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.

Keywords: Data security in the cloud, Query processing on encrypted data, Data confidentiality, Data availability.

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. However, the security of sensitive data stored in the cloud remains a big concern, and even a road block to the widespread usage of the cloud for relational data management and query processing. The shared environment of the cloud renders access control policies and authentication vulnerable [1]. Many enterprises therefore question whether adequate security and functionality can be ensured for performing their regular data storing and query processing tasks in the cloud.

Data confidentiality is one of the most important security concerns and challenges. It should be adequately provided to safeguard against attackers' analysis and inferences. In addition, data confidentiality has to be balanced with query processing functions and performance. First, although encryption is a commonly used solution to data confidentiality, encryption itself is insufficient to guarantee data confidentiality, even if the encryption scheme does not reveal any characteristics about the plaintext data and is resistant to statistical analysis. When encrypted data is frequently accessed to serve clients' queries, any potential information leakage should also be controlled, since attackers may infer the plaintext data from clients' accessed positions on the encrypted data. Many existing proposals of query processing on encrypted data do not consider both confidentiality for data residing in storage and for data being accessed by queries [2,3,4,5,6,7]. Second, different queries must be supported in the same framework, and practical query performance should not be lost in pursuit of the above data confidentiality requirements. Note that some of the previous works are only able to support one or two types of queries on encrypted data, and in general do not support data updates [3,4]. The powerful cryptographic techniques such as homomorphic encryption [8] and Private Information Retrieval (PIR) [9] can satisfy the above mentioned data confidentiality requirements, but they are computationally expensive and can adversely impact both latency and throughput. The approaches improving the performance of PIR via the use of special hardwares [10,11] may not be feasible for some small businesses who do not have the resources to make such investments.

Next to data confidentiality are the concerns of data availability and integrity. *Information Dispersal Algorithm* (IDA) [12] and similar error-correcting codes [13] have been used in recent works [12,14,15] to provide data availability, and are commercialized [16]. A recent trend in industry even considers IDA as an alternative to traditional data encryption [17], since IDA provides both data availability and a 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 types of queries on relational data in the cloud. We aim at a practical solution with balanced security and functions. We achieve confidentiality for data residing in storage using a variant of IDA, called *"salted" IDA* (Section 4). Salted IDA relies on pseudo-randomness to improve the data confidentiality of the original IDA scheme against computationally 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 and routing them via trusted proxies, which we call *column-access-via-proxy* (Section 5), so that different queries and queries among different clients are unlikely to be differentiated. We discuss the security implications of these two schemes in a comprehensive security analysis (Section 7).

To enable practical query processing, we build a secure B+-tree index [18] 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 the *column-access-via-proxy* operations. During query processing, a client retrieves and decodes only a small part of the index, based on which the client locates the candidate answer tuples. We boost query performance by caching parts of the index on the client. Caching the index also helps improve data confidentiality at accesses by confusing inferences on the index traversal paths. We are thus able to support common relational database queries such as exact queries, range queries and data updates with consistent security guarantees (Section 6) and practical performance (Section 8).

2 Related Work

To support queries on encrypted relational data, one class of solutions proposed processing encrypted data directly. However, these approaches do not provide good tradeoff between 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 [4,19] 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 [20,8], but relies on expensive public key cryptosystem and thus is not practical.

Instead of processing encrypted data directly, an alternative is to use an encrypted index which allows the client to traverse the index and to locate the data of interest in a small number of rounds of retrieval and decryption [6,7,5]. Although these works provide confidentiality for data residing in storage, they do not provide data confidentiality under dynamic query access patterns. Recent work obfuscates users' data access patterns using special oblivious RAM for data outsourced in the cloud [11], but it still incurs a lot of computation and communication costs and requires special-purpose hardware. A contemporary work to our work obfuscates users' data access pattern by shuffling index nodes [21]. In contrast to the above approaches, our work provides a comprehensive and practical secure query processing framework to protect data in storage and at accesses as well as to support different kinds of queries.

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. 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 the cloud is heavily loaded with many clients issuing many queries continuously. This is typical of modern cloud systems. We support exact, range queries and tuple updates given index keys as predicates, as well as tuple insertion and deletion.

3.2 Attacker Model

Attacker and Prior Knowledge Assumptions. We consider attackers are external entities or the servers where data is stored. We do not deal with insider attacks, such as from malicious partners. We assume client machines are safe, thus any confidential information on the client such as the secret key C is not known to attackers. Attackers do not know clients' queries. However, attackers could know the clients' data distribution and even some exact 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 target to compromise data confidentiality without compromising data availability or integrity; (2) attacks that target to compromise data integrity or availability, e.g. modifying the encoded tuples or index keys, or Denial-of-Service (DoS) attacks. We say servers are *faulty* in (2). In (1), attackers can compromise any number of servers. They can analyze the encoded data, 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, and even try to infer the secret key C.

4 Data Encryption and Dispersal by "Salted" IDA

Information Dispersal Algorithm (IDA) [12] ensures secure and reliable storage. It is widely used in emerging cloud storages [16,14,22]. We use IDA as the basis for providing data confidentiality and availability, and propose an easy-to-use data encoding and dispersal scheme called *salted IDA*.

4.1 Information Dispersal Algorithm (IDA)

We first introduce IDA [12]. 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 (faulty pieces) for data retrievals. Table 1 summarizes the notations we use in the paper.

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	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

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.* A Vandermonde matrix satisfies this property, where each row is in the form of $C_{i,:} = (1, a_i, ..., a_i^{m-1})$ $(a_i \in GF(2^s), 1 \leq i \leq n)$. For example in $GF(2^4)$,

$$C = \begin{pmatrix} 3^0 & 3^1 & 3^2 \\ 4^0 & 4^1 & 4^2 \\ 5^0 & 5^1 & 5^2 \\ 6^0 & 6^1 & 6^2 \\ 7^0 & 7^1 & 7^2 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 5 \\ 1 & 4 & 3 \\ 1 & 5 & 2 \\ 1 & 6 & 7 \\ 1 & 7 & 6 \end{pmatrix}$$

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

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

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

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. An alternative to generate salt is to employ a hash function on *ss* and the column index i, $hash(i, ss)$. The security of salted IDA is established in Section 7.1.

5 Secure Cloud Data Access

5.1 Overview

We use salted IDA to encode and disperse the data onto servers in the cloud. To be able to perform queries on salted IDA encoded matrix, we retrieve partial data by retrieving single columns of the matrix as follows.

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

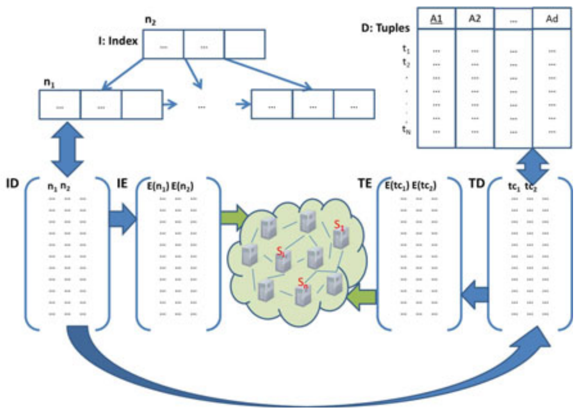


Fig. 1. Secure Cloud Data Access Framework

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

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

Using the above *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 directly on the IDA encoded matrix based on plaintext input is infeasible. 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. Each column of TD thus 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, $S_1, S_2, ..., S_n$, using 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. Every node of I then 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 this paper and discuss the case of multiple columns representing one tree node in our technical report [23].

We assign each tree node an integer column address denoting its beginning column in ID according to the order it is inserted in ID. Similarly, we assign every tuple column of TD an integer column address according to the order its tuples are added into TD. 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 $(isLeaf, col_0, col_1, key_1, col_2, key_2, ..., col_{b-1}, key_{b-1}, col_b)$, where $isLeaf$ indicates if the node 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.

	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

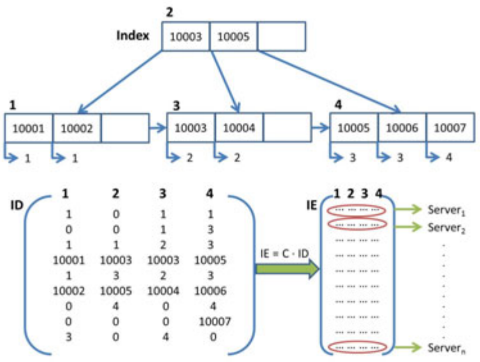


Fig. 3. Index Matrix of Employee Table

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, $key_1 = 10003$, $col_2 = 3$ is the column address of its middle child, $key_2 = 10005$, $col_3 = 4$ is the column address of its rightmost child, $key_3 = 0$ 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 with key $= 10001$, $col_2 = 1$ is the column address of the tuple with key $= 10002$, $col_3 = 0$ 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 set of servers as the index keys, the column size of the tuple matrix TD is also set to m. Initially, to organize the existing

d-dimensional tuples of D in TD, we 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) [24] on the attribute values of all p tuples, so as to verify the integrity of these tuples returned by servers.

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. 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 salt).

5.4 Secure Column Access via Proxies

In our framework, a 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 decisions. Both data updates and 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 salt. The m requests are sent in parallel.

By monitoring these column accesses, attackers cannot precisely determine the content of a query or the plaintext data involved. However, attackers could learn the initiator client's identity 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 and fault tolerance. 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, 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. Note that the answer tuples cannot be retrieved directly from the tuple matrix columns in between the tuple matrix columns corresponding to x_l and x_r, since tuples can be dynamically inserted and deleted, and the tuple matrix columns may not be ordered by index keys. After finding the qualified index keys and the associated tuple matrix column addresses, the qualified tuple matrix columns can be retrieved in batch.

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 that 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 that the root node of the index is always cached. Caching the index could also confuse attackers' inferences that infer the index structure and the data based on the order of requests, thus help improving data confidentiality during data accesses.

7 Security Analysis

In loaded cloud environment with the use of proxies between clients and servers and client side index caches, we ensure data confidentiality against polynomial size circuits bounded attackers, even when all servers are monitored by attackers. We ensure 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* [10,25] 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. The original IDA scheme [12] does not have such security guarantees, 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 the 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 with a salt value 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 with $b = 50, N = 10^6$, $2^{32N/(b-1)} > 2^{653061}$. Given such a large choice space of salts and that after adding salt to D and D', the data will be encoded and mixed together by applying an unknown secret matrix C, the ciphertext matrices E, E' obtained by salted IDA encoding are computationally indistinguishable. $\square$

Since the rows of E, E' are distributed onto n servers, two rows $E_{i,:}, E'_{i,:}$ are also computationally indistinguishable. Similarly due to the large choice space of salt, 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 may still be located due to its unique size. 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 define confidentiality as follows.

Definition 1. *A* secure relational data processing framework ensures data confidentiality *if it satisfies the following conditions: (1) Two encrypted data sets are computationally indistinguishable; (2) By observing index accesses on the encrypted data, finding out the correct correspondences between the plaintext data and the encrypted data only has negligible advantage over random guesses with prior knowledge on polynomial size circuits.*

Theorem 1 shows that our framework satisfies condition (1). We next show that it satisfies condition (2). Because query processing in our framework is performed through *column-access-via-proxy* operations, and the cloud is typically loaded with many queries from many clients, 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. *In loaded environment with index cache enabled on the client, the best that attackers can gain under the column-access-via-proxy scheme is to identify the columns that represent leaf nodes of the index.*

Proof. In loaded environment with index cache enabled on the client, all the attackers observe on the index are single and batch column accesses. The exact structure of the index are not known to the attackers. To them, single column accesses could correspond to either internal nodes or leaf nodes in processing exact or range queries, while batched column accesses could only correspond to leaf nodes in processing 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 are unlikely to get the total order of all the leaf nodes. □

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

Lemma 2. *Finding the 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. 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 framework satisfies Definition 1, we claim the following.

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

Note that the loaded environment that we assume in the above is typical in the cloud. We do not deal with under-loaded scenarios for now, but we suggest requesting redundant columns in a k-anonymous [26] fashion in each request to provide practical data confidentiality at accesses in under-loaded scenarios.

7.2 On Data Integrity and Availability

We check integrity violations on the index structure using the relationships of sorted key values and the relationships of nodes in the index, and check integrity violations on data values using the checksums. We rely on IDA to provide data availability when no more than $n - m$ servers are faulty. More details on data integrity and availability can be found in our technical report [23].

8 Experimental Evaluation

Our evaluation focuses on the following: (1) the efficiency of our framework for processing different types of queries; (2) the overhead introduced by security when compared with the *baseline* query processing with no security provided, and with the basic encrypted index approach [6] of insufficient data confidentiality and 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 the query 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 [27] for implementing IDA and MAC (Message Authentication Code) 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 dividing the communication sizes with the average internet download speed (5.1Mbps) and upload speed (1.1Mbps) in a wide area network [28]. 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 the most frequently accessed index nodes based on the query workload.

Data Set. We extracted 5 attributes, *I_ID*, *I_A_ID*, *I_RELATED1*, *I_STOCK* and *I_PAGE* from the *Item* table of TPC-W Benchmark [29] 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. Our experimental parameters are summarized 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 Fedora Core 8 OS.

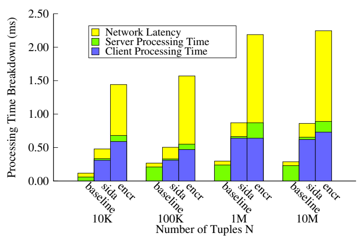
Table 2. Experimental Parameters

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

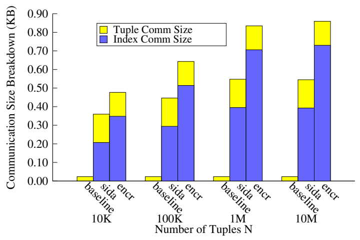
8.2 Experimental Results

General Overhead Comparison. To understand the security overhead due to *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 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* which has weaker security guarantees. 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. These results suggest that our approach is a practical security solution.

Overhead Breakdown. By breaking down the processing time in Figs. 4(a) and 5(a), we find that client processing dominates query processing in *sida* and *encr*, which is because the client directs query processing and perform all the decoding. For exact queries in which index traversal is the dominant factor, index communication dominates tuple communication, as shown in Fig. 4(b). However for range queries in which tuple processing takes more work than index traversal, tuple communication dominates index communication, as shown in Fig. 5(b).

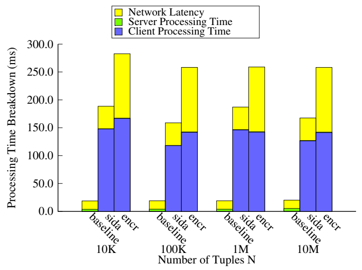


(a) Processing Time Breakdown

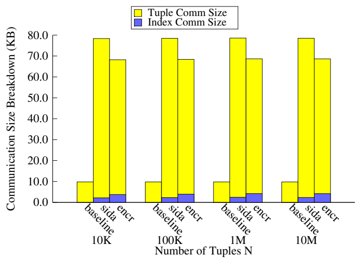


(b) Communication Size Breakdown

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



(a) Processing Time Breakdown

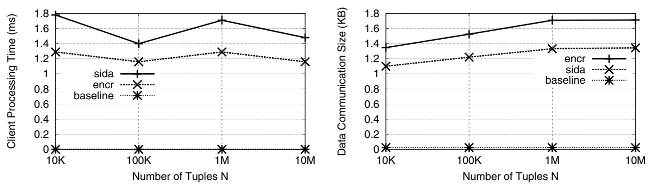


(b) Communication Size Breakdown

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

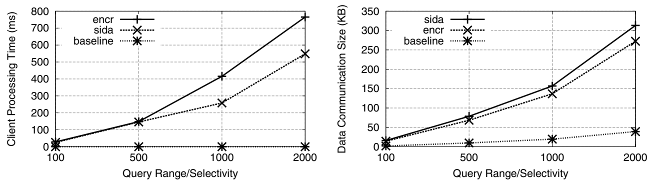
Varying Number of Tuples. We then study the effects of increasing the number of tuples N on query performance. Fig. 4 shows that the processing time and communication sizes for exact queries increase steadily with larger values of N. For data inserts shown in Fig. 6, the client processing time and the data communication sizes increase slowly. The results for data updates are similar to those of data inserts, and thus are omitted due to space limit. 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, the sum of which could be much larger than the number of traversed index nodes. In general, our approach scales well with the 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. 7 shows that the client processing time and data communication size in *sida* and *encr* increase more dramatically than those of *baseline*. This is because *sida* and *encr* must decode the encoded



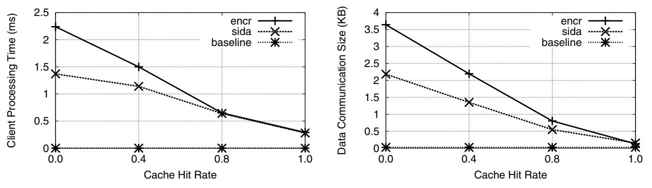
(a) Client Processing Time (b) Data Communication Size

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



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

Fig. 7. Varying Selectivity on Range Queries



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

Fig. 8. Varying Cache Hit Rate on Exact Queries

candidate answers sent from servers, so they are more sensitive to the change of the query answer size.

Varying Cache Hit Rate. We next study the effects of caching index on the client for reducing the costs for retrieving and decoding index nodes. We changed the desired cache hit rate from 0.0 (no caching) to 1.0 (caching all the index nodes). Fig. 8 indicates that caching improves the performance for processing exact queries.

We have also studied the effects of query skew in reducing the index cache size and the effects of the number of servers on query processing time. We found that the size of index cache needed for 80% cache hit rate is small enough to reside in the client memory, and our framework scales well with the increasing number of servers. These results can be found in our technical report [23].

9 Conclusion

To solve the security concern for widespread use of relational data management in the cloud, this paper has proposed a comprehensive framework for practical secure 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 proxied single operations 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 using IDA, and data integrity is provided using checksum and the index structure. A security analysis and an experimental evaluation indicate our framework achieves a practical trade-off between security and performance.

Acknowledgement. This work is supported by NSF Grant IIS-1053594.

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: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS 2009, pp. 199–212. ACM, New York (2009)
2. Hacigümüş, H., Iyer, B., Li, C., Mehrotra, S.: Executing sql over encrypted data in the database-service-provider model. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD 2002, pp. 216–227. ACM, New York (2002)
3. Hore, B., Mehrotra, S., Tsudik, G.: A privacy-preserving index for range queries. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004. VLDB Endowment, vol. 30, pp. 720–731 (2004)
4. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data. SIGMOD 2004, pp. 563–574. ACM, New York (2004)
5. Ge, T., Zdonik, S.B.: Fast, secure encryption for indexing in a column-oriented dbms. In: ICDE, pp. 676–685 (2007)
6. Damiani, E., Vimercati, S.D.C., Jajodia, S., Paraboschi, S., Samarati, P.: Balancing confidentiality and efficiency in untrusted relational dbmss. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003, pp. 93–102. ACM, New York (2003)
7. Shmueli, E., Waisenberg, R., Elovici, Y., Gudes, E.: Designing secure indexes for encrypted databases. In: IFIP Working Conference on Database Security, pp. 54–68 (2005)
8. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, pp. 169–178. ACM, New York (2009)

9. Chor, B., Kushilevitz, E., Goldreich, O., Sudan, M.: Private information retrieval. Journal of The ACM 45(6), 965–981 (1998)
10. Kantarcioglu, M., Clifton, C.: Security issues in querying encrypted data. In: IFIP Working Conference on Database Security, pp. 325–337 (2005)
11. Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In: CCS, pp. 139–148 (2008)
12. Rabin, M.O.: Efficient dispersal of information for security, load balancing, and fault tolerance. Journal of The ACM 36(2), 335–348 (1989)
13. Plank, J.S., Ding, Y.: Note: Correction to the 1997 tutorial on reed-solomon coding. Softw., Pract. Exper. 35(2), 189–194 (2005)
14. Bowers, K.D., Juels, A., Oprea, A.: Hail: a high-availability and integrity layer for cloud storage. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS 2009, pp. 187–198. ACM, New York (2009)
15. Wang, C., Wang, Q., Ren, K., Lou, W.: Ensuring data storage security in cloud computing. In: Proceedings of the 17th IEEE International Workshop in Quality of Service, pp. 1–9 (2009)
16. Cleversafe: Cleversafe responds to cloud security challenges with cleversafe 2.0 software release (2010), `http://www.cleversafe.com/news-reviews/press-releases/press-release-14`
17. www: Information dispersal algorithms: Data-parsing for network security (2010), `http://searchnetworking.techtarget.com/Information-dispersal-algorithms-Data-parsing-for-network-security`
18. Comer, D.: Ubiquitous b-tree. ACM Comput. Surv. 11(2), 121–137 (1979)
19. Emekci, F., Agrawal, D., Abbadi, A.E., Gulbeden, A.: Privacy preserving query processing using third parties. In: Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, p. 27. IEEE Computer Society, Washington, DC, USA (2006)
20. Ge, T., Zdonik, S.: Answering aggregation queries in a secure system model. In: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB 2007. VLDB Endowment, pp. 519–530 (2007)
21. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Efficient and private access to outsourced data. In: ICDCS (to appear 2011)
22. Abu-Libdeh, H., Princehouse, L., Weatherspoon, H.: Racs: a case for cloud storage diversity. In: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, pp. 229–240. ACM, New York (2010)
23. Wang, S., Agrawal, D., Abbadi, A.E.: A comprehensive framework for secure query processing on relational data in the cloud. Technical report, Department of Computer Science, UCSB (2010)
24. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 1–15. Springer, Heidelberg (1996)
25. Wang, H., Lakshmanan, L.V.S.: Efficient secure query evaluation over encrypted xml databases. In: Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB 2006. VLDB Endowment, pp. 127–138 (2006)
26. Samarati, P., Sweeney, L.: Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. Technical report (1998)
27. Dai, W.: Crypto++ library 5.6.0, `http://www.cryptopp.com`
28. www: report on internet speeds in all 50 states (2009), `http://www.speedmatters.org/content/2009report`
29. www: Tpc-w, `http://www.tpc.org/tpcw`