SEDB: Building Secure Database Services for Sensitive Data

Quanwei Cai[1,2,3], Jingqiang Lin[1,2(✉)], Fengjun Li[4], and Qiongxiao Wang[1,2]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
{qwcai,linjq,qxwang}@is.ac.cn

[2] Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing 100093, China

[3] University of Chinese Academy of Sciences, Beijing 100049, China

[4] The University of Kansas, Lawrence, KS 66045, USA
fli@ku.edu

Abstract. Database outsourcing reduces the cost of data management; however, the confidentiality of the outsourced data is a main challenge. Existing solutions [9,13,16,17] either adopt multiple encryption schemes for data confidentiality that only support limited operations, or focus on providing efficient retrieval with problematic update support. In this paper, we propose a secure database outsourcing scheme (SEDB) based on Shamir's threshold secret sharing for practical confidentiality against honest-but-curious database servers. SEDB supports a set of commonly used operations, such as addition, subtraction, and comparison, and is among the first to support multiplication, division, and modulus. We implement a prototype of SEDB, and the experiment results demonstrate a reasonable processing overhead.

Keywords: Database · Outsourcing · Confidentiality · Secret sharing

1 Introduction

In the cloud computing environment, database outsourcing can lower costs [4], thus enables organizations to focus on their core businesses. However, outsouring sensitive data to the third parties increases the risk of unauthorized disclosure, as curious administrators can snoop on sensitive data, and attackers can access all the outsourced data once it compromises the data servers.

There are two approaches to provide confidentiality in database outsouring. One is based on client-side encryption, where the clients (or proxies) encrypt the data before uploading it to database servers so that the servers perform the requested operations over the encrypted data. Fully homomorphic encryption [6] allows the servers to execute arbitrary functions over one encryption of the data. However, fully homomorphic encryption is still prohibitive impractical [7], which requires slowdowns on the order of $10^9\times$. CryptDB [13] and MONOMI [17] implement multiple cryptosystems, each of which supports a class of SQL queries,

L.C.K. Hui et al. (Eds.): ICICS 2014, LNCS 8958, pp. 16–30, 2015.
DOI: 10.1007/978-3-319-21966-0_2

such as AES with a variant of the CMC mode [10] for equality comparison, order-preserving encryption [3] for range query, Paillier cryptosystem [12] for summation, and the efficient retrieval protocol [15] for word search. As a result, they have to maintain multiple copies of a same sensitive data. Moreover, these schemes do not support the update operation well. For instance, when the data is updated by summation, only the copy encrypted with Paillier cryptosystem will be updated while the other copies remain stale, which harms the execution of other queries. Last but not least, these schemes cannot support operations such as multiplication, division, and modulus.

The other solutions are based on threshold secret sharing [14] in which the clients split the sensitive data into shares and store them in independent servers. Solutions of this category require more servers than the encryption-based ones do. However, with the advances in virtualization, the hardware cost has been decreased remarkably. It is believed that the implementation cost should not be the main obstacle to the adoption of these solutions. Several schemes are proposed to achieve efficient retrieval. For example, AS5 [9] preserves the order of the data in the shares by choosing appropriate coefficients of the polynomial for secret sharing, and a B^+ index tree is built to improve the query processing in [16]. With a focus on efficient retrieval, these solutions not only require a priori knowledge about the data, but also support the update operations poorly.

In this paper, we proposed a secure database outsourcing scheme (SEDB) based on Shamir's threshold secret sharing. SEDB employs three independent database servers to store shares of each sensitive data item, and coordinates the three servers to complete the clients' requested operations cooperatively. In summary, SEDB achieves the following properties:

- It supports a wider set of operations including multiplication, division and modulus in addition to addition, subtraction and comparison. To the best of our knowledge, it is the first practical solution that supports these operations.
- SEDB is easy to deploy as it is an out-of-the-box solution. SEDB doesn't needs any modification on the database management system (DBMS) or the applications of database services. Moreover, SEDB doesn't need any priori knowledge of the data for the setup of the database services.
- It provides a continuous database outsourcing service. Existing encryption-based solutions requires costly and problematic coordination to keep multiple copies of data consistent, and the secret sharing based solutions often need to maintain additional information (e.g., the index tree [16], the mapping [1,9]), which may interrupt the database services during data update. Unlike them, SEDB only maintains one share of the data at each database server, and thus ensures a continuous database outsourcing service. We have implemented SEDB on MySQL, which is the first prototype providing continuous database outsourcing services based on secret sharing, to the best of our knowledge.

2 System Overview

As shown in Fig. 1, SEDB consists of three backend database servers, a SEDB coordinator, and a set of SEDB client plug-ins (denoted as SEDB plug-ins), one

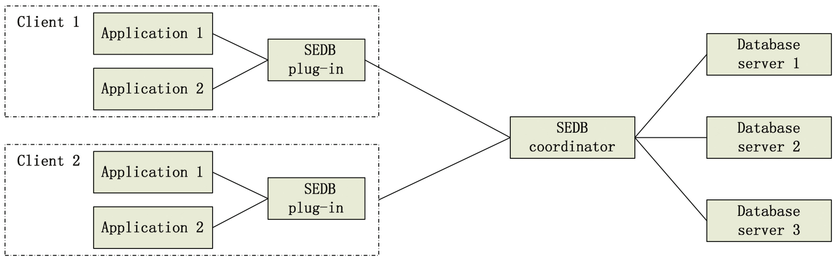

Fig. 1. The architecture of SEDB.

at each client. A client has several applications of the database service. When an application issues an SQL query, the SEDB plug-in rewrites the query according to its operation type and sends it to the SEDB coorinator which generates three SQL queries from it and distributes each to the backend database server. There is an unmodified DBMS and several user-defined functions (UDFs) in each backend database server, which executes the requested operations over the shares of sensitive data. In SEDB, the applications need no modification to execute the functions over sensitive data, they issues the SQL queries through the standard API and library; the SEDB plug-in is responsible for sharing sensitive data and recovering it for the applications; the SEDB coordinator ensures the requets processed at each database server in the same order and makes the backend database servers complete the requested operation through one or two phases of communications with the database servers.

Trust Model. In SEDB, we assume the clients that are authorized to process the sensitive data are trusted. The SEDB plug-in deployed at client side is also trusted and assumed to follow the protocol strictly without leaking any sensitive data. On the contrary, the SEDB coordinator and the backend database servers deployed at different third parties are assumed to be honest-but-curious: on one hand, the honest coordinator/server executes the requested operations without tampering with query content, the execution and results, or the shares in the DBMS; on the other hand, a curious coordinator/server may infer the sensitive data from the submitted queries, the execution results, or the priori knowledge about the outsourced data.

Network Assumption. We assume the messages transmitted between the client (the SEDB plug-in) and the SEDB coordinator, the SEDB coordinator and the database servers can be captured by attackers. Therefore, we employ AES to ensure message confidentiality. Each server shares a secret key with each of the other servers and all clients, e.g., $k_{c,si}$ between the client c and server i. The message m encrypted with the key k is denoted as $[m]_k$. Moreover, we assume the clients have limited bandwidth, while the bandwidth of the SEDB coordinator and the servers are reasonably large enough in the cloud computing environment.

Finally, although we present SEDB with a focus on the process over the integers, it can be extended to support the data of other types (e.g., char, varchar and float), by transforming them into one or more integers. As we adopt Shamir's threshold secret sharing scheme [14] to share sensitive data, we assume that there exists a large prime p such that all the computation results on the sensitive data are in the interval $[-(p-1)/2,\ (p+1)/2]$.

3 The Protocol

In SEDB, the applications, SEDB plug-ins, SEDB coordinators and database servers exchange messages through SQL queries, which ensures no modification of DBMSes and applications. For example, to insert a value v into the table *test* as the attribute *attr*, the application issues "insert into $test(attr)$ values(v)". The SEDB plug-in sends "insert into $test(attr)$ values$(encshares(v))$" to the SEDB coordinator, in which $encshares(v) = \{[share_1(v)]_{k_{c,s1}}, [share_2(v)]_{k_{c,s2}}, [share_3(v)]_{k_{c,s3}}\}$, where $[share_i(v)]_{k_{c,si}}$ is the encrypted share for server i. The SEDB coordinator splits the received SQL queries and sends "insert into $test(attr)$ values($\texttt{DecShare}(encshares(v)[i])$)" to server i, where `DecShare` is a UDF to execute decryption. The SQL queries for all the operations are detailed in the Appendix A.

We assume, in each table, there is a unique identifier of each row (i.e., the primary key). SEDB needs a shadow for each table to store the intermediate transformation of the original data. The shadow table is designed for the process of comparison, division and modulus.

3.1 Query Processing

SEDB supports a set of operations including addition, subtraction, comparison, multiplication, division, and modulus. We first describe the detailed process of each single operator and then discuss the process of SQL queries that contain multiple operators in Sect. 3.2.

3.1.1 Insert

The applications use insert operation to insert a confidential value v into the database. An application invokes an insert process by sending the SQL query with v as the parameter to the SEDB plug-in.

The SEDB plug-in parses the SQL query to get the value v, and uses Shamir's (2, 3)-threshold secret sharing scheme [14] to split v into 3 shares, where any 2 or more shares can be used to reconstruct v. To compute the shares of v, the SEDB plug-in produces the polynomial $f(x) = a_1x + a_0$, where $a_0 = v$, and a_1 is a non-zero integer chosen randomly from $[-(p-1)/2,\ (p+1)/2)$. Using the predefined vector $X = \{x_1,\ x_2,\ x_3\}$, where non-zero $x_i \epsilon [-(p-1)/2,\ (p+1)/2)$, the SEDB plug-in calculates $f(x_i)$ as the share for database server i. The large prime p and the vector X are known to all participants. Then, the SEDB plug-in rewrites the query by replacing v with the share vector $\{[f(x_1)]_{k_{c,s1}}, [f(x_2)]_{k_{c,s2}}, [f(x_3)]_{k_{c,s3}}\}$, and sends it to the SEDB coordinator. After that, the SEDB plug-in discards the polynomial. The SEDB coordinator splits the received SQL query into three queries, each with one encrypted share, for three backend database servers. On receiving the query, each database server decrypts the encrypted share, and stores it in the database.

3.1.2 Select

The applications use the select operation to retrieve the values that satisfy a specified condition. The SEDB plug-in forwards the `select` SQL query from the application to the SEDB coordinator directly, which further sends the received query to all three database servers. Then, each server encrypts its shares, and sends them to the SEDB coordinator as responses. On receiving the responses, the SEDB coordinator sorts them by the servers' identifiers, and sends them to the SEDB plug-in. Finally, the SEDB plug-in decrypts the encrypted shares, reconstructs the values and returns the retrieved values to the applications.

3.1.3 Addition and Subtraction

In SEDB, applications can achieve the addition and subtraction of two or more confidential values without recovering them. The process of the subtraction is the same as the addition, except that each server executes the subtraction instead of addition on the shares. So we only present the process for the addition here.

The application may want to perform additions on existing values in the database, or add a constant to an existing value. Without loss of generality, the former case can be expressed as updating v_3 with $v_1 + v_2$. To process it, the SEDB plug-in and SEDB coordinator simply forward the update query to three database servers, where each server updates its share of v_3 with the summation of its local shares of v_1 and v_2. Assume the polynomials for v_1 and v_2 are $f_1(x) = a_1x + v_1$ and $f_2(x) = b_1x + v_2$ respectively. Since the Shamir's secret sharing scheme is linear, v_3 is shared using the polynomial $f_3(x) = (a_1+b_1)x+(v_1+v_2)$. In the latter case, when the application wants to add a constant *const* to an existing value v_1 in the database, the SEDB plug-in needs to pre-process *const* by splitting it using the (2,3)-threshold secret sharing scheme. The encrypted shares are decrypted at each server and added to the corresponding share, respectively.

3.1.4 Multiplication

Compared to addition, the process of multiplication is more complicated since the multiplication of two shares increases the degree of the generated secret-sharing polynomial. In particular, when updating v_3 with $v_1 * v_2$, the degrees of the polynomials for v_1 and v_2 are 1, while the degree of the generated polynomial for v_3 increases to 2, which means 3 shares are needed to recover the result of the multiplication. To reduce the degree back to 1, we adopt the degree reduction scheme [5] in the process of the multiplication.

To process the multiplication, we introduce three UDFs at database servers: NewMul1, NewMul2, and MulConst. When the SEDB coordinator receives a multiplication query from the SEDB plug-in, it rewrites the query by replacing the operator $*$ with UDF NewMul1 and sends the rewritten query to all three backend database servers. To execute NewMul1, server i multiplies its shares of v_1 and v_2 to compute $mul1_i = v_1 * v_2$, and then splits $mul1_i$ using (2, 3)-threshold secret sharing scheme. The share vector $\{[share_1(mul1_i)]_{k_{s1,si}}, [share_2(mul1_i)]_{k_{s2,si}}, [share_3(mul1_i)]_{k_{s3,si}}\}$, with each subshare encrypted by pairwise secret key, is returned to the SEDB coordinator.

The SEDB coordinator combines the share vectors from three servers to generate parameters of UDF NewMul2. In particular, database server i takes parameter $([share_i(mul1_1)]_{k_{s1,si}}, [share_i(mul1_2)]_{k_{s2,si}}, [share_i(mul1_3)]_{k_{s3,si}})$ to compute $mul_i = \sum_{k=1}^{3} \lambda_k * share_i(mul1_k)$ where $\{\lambda_1, \lambda_2, \lambda_3\}$ is the first row of the following matrix in $[-(p-1)/2,\ (p+1)/2)$, and $\{x_1,\ x_2,\ x_3\}$ is the predefined vector for secret sharing. Then, mul_i is server i's share of the multiplication result using (2, 3)-threshold secret sharing scheme.

$$\begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{bmatrix}^{-1} \tag{1}$$

The multiplication with a constant can take two different approaches. The SEDB plug-in can simply invoke a UDF MulConst at three backend database servers to multiply each local share with this constant. In the case where the value of the constant needs protected, the SEDB plug-in needs to take a similar process as presented earlier in this section: it first splits the constant with a (2, 3)-threshold secret sharing scheme, sends the encrypted sub-shares to each backend database server, and executes the multiplication of two shares.

3.1.5 Division and Modulus

In SEDB, we cannot directly perform the division or modulus on the shares in the backend database servers. Certain transformations of the operands are necessary to support the division and modulus operations. In particular, to calculate v_1/v_2 or $v_1 \,\%\, v_2$, we propose to generate $v_1^{'} = t_1 * v_1 + t_1 * t_2 * v_2$ and $v_2^{'} = t_1 * v_2$, where t_1 and t_2 are non-zero integers chosen randomly from $[-(p-1)/2, (p+1)/2)$. Then we can represent the division and modulus of v_1 and v_2 as a combination of addition, subtraction and multiplication operations on v_1, v_2, and $v_1^{'}/v_2^{'}$, i.e., $v_1/v_2 = v_1^{'}/v_2^{'} - t_2$, $v_1 \,\%\, v_2 = v_1 - v_1^{'}/v_2^{'} * v_2 + t_2 * v_2$. If v_1 or v_2 is a constant, we can create a dummy column firstly, then the computing is the same as the division and modulus on existing values in the database.

Division: The SEDB plug-in and the SEDB coordinator cooperate to generate the random transformations $v_1^{'}$ and $v_2^{'}$ for v_1 and v_2 respectively, and compute the value of v_1/v_2 based on $v_1^{'}$ and $v_2^{'}$. In particular, the SEDB plug-in first chooses three different polynomials of degree 1 to share t_1, t_2 and $t_1 * t_2$, then sends the encrypted shares to the SEDB coordinator. The SEDB coordinator forwards the shares to corresponding backend database servers and invokes a UDF Div1 at each database server to calculate its shares of $v_1^{'}$ and $v_2^{'}$. With $([share_i(t_1)]_{k_{c,si}}, [share_i(t_2)]_{k_{c,si}}, [share_i(t_1 * t_2)]_{k_{c,si}})$, the server i updates the shadow of v_1 with $share_i(v_1) * share_i(t_1) + share_i(v_2) * share_i(t_1 * t_2)$, and the shadow of v_2 with $share_i(v_2) * share_i(t_1)$, encrypts the shadows for the other two servers, and returns the result to the SEDB coordinator. Then the SEDB coordinator forwards the encrypted shadows to the corresponding servers and invokes a UDF Div2 in each database server to reconstruct $v_1^{'}$ and $v_2^{'}$. As a result, each server calculates its share of v_1/v_2 as $v_1^{'}/v_2^{'} - share_i(t_2)$.

Modulus: To calculate $v_1 \,\%\, v_2$, the SEDB plug-in and the SEDB coordinator take a similar process to prepare the Shamir's (2, 3)-threshold shares of t_1, t_2, $t_1 * t_2$, and update the shadows of v_1 and v_2 by invoking a UDF Mod1 at three database servers. In the execution of Mod1, each server invokes NewMul1 to generate the share vector for $t_2 * v_2$. Then, the SEDB coordinator invokes a UDF Mod2 at each server, which generates its share of $t_2 * v_2$ by invoking NewMul2, recovers $v_1^{'}$ and $v_2^{'}$, and generates its share of $v_1 \,\%\, v_2$ by calculating $share_i(v_1) - v_1^{'}/v_2^{'} * share_i(v_2) + share_i(t_2 * v_2)$.

3.1.6 Comparison

As the shares of the confidential values are not order-preserving in SEDB, we perform the comparison by comparing the order-preserving transformations of the values. For example, to compare two values v_1 and v_2, we first compute $v_1^{'} = t_1 * v_1 + t_2$ and $v_2^{'} = t_1 * v_2 + t_2$, where t_1 is randomly chosen from $(0, (p+1)/2)$ and t_2 from $[-(p-1)/2, (p+1)/2)$. As a monotonic transformation, the order of $v_1^{'}$ and $v_2^{'}$ determines the order of v_1 and v_2.

To calculate $v_1^{'}$ and $v_2^{'}$, the SEDB plug-in prepares the three shares for t_1 using a polynomial of degree 1 and a polynomial of degree 2 for t_2, and then sends the encrypted shares to the SEDB coordinator, which further forwards $([share_i(t_1)]_{k_{c,si}}, [share_i(t_2)]_{k_{c,si}})$ to server i as the input of a UDF Compare1. Each backend database server i executes Compare1 by computing $share_i(t_1) * share_i(v_1) + share_i(t_2)$ and $share_i(t_1) * share_i(v_2) + share_i(t_2)$, encrypts the results for other two servers, and returns the encrypted results to the SEDB coordinator. After collecting the results from all three backend database servers, the SEDB coordinator invokes a UDF Compare2 to reconstruct $v_1{'}$ and $v_2^{'}$ at each server using the (3, 3)-threshold secret sharing schemes, for comparison.

To compare a confidential value with a constant *const*, the SEDB plug-in firstly computes $t_1 * const + t_2$, encrypts it, and sends it to each server, which compares it with the same transformations of the confidential value.

3.2 Discussions

SEDB supports complex SQL queries that contain a combination of the above operators. The SEDB plug-in needs to determine the order of operations, and prepares the parameters for all the operators and sends it in one SQL query to the SEDB coordinator. The SEDB coordinator parses the received query, extracts the parameters for each UDF, and then invokes multiple UDFs at the backend database servers in order.

SEDB supports aggregation queries. SEDB processes the `count()` function similarly as in the original DBMS. To calculate the `sum()` of data in a particular column, the SEDB coordinator asks all servers to return the summation of their shares for all the data in that column, and returns the result to the SEDB plug-in for recovering the summation of that column. From the results of `sum()` and `count()`, the SEDB plug-in can calculate the `average` of a particular column. To process the `min()`, `max()`, `group by`, or `order by` operations on a column, the SEDB coordinator invokes UDFs `Compare1` and `Compare2` to update the shadow of that column using its order-preserving transformation, and executes the `min()`, `max()`, `group by` and `order by` functions on the shadow column.

SEDB supports the `join` of columns as well. The process of `join` is similar to the comparison, that is, we process the `join` on the shadow columns, which is the same order-preserving transformations of the selected columns.

Since many SQL operators process NULL differently from the execution on non-NULL values, SEDB stores the NULL values in plaintext. Finally, SEDB is limited in supporting certain DBMS mechanisms such as the transactions and indexing in its current version for the complexity in multi-round processing between the SEDB coordinator and the backend database servers.

4 Security Analysis

In this section, we analyze the security of SEDB briefly. In SEDB, the sensitive data is split using Shamir's threshold secret sharing scheme. The adversaries and SEDB coordinator can never obtain any plaintext share, as they have no keys to decrypt the transmitted encrypted shares, thus they can never infer the sensitive data. Each database server cannot reconstruct the sensitive data from its local shares, as it owns only one share for each data. Moreover, the servers are assumed to be honest, they never collude with each other to acquire enough shares to reconstruct the data. In the following, we show that the database servers cannot infer the sensitive data from the process of the operations, either.

In SEDB, the servers with no priori knowledge of the data and queries, cannot infer any sensitive data during the executions. To process addition, each server summarizes its local shares and gains no information from others. To process multiplication, each server cooperatively completes the degree reduction. As analyzed in [5], the generated polynomial is random, and each server owns only one share of the multiplication. To process division and modulus, each server obtains the transformations of two confidential values v_1 and v_2, such as $t_1 * v_1 + t_1 * t_2 * v_2$ and $t_1 * v_2$. As the randomly chosen t_1 and t_2 are different in different processes and never reconstructed, each server cannot deduce v_1 and v_2 from the transformations. Each server determines the order of two confidential values v_1 and v_2 by the order-preserving transformation $t_1 * v_1 + t_2$ and $t_1 * v_2 + t_2$. As t_1 and t_2 are chosen randomly and differently each time, and never reconstructed, the servers cannot infer v_1 and v_2 or other statical information (e.g., $v_1 - v_2$, v_1/v_2) from the transformations.

SEDB prevents the curious servers inferring the sensitive data from priori knowledge of the data and queries. The server may attempt to gain the sensitive data from some known values (e.g., the minimum and maximum) or keywords in some special queries. As all data are split independently, and the statical relation of the values isn't preserved in their shares, the server cannot deduce unknown values from its local database. In the execution of addition and multiplication, each server only gains its share of the result, no useful information for statical attacks. To execute v_1/v_2 and $v_1 \% v_2$, each server obtains the transformations $v_1^{'} = t_1 * v_1 + t_1 * t_2 * v_2$ and $v_2^{'} = t_2 * v_2$. However, because t_1 (or t_2) is chosen independently in different executions, the adversarial server cannot infer t_1 (or t_2) from multiple executions; in one execution, the adversarial server cannot deduce v_2, t_1 or t_2 even if it knows v_1, $v_1^{'}$ and $v_2^{'}$; it can either infer v_1, t_1 even if it knows v_2, $v_1^{'}$ and $v_2^{'}$. Therefore, the server cannot gain any unknown value from the known ones and their transformations in the process of division and modulus.

For comparison, SEDB can be extended to prevent statistical attacks. In the current version, with two known values (e.g. the maximal and minimum values) and their order-preserving transformations, an adversary can deduce the coefficients of the transformation, and thus infers other unknown confidential values in the same column from their transformations. To compare v_1 and v_2, we extend SEDB as follows: (1) we calculate $v_1^{'} = t_1 * v_1 + t_2$ and $v_2^{'} = t_1 * v_2 + t_2^{'}$, where t_2 and $t_2^{'}$ are chosen independently and randomly from $[1,\ r]$; (2) if $v_1^{'} - v_2^{'} > (r-1)$ or $v_2^{'} - v_1^{'} > (r-1)$, the order of v_1 and v_2 is determined by $v_1^{'}$ and $v_2^{'}$, otherwise, the shares of v_1 and v_2 are returned to the SEDB plug-in who recovers v_1 and v_2 for comparison. The value r is specified at the setup of the database, if r is set as the size of that column, all comparison are processed at the SEDB plug-in, which leaks no information at the cost of efficiency. In the extended version, the adversarial server cannot infer any unknown confidential value even it has some priori knowledge about the sensitive data, and knows the transformation of the unknown confidential value and t_1, as it cannot distinguish the exact value from r potential inverses of the transformation.

5 Performance Evaluation

We have implemented the prototype of SEDB, which consists of the SEDB plug-in, SEDB coordinator and backend database servers. The SEDB plug-in and SEDB coordinator each contain a C++ library and a Lua module. The library in the SEDB plug-in rewrites the queries from applications, constructs the results based on the results from the SEDB coordinator, splits and reconstructs the confidential values, encrypts and decrypts the shares. The library in the SEDB coordinator splits the query from the SEDB plug-in for each backend database server, issues the query based on the replies from the servers, and constructs the result for the SEDB plug-in. To provide transparent database service, we adopts MySQL proxy [11] which invokes our Lua module to pass queries and results to and from the C++ library in the SEDB plug-in. MySQL proxy is also deployed in the SEDB coordinator which invokes the Lua module to capture the queries from the SEDB plug-in, and passes them to the C++ library for further process. Each backend database server uses MySQL 5.1 as DBMS, we implement 15 UDFs at each server to complete the computation on the shares. The big prime p is 128 bits, and the X vector is set as $\{1, 2, 3\}$. The secret sharing scheme and AES are implemented using NTL [18].

All experiments ran with three database servers, one trusted client and one SEDB coordinator in an isolated 100 Mbps Ethernet. The database servers and SEDB coordinator ran on identical workstations with an Intel i7-3770 (3.4 GHz) CPU and 4 GB of memory. The application and SEDB plug-in were deployed in one physical machine with an Intel 2640 M (2.8 GHz) CPU and 4 GB of memory. The operating systems of all the nodes are Ubuntu 12.04. Each database server maintains a table *test*, which sets *id* as the primary key and has two attributes $attr1$ and $attr2$.

We evaluated the processing overhead for ensuring confidentiality of sensitive data in SEDB, by comparing each operation's processing time in SEDB with it in the original MySQL. We measured the average processing time by issuing a SQL query with a single operation 100 times, to operate on *test* with different numbers of rows (denoted as n). For better comparison, we classify the operations into two classes according to the number of needed communication rounds between the SEDB coordinator and each database server. The operations select, insert, addition and subtraction, need only one round and belong to the first class. The remaining ones need two rounds and are categorized into the second class.

Table 1 lists the ratio of the processing time in SEDB to that in MySQL for the operations in the first class. To evaluate the processing time for insert, select and addition, we use SQL queries "insert into $test$($attr1$,$attr2$) values (u_1,u_2)" (u_1 and u_2 are random values), "select * from *test*" and "update *test* set $attr1 = attr1{+}attr2$" respectively. As illustrated in Table 1, the overhead for operations in the first class is modest. For insert and addition, the processing overhead is independent of n, and is at most $3.6\times$ and $2.31\times$ respectively when $n \leq 10000$. The main sources of SEDB's overhead for insert and addition is the message transmitting, as the critical communication path is 3 in SEDB and 1 in MySQL.

The processing overhead for `select` increases slightly as n increases, from $3.35\times$ when $n = 1$ to $7.33\times$ when $n = 10000$. The overhead is increasing mainly because the size of messages for the clients increases quicker in SEDB than in MySQL, as the SEDB plug-in has to receive three shares instead of the original data.

Table 2 illustrates the processing overhead for operations in the second class. We evaluate the processing time for multiplication, division, modulus and comparison, using SQL queries "update *test* set $attr1 = attr1 * attr2$", "update *test* set $attr1 = attr1/attr2$", "update *test* set $attr1 = attr1 \,\%\, attr2$" and "update *test* set $attr1 = attr1$ where $attr1 > attr2$" respectively. As among the existing solutions, only the ones based on fully homomorphic encryption can execute the multiplication at the servers, which introduces a overhead of $10^9\times$ [17], SEDB is a more practical solution, whose overhead for multiplication is at most $168.41\times$ when $n \leq 5000$. For the operations in the second class, the processing overhead

Table 1. The average response time for operations in the first class (in ms).

Num of rows		1	2	10	40	100	400	1000	3000	5000	10000
Insert	SEDB	5.00	5.01	5.34	5.37	5.06	4.99	5.47	5.70	5.73	5.86
	MySQL	1.45	1.65	2.28	1.82	1.34	1.72	1.19	1.24	1.11	1.29
	Ratio	3.45	3.04	2.34	2.95	3.78	2.91	4.62	4.60	5.18	4.56
Select	SEDB	3.23	3.19	3.72	5.38	8.33	23.11	46.97	129.95	200.17	364.87
	MySQL	0.74	0.85	0.91	1.31	1.52	3.47	6.53	14.72	28.33	43.80
	Ratio	4.35	3.75	4.07	4.10	5.49	6.66	7.19	8.83	7.07	8.33
Add	SEDB	4.65	4.55	4.79	5.11	5.82	8.81	15.56	32.60	51.44	93.79
	MySQL	1.47	1.46	1.59	1.54	2.16	3.73	6.08	12.08	18.45	35.47
	Ratio	3.17	3.11	3.00	3.31	2.69	2.36	2.56	2.70	2.79	2.64

Table 2. The average response time for operations in the second class (in ms).

Num of rows		1	2	10	40	100	400	1000	3000	5000
Mul	SEDB	9.01	10.82	18.14	41.24	87.53	273.35	742.00	1910.84	3185.17
	MySQL	1.77	1.73	1.78	1.96	2.43	3.85	5.68	12.47	18.80
	Ratio	5.09	6.27	10.17	21.00	36.03	70.92	130.73	153.19	169.41
Div	SEDB	9.10	14.36	24.95	75.50	149.92	451.35	1336.40	6918.89	15391.94
	MySQL	1.81	1.67	1.45	2.03	2.18	3.60	6.00	12.66	19.00
	Ratio	5.02	8.62	17.18	37.28	68.63	125.30	222.88	546.64	809.93
Mod	SEDB	10.94	16.05	45.31	135.35	296.01	1283.04	2417.34	6132.38	14270.56
	MySQL	1.73	1.80	1.33	1.94	2.00	4.05	6.84	15.00	27.33
	Ratio	6.31	8.92	33.94	69.65	147.77	316.71	353.22	408.84	522.18
Compare	SEDB	9.73	11.59	21.22	53.72	119.97	304.24	795.89	2545.82	5671.55
	MySQL	0.86	0.85	0.92	1.26	1.35	2.25	4.14	10.59	32.68
	Ratio	11.34	13.69	23.07	42.75	88.72	135.45	192.14	240.36	173.52

in SEDB increases with n, from a rather small one ($10.34\times$ for comparison when $n = 1$) to $808.93\times$ for division when $n = 5000$. It's mainly because, with the increase of n: (1) the time for the servers to complete the operations increases quicker in SEDB than in MySQL, as the servers in SEDB have to invoke UDFs n times; (2) the sizes of messages transmitted by the SEDB coordinator increase, the sizes are $9zn$, $15zn$, $24zn$, $12zn$ for multiplication, division, modulus and comparison, where z is the size of the share and is 16 bytes in our evaluation. The overhead can be reduced when the database servers and SEDB coordinator are deployed in the cloud, where the computing resources and bandwidth are increased remarkably.

6 Related Works

Hacigumus et al. [8] firstly proposed to provide the database as a service in 2002. They encrypt the database with a symmetric encryption, and then place it on the untrusted server. Hacigumus partitions the domain of each column into several disjoint subsets and assigns each partition with a unique partition id. When accessing the database service, the trusted client rewrites the query by replacing the confidential value with the index of the partition that the value belongs to. Therefore, the results from the server contains false ones which will require the clients to postprocess it. Moreover, this scheme doesn't support the aggregates well, for example, when the client wants to get summation of some column, it should acquire the entire column firstly, while each database returns the summation of the shares to the client (SEDB plug-in) directly in SEDB.

CryptDB [13] is a more practical solution. It uses different cryptosystems to support different types of queries, and maintains up to four columns for one column in the original database. CryptDB cannot ensure the consistency of the different shadow columns for one original column which makes the result of some queries false. For example, when an addition is executed on the column encrypted using Paillier [12], the data in other columns are stale. In SEDB, there is only one column that stores the shares which ensures the correctness of the results. Moreover, CryptDB cannot support SQL queries involving the multiplication or division as it does not adopt any cryptosystem that can handle multiplications.

MONOMI [17] extends CryptDB to support more SQL queries by splitting client/server execution of complex queries, which executes as much of the query as is practical over encrypted data on the server, and executes the remaining components on trusted clients, who will decrypt the data and process queries further. SEDB provides the transparent database service and doesn't require the modification on the clients. Cipherbase [2] provides secure database outsourcing services with the trusted hardware on the untrusted server. The trusted hardware executes arbitrary computation over the encrypted data.

Threshold secret sharing scheme has also been used to provide secure database outsourcing services. Existing works mainly focus on improving the performance for retrieval. Tian [16] builds a privacy preserving index to accelerate query. Agrawal [1] utilizes hash functions to generate the distribution polynomials to achieve efficient retrieval. AS5 [9] preserves the order of the confidential values in their shares by choosing the appropriate polynomials, to make query execution efficient. However, these schemes need to know the distribution of the data in advance, and doesn't support the update on the data.

7 Conclusion

In this paper, we propose a secure database oursourcing scheme (SEDB) to ensure the confidentiality of the outsourced database. SEDB is based on Shamir's threshold secret sharing scheme, in which the sensitive data is split into three shares and stored in three independent, honest-but-curious servers. In SEDB, the database servers execute functions over the shares without recovering the data. The execution is leaded by a honest-but-curious SEDB coordinator, which makes the servers cooperatively execute the operation. SEDB supports the functions including insert, select, addition, subtraction, multiplication, division, modulus and comparison, and provides continuous database outsourcing services.

Acknowledgments. Quanwei Cai, Jingqiang Lin and Qiongxiao Wang were partially supported by National 973 Program of China under award No. 2014CB 340603 and No. 2013CB338001, and the National Natural Science Foundation of China (Grant No. 61402470). Fengjun Li was partially supported by National Science Foundation under Award No. EPS-0903806 and matching support from the State of Kansas through the Kansas Board of Regents.

A Appendix

We describe a sample of the message exchanges in SEDB for each operation in Fig. 2. In particular, we assume the application operates on the table *test*, which sets *id* as the primary key, and has three attributes *attr*1, *attr*2 and *attr*3. The table *shadowtest* is the shadow of *test*.

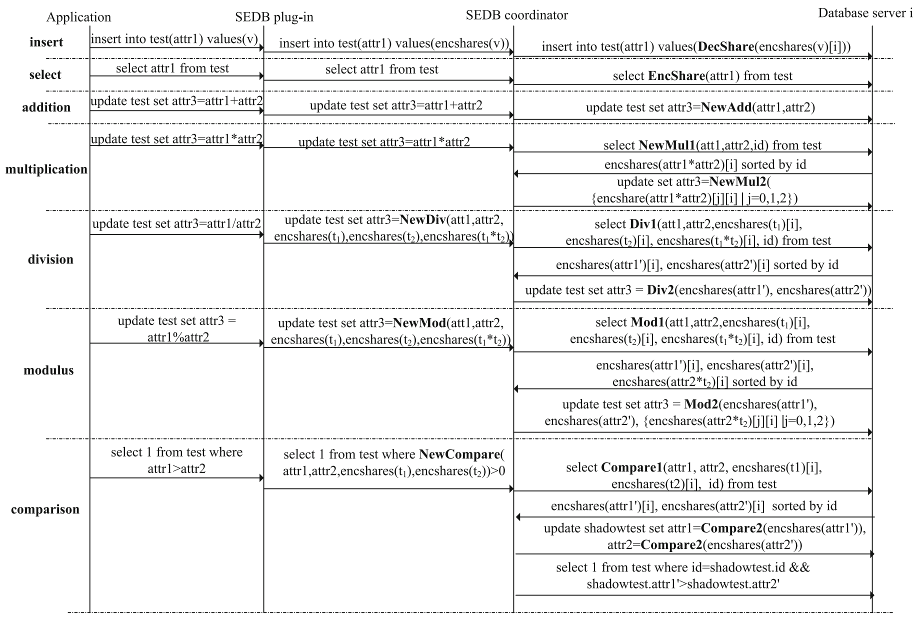
Application
SEDB plug-in
SEDB coordinator
Database server i
insert
insert into test(attr1) values(v)
insert into test(attr1) values(encshares(v))
insert into test(attr1) values(DecShare(encshares(v)[i]))
select
select attr1 from test
select attr1 from test
select EncShare(attr1) from test
addition
update test set attr3=attr1+attr2
update test set attr3=attr1+attr2
update test set attr3=NewAdd(attr1,attr2)
multiplication
update test set attr3=attr1*attr2
update test set attr3=attr1*attr2
select NewMul1(att1,attr2,id) from test
encshares(attr1*attr2)[i] sorted by id
update set attr3=NewMul2({encshare(attr1*attr2)[j][i] | j=0,1,2})
division
update test set attr3=attr1/attr2
update test set attr3=NewDiv(att1,attr2, encshares(t1),encshares(t2),encshares(t1*t2))
select Div1(att1,attr2,encshares(t1)[i], encshares(t2)[i], encshares(t1*t2)[i], id) from test
encshares(attr1')[i], encshares(attr2')[i] sorted by id
update test set attr3 = Div2(encshares(attr1'), encshares(attr2'))
modulus
update test set attr3 = attr1%attr2
update test set attr3=NewMod(att1,attr2, encshares(t1),encshares(t2),encshares(t1*t2))
select Mod1(att1,attr2,encshares(t1)[i], encshares(t2)[i], encshares(t1*t2)[i], id) from test
encshares(attr1')[i], encshares(attr2')[i], encshares(attr2*t2)[i] sorted by id
update test set attr3 = Mod2(encshares(attr1'), encshares(attr2'), {encshares(attr2*t2)[j][i] |j=0,1,2})
comparison
select 1 from test where attr1>attr2
select 1 from test where NewCompare(attr1,attr2,encshares(t1),encshares(t2))>0
select Compare1(attr1, attr2, encshares(t1)[i], encshares(t2)[i], id) from test
encshares(attr1')[i], encshares(attr2')[i] sorted by id
update shadowtest set attr1=Compare2(encshares(attr1')), attr2=Compare2(encshares(attr2'))
select 1 from test where id=shadowtest.id && shadowtest.attr1'>shadowtest.attr2'

Fig. 2. The message exchange in SEDB.

To execute insert, the SEDB plug-in replaces the confidential value v with its encrypted share vector $encshares(v)$, the SEDB coordinator invokes a UDF `DecShare` with the corresponding encrypted share $encshares(v)[i]$ at database server i, which decrypts the share and stores it in *test*. For select and addition, the SEDB plug-in doesn't modify the SQL queries, while the SEDB coordinator invokes a UDF `EncShare` to get the encrypted shares from server i to complete select, and a UDF `NewAdd` to make server i execute the addition of the columns.

In the process of multiplication, the SQL query is transmitted to the SEDB coordinator without modification, who firstly invokes a UDF `NewMul1` at each server i. Server i sorts the rows in *test* by *id*, generates the share vector for the multiplication of its local shares of values in columns $attr1$ and $attr2$ in order, then returns the encrypted share vectors $encshares(attr1*attr2)[i]$. On receiving the replies from three database server, the SEDB coordinator constructs the parameter for a UDF `NewMul2` as in Sect. 3.1.4, and invokes it to complete the degree reduction.

To execute division and modulus, the SEDB plug-in firstly chooses t_1 and t_2 for each row in *test* and generates the encrypted share vectors for each t_1, t_2 and $t_1 * t_2$, then it invokes `NewDiv` and `NewMod` at the SEDB coordinator. To process the `NewDiv`, the SEDB coordinator firstly invokes a UDF `Div1` to get the encrypted shares of $v_1^{'}$ and $v_2^{'}$ ($encshares(attr1^{'})[i]$ and $encshares(attr2^{'})[i]$) of all rows ordered by *id* in *test* from server i; then the SEDB coordinator combines them to generate $encshares(attr2^{'})$ and $encshares(attr2^{'})$, and invokes `Div2` to complete the division at each server. For `NewMod`, by invoking a UDF `Mod1`, the SEDB coordinator gains $encshares(attr1^{'})[i]$ and $encshares(attr2^{'})[i]$ as in the process of `Div1`, together with the encrypted share vector ($encshare$ $(attr2 * t_2)[i]$) of the multiplication of the corresponding local shares of $attr2$ with the share of t_2 from server i, where $encshare(attr2 * t_2)[i]$ is also ordered by *id*. Then, the SEDB coordinator recombines the $encshare(attr2 * t_2)[i]$ as in Sect. 3.1.4, and invokes a UDF `Mod2`. To execute `Mod2`, each database server firstly complete the degree reduction as in `NewMul2` to get its share of $attr2 * t_2$, then calculates its share of the modulus as described in Sect. 3.1.5.

To compare two columns in *test*, the SEDB plug-in invokes the `NewCompare` with the encrypted share vector of t_1 and t_2 at the SEDB coordinator. The coordinator firstly invokes a UDF `Compare1` at each server i, to get the encrypted shares of $t_1^{'}$ and $t_2^{'}$ of all rows ordered by *id* in *test* from server i, then combines them to update the *shadowtest* through a UDF `Compare2`, finally the coordinator sends a query to make each server complete the comparison on the *shadowtest*.

References

1. Agrawal, D., El Abbadi, A., Emekci, F., Metwally, A., Wang, S.: Secure Data Management Service on Cloud Computing Infrastructures. In: Agrawal, D., Candan, K.S., Li, W.-S. (eds.) Information and Software as Services. LNBIP, vol. 74, pp. 57–80. Springer, Heidelberg (2011)
2. Arasu, A., Blanas, S., Eguro, K., et al.: Secure database-as-a-service with Cipherbase. In: Proceedings of the 2013 international conference on Management of data, pp. 1033–1036. ACM (2013)
3. Boldyreva, A., Chenette, N., Lee, Y., O'Neill, A.: Order-preserving symmetric encryption. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 224–241. Springer, Heidelberg (2009)
4. Elmore, A.J., Das, S., Agrawal, D., El Abbadi, A.: Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In: Proceedings of SIGMOD, pp. 301–312. ACM (2011)

5. Gennaro, R., Rabin, M.O., Rabin, T.: Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In: Proceedings of the Annual ACM Symposium on Principles of Distributed Computing, pp. 101–111. ACM (1998)
6. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the 41st Annual ACM Symposium on Symposium on Theory of Computing (STOC), pp. 169–169. ACM Press (2009)
7. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 850–867. Springer, Heidelberg (2012)
8. Hacigümüş, H., Iyer, B., Li, C., et al.: Executing SQL over encrypted data in the database-service-provider model. In: Proceedings of SIGMOD, pp. 216–227. ACM (2002)
9. Hadavi, M.A., Damiani, E., Jalili, R., Cimato, S., Ganjei, Z.: AS5: a secure searchable secret sharing scheme for privacy preserving database outsourcing. In: Di Pietro, R., Herranz, J., Damiani, E., State, R. (eds.) DPM 2012 and SETOP 2012. LNCS, vol. 7731, pp. 201–216. Springer, Heidelberg (2013)
10. Halevi, S., Rogaway, P.: A tweakable enciphering mode. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 482–499. Springer, Heidelberg (2003)
11. Taylor, M.: MySQL proxy. https://launchpad.net/mysql-proxy
12. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 223–238. Springer, Heidelberg (1999)
13. Popa, R.A., Redfield, C., Zeldovich, N., Balakrishnan, H.: CryptDB: Processing queries on an encrypted database. Commun. ACM **55**, 103–111 (2012)
14. Shamir, A.: How to share a secret. Commun. ACM **22**, 612–613 (1979)
15. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceedings of IEEE S&P, pp. 44–55. IEEE (2000)
16. Tian, X.X., Sha, C.F., Wang, X.L., Zhou, A.Y.: Privacy preserving query processing on secret share based data storage. In: Yu, J.X., Kim, M.H., Unland, R. (eds.) DASFAA 2011, Part I. LNCS, vol. 6587, pp. 108–122. Springer, Heidelberg (2011)
17. Tu, S., Kaashoek, M.F., Madden, S., Zeldovich, N.: Processing analytical queries over encrypted data. In: Proceedings of the VLDB Endowment, vol. 6, pp. 289–300. VLDB Endowment (2013)
18. Shoup, V.: NTL: A library for doing number theory. http://www.shoup.net/ntl/