-dimension data space is divided into a 4×4 grid. We count the number of tuples in each grid cell, and generate fake tuples using the deterministic method in an amount proportional to the tuple counts.
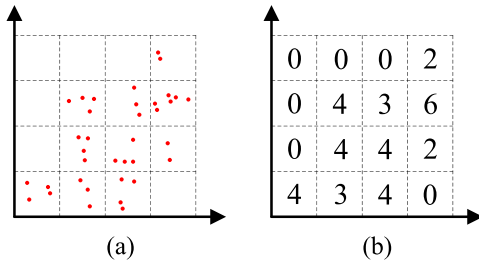

Figure 6: Divide Data into Grid of Buckets

We show the advantage of the distribution-guided approach using an experiment. We use the $lineitem$ table in the 1GB TPCH data [17], and we randomly generate 10 batches of 100 test queries using TPCH query $Q6$ as a template, which is a range query on three attributes $l\_shipdate$, $l\_quantity$, and $l\_extendedprice$. Each query is a "unit" query in the sense that its range has the same volume as that of a grid cell. For the randomized approach, we generate fake tuples randomly in the whole feature space. For the distribution-guided approach, we divide the feature space of the 3 attributes into a $10 \times 10 \times 10$ grid, for each cell in the grid we generate fake tuples in an amount that matches its density. The 1000 test queries are generated in such a way that the centers of their query range follow the distribution of the data.

The experiment result in Figure 7 validate that the distribution guided approach has more fake tuples covered in average when query distribute following the distribution of original data.

8.2 Optimize the histogram structure

For high-dimensional data, the grid becomes very sparse and many cells have very low density and contain no fake tuples. Such cells are useless for integrity audit, but the grid itself may introduce huge storage overhead at the client side. It is thus necessary to shrink the grid structure.
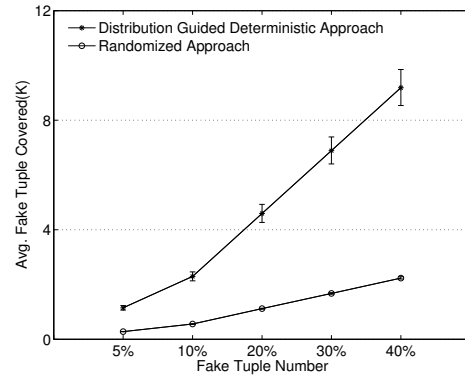

Figure 7: Fake Tuple Coverage

Instead of partitioning the feature space into grids which may contain lots of useless cells, we create cells in an incremental way: we first divide the space into a small number of cells, and then iterately split the cells whose density is above a upperbound threshold θ_h, and eliminate cells whose density is below a lowerbound threshold θ_l.

As an example, let $\theta_l = 12\%$ and $\theta_h = 24\%$ for the data shown in Figure 6. In the first iteration, we divide the feature space on dimension X into 4 cells: one cell's density is between θ_1 and θ_h, and the other three all have density larger than θ_h, so we divide the three cells into smaller cells on dimension Y in the second iteration. Those cells with density smaller than θ_l (shown as dotted circle in Figure 8) are deleted.
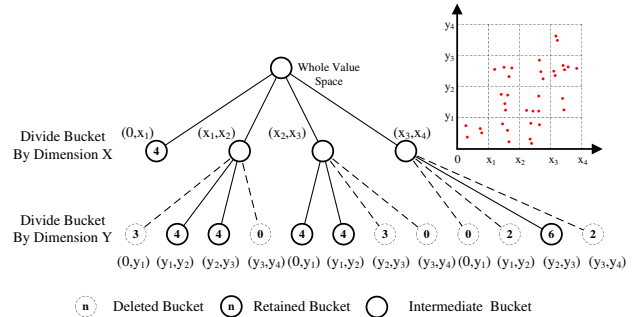

Figure 8: Iterative Gridding

In our optimization, we split the dense cells along one attribute at a time. Since our goal is to reduce the number of cells, a heuristic is to pick the attribute that has the most skew distribution to split. The reason is that dividing an attribute with uniform value distribution creates less lower opportunity to eliminate low density cells. In our approach, we simply use *variance* as a measure of the skewness of the attributes.

Finally, to audit query *completeness*, we need to distinguish the cells that are partially covered by the query from the cells that are fully covered by the query. Since the cells are no longer regular, coverage analysis becomes more difficult. Thus, we are actually trading efficiency for storage. However, in the database outsource scenario, the clients are often resource limited devices, which means reducing storage overhead is more important.

9. EMPIRICAL STUDIES

In this section, we evaluate the security assurance and performance overhead of our integrity auditing scheme.

9.1 Experiment Environment

We use a Pentium IV D 2.8GHz PC with 512MB RAM and a 160GB SCSI hard disk as the server, and Pentium IV 1.6GHz PCs with 256MB RAM and 40GB hard disks as clients. Clients and the server are connected with a local Ethernet network running at 100MBps. We use IBM DB2 v8.2 to store data on the server side. Code on the client side is developed in JAVA with the JDK/JRE 1.5 develop kit and Runtime Library.

Data setup. The data we used in our experiment is derived from the TPC-H benchmark [17], which models decision support systems that store large volumes of data and process queries of a high degree of complexity.

For auditing simple range queries, we generate a TPC-H benchmark database with a scale factor of 1 which has 1GB data. We use the *lineitem* table for our experiment on range queries. This table has roughly 6 million records. While the *lineitem* table has many attributes, we are particularly interested in three numerical attributes: $l\_shipdate$, $l\_extendedprice$ and $l\_quantity$. We tailor the data in the table by extracting the values of the three attributes along with a tid value for each tuple. We form a derived table defined as $T(tid, l\_quantity, l\_extendedprice, l\_shipdate)$, which is then encrypted using the scheme discussed in Section 3 to be stored at the outsourced database server.

We create another TPC-H benchmark database with a scale factor of 0.1 for our experiment on join operations. We use table *customer* and *orders* for our experiment and also experiment with range queries. We encrypt and store the tailored tables *customer(tid, c_custkey), orders(tid, o_orderkey , o_orderdate)* at the server side.

Fake tuple setup. In our experiment, we assume that the queries distribute according to the distribution of original data. As discussed in section 8, we use a *Equal-Width Histogram* to estimate the original data's distribution. (For the *lineitem* table. We divide it into $10 \times 10 \times 10$ grids using the three attributes $l\_quantity$, $l\_extendedprice$, and $l\_shipdate$, and similarly for the other two tables *customer* and *orders*). And we use the grids to guide the process of fake tuple generation and integrity auditing with *linear function* as discussed in Section 5.

Query setup. The type of queries for experiment with range query is derived from TPC-H benchmark query $Q6$, which is shown as follows:

```
SELECT *
FROM lineitem
WHERE l_shipdate BETWEEN ':d₁' AND ':d₂'
    AND l_extendedprice BETWEEN ':p₁' AND ':p₂'
    AND l_quantity BETWEEN ':q₁' AND ':q₂'
```

The quoted variables are all template parameters, and in our experiments, we generate 100 *Unit Queries* which have the same range as a grid in the *Histogram* by changing these template parameters. And to ensure that the set of queries following original data's distribution, we select the center of the query range(which can be represented by $(\frac{d_1+d_2}{2}, \frac{p_1+p_2}{2}, \frac{q_1+q_2}{2})$) according to the distribution information from the $Histogram$(By signing a higher probability to fall into a grid/bucket of the $Histogram$ which has higher density).

The type of queries for experiment with join is derived from

TPC-H benchmark query $Q3$, which is in the following form:

```
SELECT *
FROM customer, orders
WHERE c_custkey = o_custkey
    AND o_orderdate BETWEEN ':d₁' AND ':d₂'
```

The template parameters d_1 are selected randomly in the domain range of $o\_orderdate$, and we guarantees the value of $d_2 - d_1$ equals the range of a grid on dimension $o\_orderdate$ to form a *Unit Selection Join Query*.

9.2 Compare with existing methods

We first compare our method with existing integrity auditing schemes: Merkle Hash tree [13] and Challenge Token [15].

Our scheme is more practical than the existing schemes because our method is *server transparent* – we do not require changes of the DBMSs of the service provider, while the Merkle Hash tree based integrity auditing scheme need the server to maintain a Merkle Hash tree for the data and the Challenge Token based scheme need the service provider to be aware of the Challenge/Answer protocol. This *transparency* of our integrity auditing scheme makes it more easily deployable in database services.

In the experiment, we implemented the Merkle Hash tree as an in memory tree structure with a fan-out of 50, and we use the most commonly used public digital signature scheme RSA [14] as the signature function, and MD5 [2] as the hashing function. For the Challenge Token based scheme, we divide the data into 10 segments and use MD5 Hashing [2] to generate the challenge token. In our experiment, we use an one dimension table and we generate batches of simple range queries with same value range length.
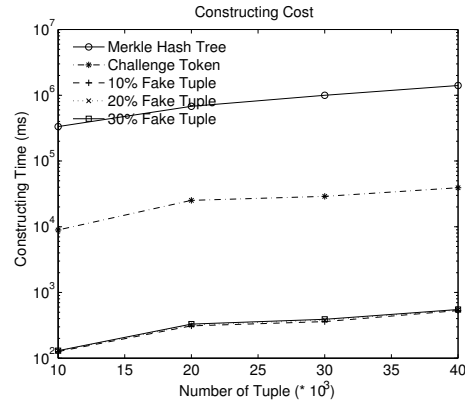
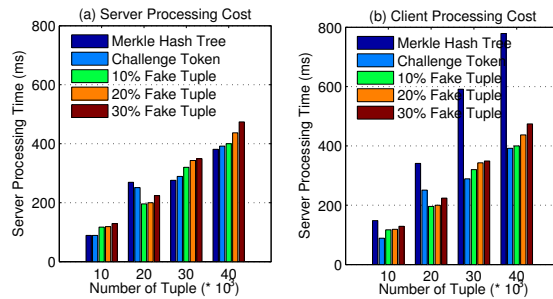**Figure 9: Analysis of Setup Cost**

Figure 10: Server & Client Cost Analysis

From Figure 9, we see that our integrity auditing scheme has a

very low setup cost because we only need to select the deterministic functions and use them to generate the fake tuples. The construction cost of Merkle hash tree is very high because lots of signatures and hash functions must be computed. The construction time for Challenge Token is also very high because for each client we need to pre-compute the hash value for each query that may need to be sent to the server, which further makes this scheme not practical in our application scenario where the clients are only small devices with limited computing and storage resources.

The computation cost on the server shown in Figure 10(a) indicates that our scheme poses little additional cost on the service provider, if any. Indeed, the only overhead is additional scan cost of the fake tuple. But, with our deterministic method, as shown in Figure 10(b), we reduce the client's computing cost, which is very important for our outsourcing database scenario where the clients are often small devices.

9.3 Simple selection query experiment

Security evaluation. As the *correctness* aspect of integrity has been guaranteed by the special *header* column which is discussed extensively in Section 3, we focus on analyzing the *completeness* aspect.

We simulate *data deletion attacks* by randomly deleting m tuples from the original data. The only chance that an attacker can avoid being caught is when none of the m tuples is a fake tuple. For each m range from 1 to 30, we repeats the random deletion 100 times using 5 different random seeds. The averaged results and confidential interval is shown in Figure 11(a). From the figure, we can easily find that the escape probability decreases sharply as the number of deletion increases. And specifically, with only 10% fake tuples generated, we can guarantee an escape probability lower than 25% if more than 20 tuples are deleted, which is a small number given that the total number of tuples is 6 million.
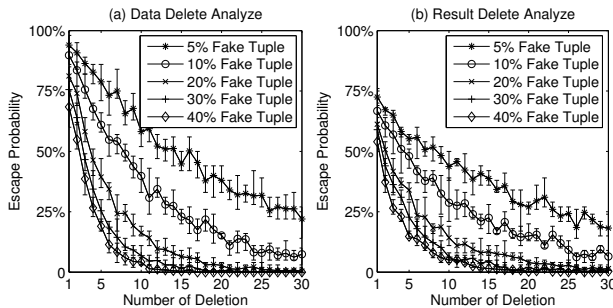

Figure 11: Security Evaluation

We simulate *query result deletion attacks* by randomly deleting m tuples from each of the 100 test *unit queries'* results. For each m range from 1 to 30, we repeat the experiment 5 times and find the average probability of escaping detection among the 100 queries ($\frac{\text{Auditing Failed Query Number}}{\text{Total Query Number}}$) and the confidential interval. The experiment results are shown in Figure 11(b). We can see from the curves in Figure 11(b), the probability of escaping from being caught with a deletion attack approaches 0 rapidly as the number of deletion increases. Additionally, by varying the percentage of fake tuples from 5% to 50%, we can see from Figure 11(b) the more fake tuples we have for the data the lower the escaping probability, which coincident with our intuition.

Client performance analysis. We compare the client side performance in the following two settings: (i) just as in the randomized

approach, storing the set of fake tuples in a relation table at the client side; (ii) Instead of explicitly storing all the fake tuples, auditing the integrity against the deterministic function F. We repeat the 100 *Unit Queries* at the client side 5 times and get the average which is shown in Figure 12. It can be see easily that auditing the deterministic function cost nearly nothing compared to auditing the table, which have a cost grow linearly with the increase of fake tuples. So the experiment validates our scheme's efficiency in integrity auditing, which may benefit our scheme in the outsource database scenario, as in such scenario we may have some mobile devices as the client which have very limited computation capability.
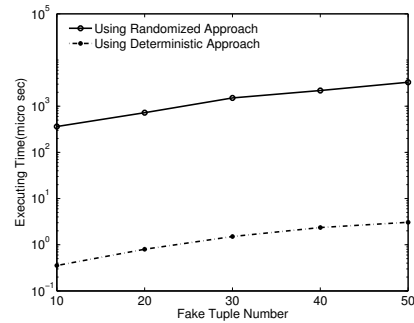

Figure 12: Client Side Performance Analysis

Server performance analysis. Using our scheme for integrity auditing will introduce query overhead at the server side as the additional fake tuples need to be processed as discussed in Section 5.5. In the experiment, we vary the percentage of fake tuples from 5% to 50%. By submitting the 100 unit queries to the server, we collect the average processing cost of these queries using the performance monitor feature of DB2, we repeat the experiment 5 times and get an averaged result shown in Figure 13.

Moreover, in this experiment, we compared the cost for three cases: (i) Without integrity auditing scheme, which is shown as the thick black line; (ii) Without fake tuples but having the authentication header and encryption scheme enabled, which is shown as the thick red line; (iii) All features of the integrity auditing scheme are enabled, which is shown as the bar chart. The result is shown in Figure 13. From this figure, we can easily figure out that the cost increases slowly as we increase the number of fake tuple stored at the server side. And the additional fake tuples will not cause great degradation of performance at the service side, which satisfies our motivation to lower the cost at the server side.
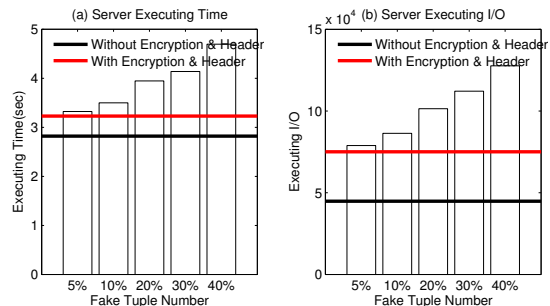

Figure 13: Server Side Performance Analysis

9.4 Join query experiment

We analyze the security of join auditing scheme by randomly deleting from the 100 random join query's result m tuples and computing the average escape detection probability among the 100 queries.

As described in Section 7.1, we have two types of information which can be utilized to audit a join query's result: (i) the set of fake tuples generated using the deterministic function; (ii) join attribute value from a result tuple. In the experiment, we compare the security performance when using only (i) with that when using both (i) and (ii). We repeat the experiment 5 times with the number of fake tuples ranging from 10% to 40% of original data. The averages are shown in Figure 14. From the Figure, we can easily find out that using the information given by the join result tuple can greatly increase the security level.
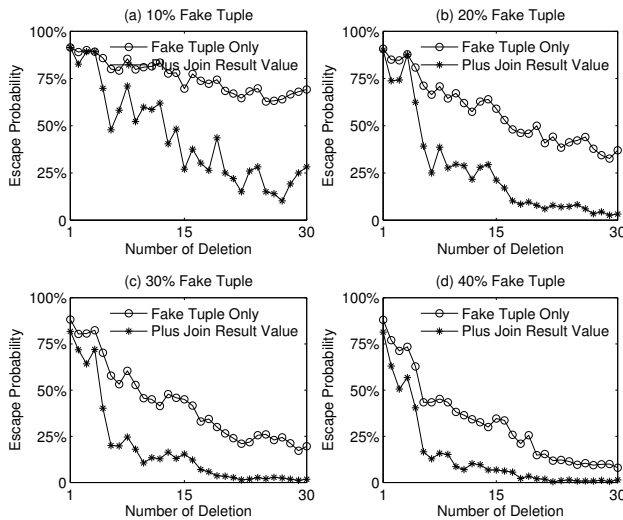

Figure 14: Join Security Analysis

Moreover, the additional overhead at the client side when utilizing the information from both case (i) and case (ii) will not cause a great performance degradation at the client side as being shown by Figure 15.
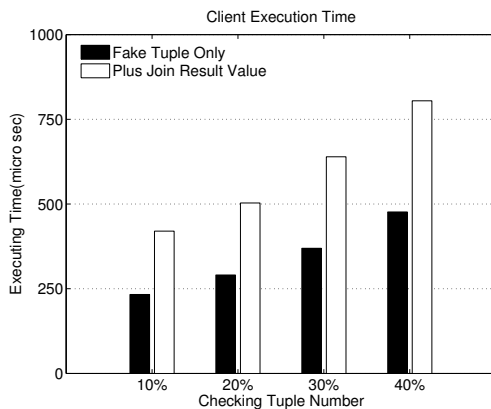

Figure 15: Join Efficiency Analysis

10. CONCLUSION

IT outsourcing has become critical to business operations and vital for businesses to sustain their competitive advantages. Maintaining security in IT outsourcing is important for maintaining the growth of IT outsource services. As data processing is among the most important components of IT services, we address the problem of how to audit the integrity of database services in the paper. Previous approaches can only be effective when the verifier or the user of the service maintains a copy of some outsourced data. However, the overhead in maintaining such a copy undoes the benefit of database outsourcing. Our approach uses deterministic functions to embed fake tuples in the outsourced data. By simply keeping track of the definition of the deterministic functions, the client keeps track of all the fake tuples in the outsourced data, which enables efficient auditing of the query integrity of the service provider.

11. REFERENCES

[1] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order-preserving encryption for numeric data. In *SIGMOD Conference*, pages 563–574. ACM, 2004.

[2] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk. Cryptographic hash functions: A survey. Technical Report 95-02, Department of Computer Science, University of Wollongong, 1995.

[3] Mihir Bellare. Practice-oriented provable-security. In *ISW*, volume 1396 of *Lecture Notes in Computer Science*, pages 221–231. Springer, 1997.

[4] Mihir Bellare, Anand Desai, E. Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *FOCS*, pages 394–403, 1997.

[5] D. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[6] Premkumar T. Devanbu, Michael Gertz, Charles U. Martel, and Stuart G. Stubblebine. Authentic third-party data publication. In *DBSec*, volume 201 of *IFIP Conference Proceedings*, pages 101–112. Kluwer, 2000.

[7] Hakan Hacigümüs, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *SIGMOD Conference*, pages 216–227. ACM, 2002.

[8] Hakan Hacigümüs, Sharad Mehrotra, and Balakrishna R. Iyer. Providing database as a service. In *ICDE*, pages 29–. IEEE Computer Society, 2002.

[9] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD Conference*, pages 121–132. ACM, 2006.

[10] Ralph C. Merkle. A certified digital signature. In *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989.

[11] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Authentication and integrity in outsourced databases. In *NDSS*. The Internet Society, 2004.

[12] HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee Tan. Verifying completeness of relational query results in data publishing. In *SIGMOD Conference*, pages 407–418. ACM, 2005.

[13] HweeHwa Pang and Kian-Lee Tan. Authenticating query results in edge computing. In *ICDE*, pages 560–571. IEEE Computer Society, 2004.

[14] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[15] Radu Sion. Query execution assurance for outsourced databases. In *VLDB*, pages 601–612. ACM, 2005.

[16] Douglas R. Stinson. *Cryptography, Theory and Practice*. CRC Press, 1995.

[17] TPC-H. *Benchmark Specification*. http://www.tpc.org.