A Comprehensive Framework for Secure Query Processing on Relational Data in The Cloud

Shiyuan Wang, Divyakant Agrawal, and Amr El Abbadi

Department of Computer Science, University of California at Santa Barbara
{sywang, agrawal, amr}@cs.ucsb.edu

Abstract. Data security in the cloud is a big concern that blocks the widespread use of the cloud for relational data management. First, to ensure data security, data confidentiality needs to be provided when data resides in storage as well as when data is dynamically accessed by queries. Prior works on query processing on encrypted data did not provide data confidentiality guarantees in both aspects. Tradeoff between secrecy and efficiency needs to be made when satisfying both aspects of data confidentiality while being suitable for practical use. Second, to support common relational data management functions, various types of queries such as exact queries, range queries, data updates, insertion and deletion should be supported. To address these issues, this paper proposes a comprehensive framework for secure and efficient query processing of relational data in the cloud. Our framework ensures data confidentiality using a *salted IDA* encoding scheme and *column-access-via-proxy* query processing primitives, and ensures query efficiency using matrix column accesses and a secure B+-tree index. In addition, our framework provides data availability and integrity. We establish the security of our proposal by a detailed security analysis and demonstrate the query efficiency of our proposal through an experimental evaluation.

1 Introduction

Cloud computing has been gaining interests in the commercial arena due to its desirable features of scalability, elasticity, fault-tolerance, self-management and pay-per-use. On the other hand, the security of sensitive data stored in the cloud remains a big concern, and even a road block to the enterprise usage of the cloud for relational data management and query processing. The cloud, unfortunately, has some unique security problems due to the shared environment it creates, even if access control policy and authentication are in place. For example, a recent vulnerability found in Amazon EC2 allows crossing virtual machine boundary and gaining access to another tenant's data co-located on the same physical machine [1]. Many enterprises therefore question about whether adequate security can be ensured for storing and processing their relational data in the cloud.

To solve the above concern of data security, data confidentiality is a major challenge. Although encryption is often used to protects data, encryption itself is insufficient to guarantee data confidentiality, even if the encryption scheme does not reveal any characteristics about the plaintext data and thus can resist statistical analysis on encrypted data. When the encrypted data is frequently accessed for serving clients' queries, any potential information leakage should also be controlled and minimized, since attackers may infer the plaintext data from clients' accessed positions on the encrypted data. Many existing proposals on processing queries on encrypted data did not consider both confidentiality for data residing in storage and for data being accessed by queries [2, 3, 4, 5, 6, 7]. Some of them are only able to support one or two types of queries on encrypted data, and do not even support other data processing operations such as data updates [3, 4]. Although powerful cryptographic techniques such as homomorphic encryption [8] and Private Information Retrieval [9] can overcome some of the confidentiality drawbacks, they are computationally expensive and can adversely impact both latency and throughput.

In addition to data confidentiality, the availability and integrity of data in the cloud should also be considered. *Information Dispersal Algorithm* (IDA) [10] and the similar error-correcting codes [11] are used in recent work [10, 12, 13] to provide data availability, and are commercialized [14]. A recent trend in industry even consider IDA as an alternative to the traditional data encryption [15], since IDA provides both data availability and certain degree of data confidentiality.

Our goal in this paper is to provide a comprehensive secure query processing framework that addresses the issues of data confidentiality, availability and integrity, and supports practical processing of various queries on the relational data in the cloud. We achieve confidentiality for data residing in storage using a modified scheme to IDA, called *"salted" IDA* (Section 4). Salted IDA relies on randomness to improve data confidentiality of the original IDA scheme against computation bounded adversaries and relies on the original IDA scheme to provide data availability. We achieve confidentiality for data dynamically accessed by queries by transforming query requests to single operations called *column-access-via-proxy* (Section 5), so that different queries and queries among different clients are not likely to be differentiated. We discuss the security of these two schemes in a security analysis (Section 7).

To enable practical query processing, we build a secure B+-tree index [16] on frequently queried attributes. We encode and disperse the index and the data tuples into matrix column pieces using salted IDA, and access the index and the tuples using column-access-via-proxy operation. During query processing, a client retrieves and decodes only a small part of the index, based on which information she locates the candidate answer tuples. We further boost query performance by caching partial index on the client. Caching index also helps improve data confidentiality at accesses by confusing inferences on the index traversal paths. Thus, we are able to support common relational database queries such as exact queries, range queries and data updates with consistent security guarantees (Section 6). Our experimental evaluation indicates the query performance of our framework is practical, i.e. it can process an exact query within 1 milliseconds, and a range query within 200 milliseconds on a data table of size 10^7 (Section 8).

2 Related Work

To support database style queries on encrypted relational data, previous proposals designed techniques to directly filter or process encrypted data. However, they did not achieve a good tradeoff of data confidentiality and query efficiency. For example, the methods that attach range labels to encrypted data [2, 3] reveal the underlying data distributions. Methods relying on order preserving encryption [4, 17] reveal the data order. These methods cannot overcome attacks based on statistical analysis on encrypted data. On the other hand, homomorphic encryption is secure and enables calculation on encrypted data [18, 8], but it relies on expensive public key cryptosystem and thus is not practical.

Instead of direct filtering or processing encrypted data, an alternative is to use an encrypted index which allows the client to traverse the index and to locate data of interests in a small number of rounds of retrieval and decryption [6, 7, 5]. However, previous works using encrypted index did not provide satisfactory data confidentiality. For example, deterministic symmetric key encryption (e.g. 3DES) is used in [6], revealing the underlying data distribution. Each value of the index entries is encrypted in [7], revealing the index structure and consequently revealing client access pattern. Although confidentiality for data residing in storage is proved in [5], data confidentiality in a dynamic query access environment is not guaranteed. Recent work studied obfuscating query access patterns [19] for data outsourced in the cloud, but it still incurs a lot extra computation and communication costs and requires a special secure hardware, while we only bring additional communication costs by routing low level data requests via proxies. Our work is more comprehensive in that we provide data confidentiality, availability and supports flexible queries and data updates.

3 System and Attacker Model

3.1 System Model

Data Model. We consider a relational table D with N tuples. Each tuple t has d attributes, $A_1, A_2, ..., A_d$. An index I is built on the frequently queried attributes of D, such as the primary key. Without loss of generality, we refer to I as a one-dimensional index with one-to-one mapping to the tuples in D. We assume each attribute value (and each index key) can be mapped to an integer in the range of $[1, ..., MAX]$.

Data Storage Model. The tuples and the index are encoded under separate secret keys C and then stored on n servers, $S_1, S_2, ..., S_n$, hosted by cloud storage providers such as Amazon EC2. The same keys C are used for decoding the tuples and the index retrieved from servers. The tuples and the index are only accessible to the clients who own the data or the trusted partners of the clients (partners are also referred to as clients hereinafter).

Data Access Model. We assume that multiple clients always issue multiple queries on a data table in the same time period, e.g. in working time. We process exact queries, range queries and tuple updates given index keys as predicates. We also process tuple insertion and deletion.

3.2 Attacker Model

Attacker and Prior Knowledge Assumptions. We consider attackers are external entities or the servers which store the data. We do not deal with insider attacks, such as from malicious partners. We assume client machines are safe, thus any confidential information of the client such as the secret key C is not known to attackers. Attackers do not know clients' queries. However, attackers could know clients' data distribution and even some exact data values and their occurrence frequencies. We assume attackers' computations are bounded by polynomial size circuits.

Attacks. We consider two types of attacks: (1) attacks that try to compromise data confidentiality without compromising data availability or integrity; (2) attacks that modify the encoded tuples or index keys (compromising data integrity), or conduct Denial-of-Service (DoS) attacks to bring down servers (compromising data availability). We say servers are *faulty* in Type (2) attacks. In Type (1) attacks, attackers can compromise any number of servers (or any number of servers can collude) to analyze the encoded data. They can monitor index and data accesses, and perform inference or linking attacks [6], in which they try to infer the correspondence between the positions of encoded data in storage and plain-text values in the data domain, or even try to infer the secret key C.

4 Data Encryption and Dispersal by "Salted" IDA

We leverage Information Dispersal Algorithm (IDA) [10] for providing data confidentiality and availability. We propose an easy-to-use data encoding and dispersal scheme called *salted IDA* based on the original IDA scheme.

4.1 Information Dispersal Algorithm (IDA)

We first introduce the original IDA scheme [10]. IDA *encodes and disperses data into n uninterpretable pieces so that only m ($m \leq n$) pieces are required to reconstruct the data, and the total storage size of the dispersed pieces is only n/m times of the data size*. Consider that n pieces are distributed onto n servers, then IDA can tolerate up to $(n - m)$ faulty servers for data retrievals. Table 1 summarizes the notations we use in the paper.

Given a matrix M, let $M_{i,:}$ be its ith row, $M_{:,i}$ be its ith column, and $M_{i,j}$ or M_{ij} be the entry at the ith row, jth column of M. Consider an $m \times w$ data matrix D. Each entry in D is an integer in a finite field $GF(2^s)$, or a residue mod $B = 2^s$. The following data values and arithmetic operations are on $GF(2^s)$. To encode and disperse D, IDA uses an $n \times m$ information dispersal matrix C, in which *every m rows are linearly independent, or any submatrix C^* formed by any m rows of C is invertible,* e.g. in $GF(2^4)$, $C = \begin{pmatrix} 1 & 3 & 5 \\ 1 & 4 & 3 \\ 1 & 5 & 2 \\ 1 & 6 & 7 \\ 1 & 7 & 6 \end{pmatrix}$.

Let the encoded data matrix be $E = C \cdot D$, then each row of E, $E_{i,:}$ ($1 \leq i \leq n$), is a dispersed piece stored on a server. To reconstruct D, we collect m dispersed pieces, corresponding to m rows of E. Let these rows form an $m \times w$ submatrix of E, E^*. Keep the corresponding m rows of C to form an $m \times m$ submatrix of C, C^*. Then

Table 1. Table of Frequently Used Notations

Notation	Description
n	number of dispersed data pieces (number of servers to distribute the data)
m	threshold number of pieces to recover the data (threshold number of servers to retrieve the data)
N	number of data tuples or keys
d	number of attributes in one tuple
C	$n \times m$ secret key matrix
ID, TD	plaintext index matrix, data tuples matrix
IE, TE	securely encoded index matrix, data tuples matrix
$E_{i,:}$, $E_{:,i}$	ith row, ith column of matrix E
E^*	$m \times m$ sub matrix obtained by deleting rows in E
b	number of branches in a B+-tree index node
col	column address pointing to a column in a matrix
key	key in a B+-tree index node

$$D = {C^*}^{-1} \cdot E^* \tag{1}$$

For example in $GF(2^4)$, consider a matrix $D = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$. Using the above information dispersal matrix C, we get

$$E = C \cdot D = \begin{pmatrix} 1 & 3 & 5 \\ 1 & 4 & 3 \\ 1 & 5 & 2 \\ 1 & 6 & 7 \\ 1 & 7 & 6 \end{pmatrix} \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} = \begin{pmatrix} 8 & 6 & 7 \\ 12 & 9 & 9 \\ 13 & 10 & 8 \\ 4 & 8 & 8 \\ 5 & 11 & 9 \end{pmatrix}$$

We distribute five rows $E_{1,:}, E_{2,:}, ..., E_{5,:}$ onto five servers $S_1, S_2, ..., S_5$ respectively. If S_2 and S_3 are faulty, we obtain $E_{1,:}$, $E_{4,:}$ and $E_{5,:}$ from S_1, S_4 and S_5 to form E^*. We then delete $C_{2,:}$ and $C_{3,:}$ from C to form C^*, and reconstruct D using Equation (1).

$$D = {C^*}^{-1} \cdot E^* = \begin{pmatrix} 1 & 3 & 5 \\ 1 & 6 & 7 \\ 1 & 7 & 6 \end{pmatrix}^{-1} \begin{pmatrix} 8 & 6 & 7 \\ 4 & 8 & 8 \\ 5 & 11 & 9 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

4.2 "Salted" IDA

IDA ensures data availability, but does not ensure adequate data confidentiality. An encryption scheme with adequate confidentiality should be resistant to statistical analysis on a set of encrypted data. That is, the encrypted data set should not reveal any characteristics of the corresponding plaintext data set. Note that simple symmetric encryption such as 3DES reveal the underlying data distribution; order-preserving encryption [4] reveals the data order.

Based on IDA, we propose a scheme called *salted IDA* to achieve such data confidentiality. As in IDA, a client maintains an $n \times m$ secret matrix C as the information dispersal matrix and the keys for encoding and decoding a data matrix D, where n, m

are determined by the client based on the number of servers that she plans to use and the estimated number of non-faulty servers. In addition, the client keeps a secret seed ss, and a deterministic function fs for producing random factors based on ss and the address of data entries on D. We call these random factors *salt*.

Function fs feeds ss into a pseudorandom number generator (PRNG). Before encoding and dispersing D onto n servers using IDA, for each column of D, $D_{:,i}$, the client calls the PRNG procedure i times, sets the last generated random number as the *salt*, and then adds the salt to each data entry of $D_{:,i}$, $D_{j,i}$ ($1 \leq j \leq m$). After decoding the encoded data retrieved from m non-faulty servers, the client reconstructs salts by calling fs and then deducts these salts from the decoded data entries, recovering D. The security of salted IDA is established in Section 7.1.

5 Secure Cloud Data Access

In this section, we present the core design of our secure query processing framework for relational data stored in the cloud.

5.1 Overview

We use salted IDA to encode and disperse the data onto servers in the cloud. Now we consider the problem of query processing on salted IDA encoded matrix. First, we note that in Equation (1), if the encoded matrix E^* is substituted by a column $E^*{}_{:,i}$, we can get the corresponding data column $D_{:,i}$.

$$D_{:,i} = C^{*-1} \cdot E^*{}_{:,i} \tag{2}$$

Similarly we can encode a column $D_{:,i}$ separately as follows.

$$E_{:,i} = C \cdot D_{:,i} \tag{3}$$

For the example data in Section 4.1, if we retrieve $E_{11} = 8$ from server S_1, $E_{41} = 4$ from server S_4 and $E_{51} = 5$ from server S_5 to form a column $E^*{}_{:,1} = (8\ 4\ 5)^T$, we get $D_{:,1} = (1\ 2\ 3)^T$ using Equation (2). However, note that a row of the data matrix cannot be accessed in a similar way by accessing only a row of the encoded matrix.

Using this *column access* property, we can process a query or an update by accessing a few columns at a time. However, selecting which columns to access is still difficult, because searching data on the IDA encoded matrix based on plaintext input is fundamentally difficult. We solve this problem by building a B+-tree index on the key attribute. The index is kept secure and is only known to the client.

Given a table D with N tuples and a B+-tree index I on the key attributes of D, we store D into a tuple matrix TD, and I into an index matrix ID. TD and ID have a fixed column size, m. As a result, each column of TD corresponds to one or more tuples in D. One or more columns of ID correspond to a tree node in I. Each leaf node of I maintains the pointers to the columns of TD where the tuples with the keys in this leaf node are stored. We encode ID into IE and TD into TE, and then disperse IE and TE onto n servers, $S_1, S_2, ..., S_n$, by salted IDA (see Fig. 1). Queries on the index key attribute can be efficiently processed by locating the columns of ID (tree nodes) that store the query keys and then retrieving the corresponding tuples from columns of TD.

5.2 Organization of Index

Let the branching factor of the B+-tree index I be b. Then every node of I has $[\lceil\frac{b-1}{2}\rceil, b-1]$ keys, and every internal node of I has $[\lceil\frac{b}{2}\rceil, b]$ children. We fix the size of a tree node as $2b+1$. Since the column size of the index matrix ID is fixed to m, the ideal case would be $m = 2b+1$, one column for one tree node. We assume the ideal case in the paper for simplicity [1].

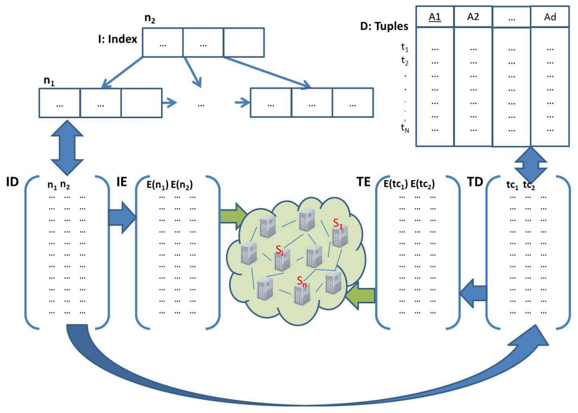


Fig. 1. Secure Cloud Data Access Framework

We assign each tree node an integer column address denoting its beginning column in ID according to the order it is inserted. Similarly, we assign every tuple column of TD an integer column address according to the order its tuples are added into TD. Since the tree is dispersed, these column addresses serve as pointers to the tree nodes.

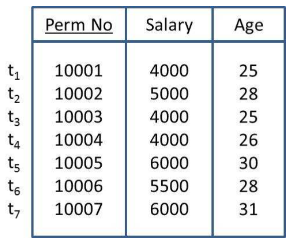
We represent a tree node of I, $node$, or the corresponding consecutive columns in ID, $ID_{:,g}$ as

$$(isLeaf, col_0, col_1, key_1, col_2, key_2, ..., col_{b-1}, key_{b-1}, col_b) \quad (4)$$

where $isLeaf$ indicates if it is a leaf node. key_i is an index key, or 0 if $node$ has less than i keys. For an internal node, $col_0 = 0$, col_i $(1 \leq i \leq b)$ is the beginning column address of the ith child node of $node$ if key_{i-1} exists, otherwise $col_i = 0$. For a leaf node, col_0 and col_b are the beginning column addresses of the predecessor/successor leaf nodes respectively, and $col_i (1 \leq i \leq b-1)$ is the column address of the tuple with key_i.

Given an example *Employee* table shown in Fig. 2, Fig. 3 gives an index (the upper part) built on *Perm No* and the corresponding index matrix ID (the lower part). In the figure, the branching factor $b = 4$, and the column size of ID, $m = 9$. Keys are inserted into the tree in ascending order. The numbers shown on top of the tree nodes are the column addresses of these nodes. The numbers pointed to by arrows below the keys of the leaf nodes are the column addresses of the tuples with those keys. For the root node $ID_{:,2}$, $isLeaf = 0$, $col_0 = 0$, $col_1 = 1$ is the column address of its leftmost child, $key_1 = 10003$, $key_3 = 0$ and $col_4 = 0$ for no third key. For the leaf node $ID_{:,1}$, $isLeaf = 1$, $col_0 = 0$ for no predecessor, $col_1 = 1$ is the column address of the tuple

[1] In the cases of multiple columns representing a tree node, the columns representing a tree node are consecutive, and the number of these columns is fixed as $\lceil\frac{2b+1}{m}\rceil$. Given the column address of the tree node, which is also the column address of the beginning column that represents the node, the column addresses of the following columns can be easily derived from the beginning column address, since consecutive columns have consecutive integer addresses.



	Perm No	Salary	Age
t_1	10001	4000	25
t_2	10002	5000	28
t_3	10003	4000	25
t_4	10004	4000	26
t_5	10005	6000	30
t_6	10006	5500	28
t_7	10007	6000	31

Fig. 2. Employee Table

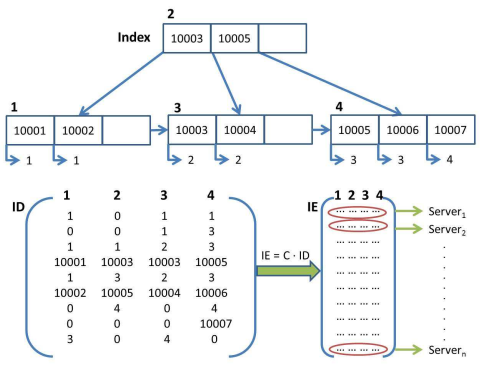


Fig. 3. Index Matrix of Employee Table

with key $= 10001$, $col_3 = 0$ and $key_3 = 0$ for no third key, and $col_4 = 3$ is the column address of the successor $ID_{:,3}$.

5.3 Organization of Data Tuples

To disperse data tuples on the same setting of servers as the index keys, the column size of the tuple matrix TD is also set as m. Initially, to organize the existing d-dimensional tuples of D in TD, we can sort these tuples in ascending order of their keys, and pack every p tuples in a column of TD such that $p \cdot d \leq m - k$ and $(p + 1) \cdot d > m - k$, where k is the size of a secure checksum. The checksum is calculated by applying the *Message Authentication Code* (MAC) [20] on the attribute values of all p tuples, so as to verify the integrity of these tuples returned by servers. The size of the checksum is fixed regardless of the length of the data.

After initialization, a new tuple t is inserted in the last column of TD if the column can accommodate t, or inserted into a new column at the end of TD. Tuples are not stored in the order of their index keys as in the initialization. This approach speeds up tuple insertion and makes it flexible. A deleted tuple is removed from the corresponding column by leaving the d entries it occupied previously empty (the corresponding encoded entries are not empty, but are filled with salts).

5.4 Secure Column Access Via Proxies

In our framework, the client directs query processing, while the servers store or retrieve columns on the index matrix ID and the tuple matrix TD based on the client's decision. Data updates or the initial data uploading need to store columns. Read-only queries only need to retrieve columns.

To store a column of ID (or TD), $ID_{:,i}$ (or $TD_{:,i}$), the client adds salts into the data entries in the column, encodes the column using Equation (3), and disperses it onto n servers. To retrieve $ID_{:,i}$ (or $TD_{:,i}$), the client retrieves m pieces from m non-faulty servers, decodes the assembled column using Equation (2), and deducts salts. The m requests are sent in parallel.

From these column accesses, attackers cannot precisely determine the content of a query or the plaintext data involved. However, attackers can easily learn the initiator client through social engineering attacks, and then infer the client's query and the data accessed in the query. To hide query initiators from attackers, we route column access requests and responses for different clients through a *trusted proxy*, so that attackers cannot even distinguish between different queries sent from different clients. Multiple proxies can be used for load balancing. A client can switch to another proxy whenever needed. We call this scheme *column-access-via-proxy*. Its security guarantee is analyzed in Section 7.1.

6 Query Processing

Our framework supports exact queries, range queries, as well as updates, inserts and deletes. These common queries form the basis for general purpose relational data processing.

Exact Query. To find the tuple t for a given index key x, the client traverses the index downwards from the root. This traversal is similar to the traversal on a traditional B+-tree index, except that retrieving each tree node requires retrieving the corresponding index matrix column. At the end of the index traversal, if the client finds x in a leaf node, the client follows the tuple matrix column address associated with x to locate t, which also needs retrieving the column where t is stored.

Range Query. To find the tuples whose keys fall in a given range $[x_l, x_r]$, the client locates all qualified keys in the leaf nodes of the index, gets the addresses of the tuple matrix columns associated with these keys, and then retrieves the answer tuples from these tuple matrix columns. The qualified index keys are located by performing an exact query on either x_l or x_r, and then following the successor or predecessor links at the leaf level. Since tuples can be dynamically inserted and deleted, and the tuple matrix columns may not be ordered by index keys, we cannot directly retrieve the tuple matrix columns in between the tuple matrix columns corresponding to x_l and x_r. After finding out the qualified index keys and the associated tuple matrix column addresses, we can request the tuple matrix columns in batch to save disk read time on the server.

Tuple Update. Update to a tuple without changing its index key can be done by performing an exact query on the key to get the target tuple column and then storing the updated tuple column.

Insertion and Deletion. Data insertion is done in two steps: tuple insertion and index key insertion. The corresponding columns in the tuple matrix TD and in the index matrix ID need to be updated by re-storing these columns. Data deletion follows a similar process, with the exception that the tuple to be deleted is first located based on the tuple's key. The order that a TD column is updated before the ID column is important, since the column address of the TD column is the link between the two and needs to be recorded in the ID column. Index key insertion and deletion are always done on the leaf nodes, but node splits or merges may be needed to maintain the B+-tree structure. The overhead in these cases is still small, since the number of nodes (columns) to be updated is bounded by the height of B+-tree, $log_b N$.

Boosting Performance and Improving Data Confidentiality at Accesses by Caching Index Nodes on Client. The above query processing relies heavily on index traversals, which means the index nodes are frequently retrieved from servers and then decoded on the client, resulting in a lot of communication and computation overhead. Query performance can be improved by caching some of the most frequently accessed index nodes in clear on the client. Top level nodes in the index are more likely to be cached. We assume the root node of the index is always cached. Caching index can also confuse the inferences that depend on the order of requests to infer the index structure and the data, thus help improving data confidentiality at accesses.

7 Security Analysis

As analyzed below, we provide data confidentiality against polynomial size circuits bounded attackers, even when all servers are controlled and monitored (but they are not faulty) by attackers. We also provide data integrity and availability when no more than $n - m$ servers are faulty.

7.1 On Data Confidentiality

We rely on the definition of *data indistinguishability* [21, 22] to prove the confidentiality of "salted" IDA encoding. *Data indistinguishability* means that *the encryption of any two database tables with the same schema and the same number of tuples should be computationally indistinguishable for any polynomial size circuit*. It is a strong security guarantee in that it invalidates statistical analysis on encoded data, while such attacks would work on simple symmetric encryption such as 3DES or order-preserving encryption [4]. The original IDA scheme [10] does not have such strong security guarantee, e.g. equal plaintext columns would be encoded into equal ciphertext columns, and constructing $m \times m$ correct correspondences between plaintext and ciphertext data could reveal the secret key C. We show in the following that salted IDA achieves data indistinguishability.

Theorem 1. *If the random numbers generated by a pseudorandom number generator (PRNG) are indistinguishable from truly random numbers, $\forall$ two $m \times w$ data matrices D, D' in $GF(2^{32})$, their encryption under salted IDA scheme are computationally indistinguishable.*

Proof. Given that the random numbers generated by PRNG are drawn uniformly from $GF(2^{32})$, each column of a matrix $D_{:,i}$ will be added by a salt which is uniformly distributed in $[1, 2^{32}]$, thus the number of possible choices of salts for each column is 2^{32}. For w columns, the total number of possible choices of salts is 2^{32w}. Since $w > N/(b-1)$, $2^{32w} > 2^{32N/(b-1)}$, which is exponential in N. For a typical database index of $b = 50, N = 10^6$, $2^{32N/(b-1)} > 2^{653061}$. Given such a large choice space of salts and that after adding salt to D and D', they will be encoded with an unknown secret matrix C, the ciphertext matrices E, E' obtained by salted IDA encoding are computationally indistinguishable.

Since the rows of E, E' are distributed onto n servers, two rows from them $E_{i,:}, E'_{i,:}$ are also computationally indistinguishable. Similarly, because of the large choice space of salts, C is unbreakable on polynomial size circuits. However, ensuring the security of the salted IDA encoding scheme itself does not ensure data confidentiality. For example, a target data table can still be located on the servers when its data size or index size is unique. Then attackers could monitor data accesses on its index matrix and infer the keys based on user accessed positions and known index key distribution. We therefore give a definition for a secure relational data processing framework to ensure data confidentiality.

Definition 1. *A* secure relational data processing framework ensures data confidentiality *if it satisfies the following conditions: (1) Data is encrypted under a secure encryption scheme that satisfies data indistinguishability; (2) By observing index accesses on the encrypted data, finding out correct correspondences between the plaintext data and the encrypted data only has negligible advantage over random guesses with prior knowledge on polynomial size circuits.*

We have shown in Theorem 1 that our framework satisfies condition (1). We next show that it satisfies condition (2). It is easy to see that if client data access patterns are clearly revealed to attackers, attackers may figure out the index structure and then perform linking analysis between possible plaintext data and ciphertext data. However, since query processing is performed by *column-access-via-proxy*, a client's data access pattern is obfuscated among multiple clients, and a query's data access pattern is obfuscated among multiple queries.

Lemma 1. *Under multiple clients, multiple queries scenarios with index cache enabled on the client, the best that attackers can get under the column-access-via-proxy scheme is to identify the columns that represent leaf nodes of the index.*

Proof. Under multiple clients, multiple queries scenarios with index cache enabled on the client, all the attackers observe on the index are single and batch column accesses. The exact structure or even the height of the index are not known to the attackers. To them, single column accesses could correspond to internal nodes or leaf nodes for exact and range queries, while batch columns accesses could only correspond to leaf nodes for range queries. In the long term, attackers may be able to identify a large number of leaf nodes and sort some of them in the natural order of key values, but they may not get the total order of all leaf nodes.

Based on Lemma 1, we show our framework satisfies condition (2) of Definition 1.

Lemma 2. *Finding correct correspondences between plaintext keys and encoded leaf nodes only has negligible advantage over random guesses on polynomial size circuits.*

Proof. Assume the exact plaintext key values and the exact order of all the leaf nodes are known. Consider the possible ways of distributing N ordered key values into w ordered leaf nodes with the constraint that each node holds $[\frac{b-1}{2}, b-1]$ keys. Note that only after $node_i$ holds $\frac{b-1}{2}$ keys, can the succeeding node $node_{i+1}$ start to hold keys, which means after $node_i$ holds $\frac{b-1}{2}$ keys, the next $[\frac{b-1}{2}, b-1]$ keys can only be distributed between $node_i$ and $node_{i+1}$, yielding $\binom{\frac{b-1}{2}+1}{1} = \frac{b+1}{2}$ choices. As there are $(w-1)$ pairs of preceding and succeeding nodes in total, the total number of choices is $(\frac{b+1}{2})^{w-1}$. Since $w > N/(b-1)$, $(\frac{b+1}{2})^{w-1} > (\frac{b+1}{2})^{N/(b-1)-1}$, which is exponential in N. For a typical database index of $b = 50, N = 10^6$, $(\frac{b+1}{2})^{N/(b-1)-1} > 2^{27957}$. Given such a large choice space, finding the correct correspondences between plaintext key values and encoded leaf nodes only has negligible advantage over random guesses.

Since our proposed framework satisfies both conditions of Definition 1, we claim the following.

Theorem 2. *The proposed secure relational data processing framework ensures data confidentiality.*

Note that the multiple clients, multiple queries scenarios that we assume in the above are typical in working time, especially when the rent of the cloud resources is related to the time of usage. We do not deal with other scenarios such as single client queries for now, but we suggest requesting redundant columns in a k-anonymity [23] fashion in each request to provide practical data confidentiality at accesses in those scenarios.

7.2 On Data Integrity and Availability

We check the integrity violation on the index structure using the relationships of sorted key values and the relationships of nodes in the index, and check the integrity violation on data values using the checksums. We rely on IDA to provide data availability when no more than $n - m$ servers are faulty.

For the integrity of the index, the keys in a node and among all leaf nodes are sorted, and the keys in parent/child or successor/predecessor nodes have certain relationships. As a result, malicious changes on the encoded index, including *spoofing attacks* which replace an encoded data piece and *splicing attacks* which swap two encoded data pieces, will be detected if these changes violate the relationships between nodes. For the integrity of tuples, the client checks if the checksum stored in the same tuple column as these tuples matches the attribute values of these tuples. If one data piece is found to be corrupted, the client fetches a piece from another server, as long as m servers are not faulty. Similarly, our framework can withstand DoS attacks on up to $n - m$ servers for supporting read only queries. For writes such as data updates, all n servers are required to be not faulty. However, this constraint can be relaxed by using the quorum consensus idea proposed in [24].

8 Experimental Evaluation

Our evaluation focuses on the following: (1) the efficiency of our proposal for processing different queries (exact, range, insert and update); (2) the overhead imposed by security based on comparisons with the *baseline* query processing of no security provided, and with the basic encrypted index approach [6] which does not provide sufficient data confidentiality and provides no data availability; (3) the overhead breakdown in terms of client processing time, server processing time and network latency as well as the communication sizes for index and tuples; (4) the effects of data size, query selectivity and index caching on query performance.

8.1 Implementation and Setup

Implementation. We implemented the *baseline* approach, denoted as *baseline*, the basic encrypted index approach [6], denoted as *encr*, and our approach, denoted as *sida*, in C++. For *baseline*, all processing is done on the server and a plaintext B+-tree index is used. For *encr*, a B+-tree index is stored on the server, with each node encrypted using 3DES. We used Crypto++ Library 5.6.0 [25] for implementing IDA and MAC in *sida* and for implementing 3DES in *encr*. We simulated servers in the cloud by using exactly the same number of local files, and simulated network latency by multiplying the communication sizes with the average internet download speed (5.1Mbps) and the average internet upload speed (1.1Mbps) in a wide area network [26]. To account for the overhead of proxy in *sida*, we doubled the calculated network latency. We implemented the client side index cache for all three approaches for fairness of performance comparison. Given a client desired cache hit rate, we cached most frequently accessed index nodes based on the query workload.

Data Set. We extracted 5 attributes, *I_ID*, *I_A_ID*, *I_RELATED1*,...,*I_RELATED5*, *I_STOCK* and *I_PAGE* from the *Item* table of TPC-W Benchmark [27] to form the test data table and built an index on the primary key *I_ID*. We used a TPC-W data generating tool to generate different sizes of tuple sets.

Setup. We fixed the branching factor of B+-tree index $b = 50$. We used $m = 13, n = 21$ servers for *sida*, and only one server for *baseline* and *encr* respectively. We summarize our experimental parameters in Table 2. For each combination of parameters, we generated 1000 exact queries, range queries, data updates and inserts respectively. A query key was generated by randomly picking a value from the domain of *I_ID* based on Zipf distribution with the specified query skew (default skew=1). For a range query, we used this generated query key as the pivot value, and picked a fixed size query range (query range/selectivity in Table 2) around the pivot value. For an update or insert, the new values of the tuple were generated using the TPC-W tool. The reported results were averaged over 1000 queries of the same type. Experiments were run on Linux servers with Intel 2.40GHz CPU, 3GB memory and Federal Core 8 OS.

Table 2. Experimental Parameters

Parameter	Domain	Default
Number of Tuples N	$10K, 100K, 1M, 10M$	$1M$
Query Range/Selectivity	100, 500, 1000, 2000	500
Index Cache Hit Rate for Client	0.0, 0.4, 0.8, 1.0	0.8
Query Skew	1, 1.5, 2, 2.5, 3	1
Threshold Number of Servers m, Total Number of Servers n	(10, 15), (11, 17), (12, 19), (13, 21)	(13, 21)

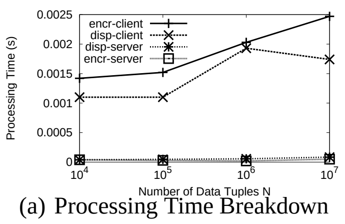
8.2 Experimental Results

General Overhead Comparison. To understand the security overhead brought by our approach *sida*, we first evaluate the efficiency of *sida* for processing different types of queries. We varied the number of tuples N from 10K to 10M as shown on the x-axis while fixing other parameters as default. These figures show that having strong data security schemes in *sida* do not dramatically degrade query performance as compared to *baseline* with no security schemes at all. Take 10M tuples as an example, from Figs. 4(a) and 5(a), we can see that the total processing time of *sida* (shown as the middle bar) for an exact query is 0.86ms vs. 0.28 ms of that of *baseline* (shown as the left bar), and the total processing time of *sida* for a range query is 167ms vs. 20ms of that of *baseline*. The communication size of *sida* for an exact query is around 0.5KB vs. 0.023KB of that of *baseline*, as shown in Fig. 4(b), and the communication size of *sida* for a range query is 78KB vs. 9.8KB of that of *baseline*, as shown in Fig. 5(b). In many cases, *sida* even outperforms *encr* with weaker security schemes. We note that *encr* has larger network latency in general. Although *sida* sometimes transmits more data than *encr* because *sida* packs tuples into tuple matrix columns and uses checksums, as shown in Fig. 5(b), the data communication of *sida* happens between the client and multiple servers in parallel, so *sida* incurs smaller network latency. The comparison result of total processing time on data inserts is similar, so we do not show it here and specifically study its client processing time below. Considering the overhead of security, these results suggest that our approach is efficient to be used in practice.

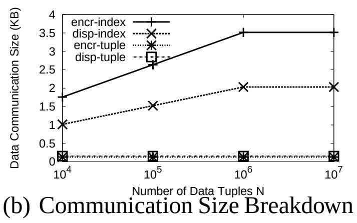
Overhead Breakdown. By breaking down the processing time in Fig. 4(a) and Fig. 5(a), we find that client processing dominates query processing in *sida* and *encr*, which is because the client directs query processing and encoding/decoding all happen-s on the client. For exact queries where index traversal is the dominant factor, index communication dominates tuple communication, as shown in Fig. 4(b). However for range queries where the amount of tuple processing is more than index traversal, tuple communication dominates index communication, as shown in Fig. 5(b). For the above read-only queries, the client processing time of *sida* is slightly better than that of *encr*. However for data inserts and updates in Fig. 6(a) and 7(a), the client processing time of *sida* is slightly worse than *encr*, which suggests that salted IDA decoding is faster, and only its encoding is a bit slower.

Varying Number of Tuples. We then study the effects of increasing number of tuples N on query performance. Fig. 4 shows that the processing time and communication sizes for exact queries increase steadily with bigger values of N. For data inserts and updates shown in Figs. 6 and 7, the client processing time and the data communication sizes increase slowly. However for range queries shown in Fig. 5, these overheads almost do not change. This is because the range query size is fixed, and the major part of processing for a range query is to process the tuples in the requested range, which number could be much larger than the number of traversed index nodes (the height of the B+-tree index). In general, our approach scales well with increasing number of tuples.

Varying Query Range/Selectivity. We then study the effects of range query size/query selectivity on query performance. We varied the range query size from 100 to 2000 while fixing other parameters as default. As a result, the answer size for the range query would increase. Fig. 8 shows that the client processing time and data communication

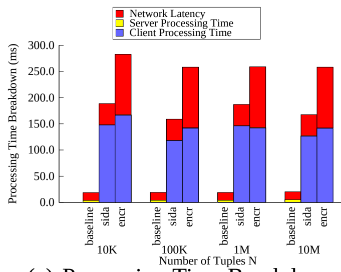


(a) Processing Time Breakdown

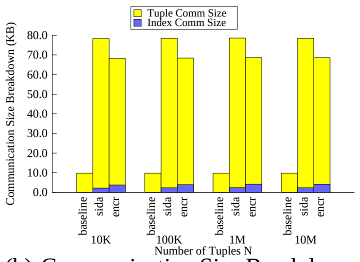


(b) Communication Size Breakdown

Fig. 4. Effects of Varying Number of Tuples N on Exact Queries.

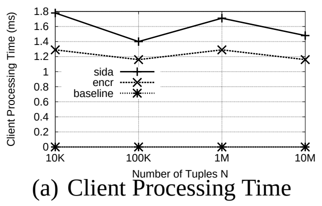


(a) Processing Time Breakdown

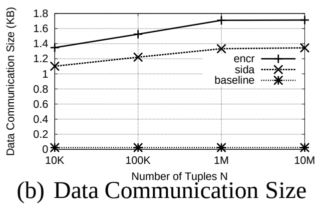


(b) Communication Size Breakdown

Fig. 5. Effects of Varying Number of Tuples N on Range Queries.

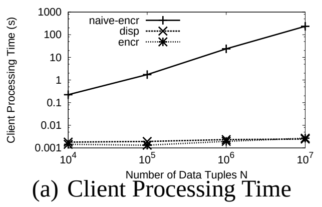


(a) Client Processing Time

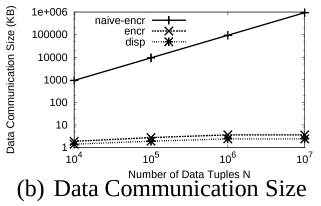


(b) Data Communication Size

Fig. 6. Effects of Varying Number of Tuples N on Inserts



(a) Client Processing Time

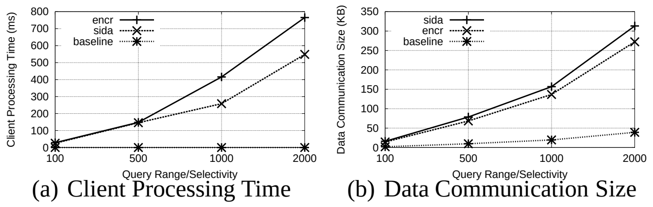


(b) Data Communication Size

Fig. 7. Effects of Varying Number of Tuples N on Updates

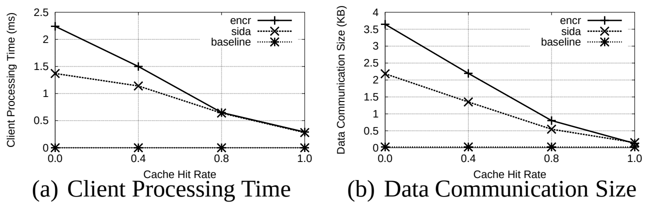
size in *sida* and *encr* increase more dramatically than those of *baseline*. This is because *sida* and *encr* must decode the encoded candidate answers sent from servers, so they are more sensitive to the change to the query answer size.

Varying Cache Hit Rate. We next study the effects of index cache on the client on saving the client from decoding index notes, and thus boosting query performance. We changed the desired cache hit rate from 0.0 (no caching) to 1.0 (caching all the index



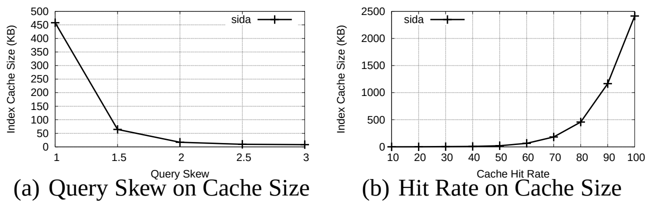
(a) Client Processing Time (b) Data Communication Size

Fig. 8. Varying Selectivity on Range Queries



(a) Client Processing Time (b) Data Communication Size

Fig. 9. Varying Cache Hit Rate on Exact Queries



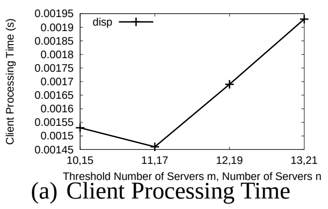
(a) Query Skew on Cache Size (b) Hit Rate on Cache Size

Fig. 10. Effects on Cache Size

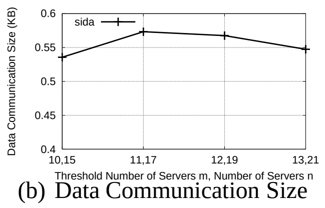
nodes). Fig. 9 indicates that the effect of caching is significant for processing exact queries. For range queries however, we do not see such a significant effect, which is because the major processing for a range query is on tuples instead of on the index. We do not report results for range queries due to space limit.

Effects of Query Skew on Cache Size. Continuing the previous study on cache hit rate, we now look at how large size needed by an index cache for a table with $N = 10^6$ tuples. Fig. 10(b) shows that the size needed by an index cache increases exponentially with the desired cache hit rate. This size is not big though, only around 450KB for 80% cache hit rate, which suggests it is practical to cache partial and even major part of index on the client. Caching reduces the overhead brought by security, making both security and efficiency achieved at the same time. When query access is more skewed, i.e. from 1 to 3 as shown in Fig. 10(a), the size of cache needed is smaller.

Results of Varying Number of Servers for Our Approach. We varied the number of the threshold number of servers for retrieving data, m, from 10 to 13, and the corresponding number of servers for distributing data, n, from 15 to 21, while fixing other parameters as default. As shown in Fig. 11(a), the increasing number of servers and fault tolerance support $(n - m)$ increases the number of data pieces that the client needs to calculate, disperse and recover from, thus increasing the client processing time. Data



(a) Client Processing Time



(b) Data Communication Size

Fig. 11. Effects of Varying Number Of Servers m, n on Exact Queries

storage size also increases (from 15/10 to 21/13 times of the size of data before dispersing), but the data communication size is not much affected, as shown in Fig. 11(b).

9 Conclusion

To solve the security concern for enterprise use of relational data management in the cloud, this paper has proposed a comprehensive framework for secure and efficient query processing on relational data in the cloud. Our work is distinguished from previous works in that data confidentiality is ensured in both storage and at access time, and different queries and data updates are supported. Data confidentiality in storage is ensured using the “salted” IDA scheme to encode and disperse the data. Data confidentiality in query accesses is ensured by only allowing a single operation called *column-access-via-proxy* between clients and servers. To support efficient query processing, a B+-tree index is built on frequently queried key attributes. Both the index and the data table are organized into matrices, encoded and dispersed using salted IDA. Moreover, data availability is provided by leveraging IDA, and data integrity is provided by leveraging checksum and index structure. A security analysis and an experimental evaluation indicate our framework achieves a practical trade-off between security and efficiency.

References

[1] Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: CCS. (2009) 199–212
[2] Hacigumus, H., Iyer, B.R., Li, C., Mehrotra, S.: Executing SQL over encrypted data in the database service provider model. In: SIGMOD. (2002)
[3] Hore, B., Mehrotra, S., Tsudik, G.: A privacy-preserving index for range queries. In: VLDB. (2004) 720–731
[4] Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. In: SIGMOD. (2004) 563–574
[5] Ge, T., Zdonik, S.B.: Fast, secure encryption for indexing in a column-oriented dbms. In: ICDE. (2007) 676–685
[6] Damiani, E., di Vimercati, S.D.C., Jajodia, S., Paraboschi, S., Samarati, P.: Balancing confidentiality and efficiency in untrusted relational dbmss. In: CCS. (2003) 93–102
[7] Shmueli, E., Waisenberg, R., Elovici, Y., Gudes, E.: Designing secure indexes for encrypted databases. In: DBSec. (2005)

[8] Gentry, C.: Fully homomorphic encryption using ideal lattices. In: STOC. (2009) 169–178

[9] Chor, B., Kushilevitz, E., Goldreich, O., Sudan, M.: Private information retrieval. J. ACM **45**(6) (1998) 965–981

[10] Rabin, M.O.: Efficient dispersal of information for security, load balancing, and fault tolerance. J. ACM **36**(2) (1989) 335–348

[11] Plank, J.S., 0002, Y.D.: Note: Correction to the 1997 tutorial on reed-solomon coding. Softw., Pract. Exper. **35**(2) (2005) 189–194

[12] Bowers, K.D., Juels, A., Oprea, A.: Hail: a high-availability and integrity layer for cloud storage. In: CCS. (2009) 187–198

[13] Wang, C., Wang, Q., K.Ren, Lou, W.: Ensuring data storage security in cloud computing. In: Proceedings of the 17th IEEE International Workshop in Quality of Service. (2009) 1–9

[14] : Cleversafe responds to cloud security challenges with cleversafe 2.0 software release. http://www.cleversafe.com/news-reviews/press-releases/press-release-14 (2010)

[15] : Information dispersal algorithms: Data-parsing for network security. http://searchnetworking.techtarget.com/Information-dispersal-algorithms-Data-parsing-for-network-security (2010)

[16] Comer, D.: Ubiquitous b-tree. ACM Comput. Surv. **11**(2) (1979) 121–137

[17] Emekci, F., Agrawal, D., Abbadi, A.E., Gulbeden, A.: Privacy preserving query processing using third parties. In: ICDE. (2006)

[18] Ge, T., Zdonik, S.B.: Answering aggregation queries in a secure system model. In: VLDB. (2007) 519–530

[19] Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In: CCS. (2008) 139–148

[20] Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: CRYPTO. (1996) 1–15

[21] Kantarcioglu, M., Clifton, C.: Security issues in querying encrypted data. In: DBSec. (2005)

[22] Wang, H., Lakshmanan, L.V.S.: Efficient secure query evaluation over encrypted xml databases. In: VLDB. (2006) 127–138

[23] Samarati, P., Sweeney, L.: Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. Technical report (1998)

[24] Agrawal, D., Abbadi, A.E.: Quorum consensus algorithms for secure and reliable data. In: Proceedings of the Sixth IEEE Symposium on Reliable Distributed Systems. (1988) 44–53

[25] : Crypto++ library 5.6.0. http://www.cryptopp.com/

[26] : 2009 report on internet speeds in all 50 states. http://www.speedmatters.org/content/2009report

[27] : Tpc-w. http://www.tpc.org/tpcw/