QSDB: An Encrypted Database Model for Privacy-Preserving in Cloud Computing

Guoxiu Liu[1,2], Geng Yang[1,3], Haiwei Wang[1], Hua Dai[1] and Qiang Zhou[2]

[1]School of Computer Science and Technology, Nanjing University of Posts and Telecommunications
Nanjing 210003, China
[e-mail: liuxiu.03@163.com]

[2] School of Computer and Information Engineering, Chuzhou University
Chuzhou 239000, China
[e-mail: liuxiu.03@163.com]

[3] Jiangsu High Technology Research Key Lab for Wireless Sensor Networks
Nanjing 210003, China
[e-mail: liuxiu.03@163.com]

*Corresponding author: Geng Yang

Received April 25, 2017; revised October 20, 2017; revised December 9, 2017; accepted February 28, 2018; published July 31, 2018

Abstract

With the advent of database-as-a-service (DAAS) and cloud computing, more and more data owners are motivated to outsource their data to cloud database in consideration of convenience and cost. However, it has become a challenging work to provide security to database as service model in cloud computing, because adversaries may try to gain access to sensitive data, and curious or malicious administrators may capture and leak data. In order to realize privacy preservation, sensitive data should be encrypted before outsourcing. In this paper, we present a secure and practical system over encrypted cloud data, called QSDB (queryable and secure database), which simultaneously supports SQL query operations. The proposed system can store and process the floating point numbers without compromising the security of data. To balance tradeoff between data privacy protection and query processing efficiency, QSDB utilizes three different encryption models to encrypt data. Our strategy is to process as much queries as possible at the cloud server. Encryption of queries and decryption of encrypted queries results are performed at client. Experiments on the real-world data sets were conducted to demonstrate the efficiency and practicality of the proposed system.

http://doi.org/10.3837/tiis.2018.07.021 **ISSN : 1976-7277**

Keywords: privacy-preserving; SQL query; encryption model; cloud computing; DAAS

1. Introduction

With the introduction of remote storage and cloud computing services, more and more companies are offering outsourced services based on databases [1,2], as witnessed by Amazon's RDS (relational database service) and Microsoft's SQL Azure. Today, many enterprises and end users may outsource their data to those cloud service providers for lower cost and better performance. In fact, a number of database systems on the cloud have been developed recently, which offer high availability and flexibility at relatively low costs. However, an adversary can exploit the vulnerability of software to gain access to sensitive information; curious or malicious administrators at those cloud service providers can snoop on private data and leak data. Undoubtedly, security and privacy of data are main concerns. To minimize the risk of data leakage, one approach to address these concerns is that the data owners opt to encrypt their sensitive data before outsourcing to the cloud database. However, encrypted data also brings significant difficulties in executing SQL and computing over these data. For example, encrypted data loses its original order without set of primitive operators, such as equality checks, order comparisons, aggregates (sums), and joins. It is obviously impractical to download encrypted data from cloud database and decrypt locally.

To date, researchers have designed theoretical solutions with fully homomorphic encryption [3] to address the above problem. However, because of expensive computational overhead, fully homomorphic encryption schemes are not practical for either cloud database providers or users. On the contrary, CryptDB's approach is to protect private data and support efficiently SQL query over encrypted data, and the key insight that makes it practical is that SQL uses a well-defined set of operators [4,5].

In an e-heathcare cloud system, the personal health data (e.g., blood glucose, insulin, and C-peptide levels) always contain real numbers, which are sent to the cloud database for storage and processing. However, CryptDB protects only integer values and computations on integers, which affects the accuracy of data and decision making results, and it may even lead to wrong diagnosis of a patient's illness. Moreover, CryptDB implemented certain operations by using user-defined functions (UDFs), which is not suitable in some cases. For example, user has no permission to create UDFs in the cloud database, and CryptDB uses mOPE (mutable order-preserving encoding) to support order operations. Although mOPE has ideal security, the interaction and tree balancing will affect its efficiency.

The objective of this paper is to design an efficient system called QSDB to address the issues mentioned above, which makes tradeoff between security, privacy and practicality.

We propose a new encryption model to implement the direct addition, multiplication, order comparison and equality without using user-defined functions. Therefore, QSDB can support floating point numbers calculation without compromising the privacy of sensitive data. Moreover, the frequent interactive communication between the cloud server and the client is not required. Once user issues a floating point calculation request, the client just encrypts the sensitive data, sends the query, and waits for the results from cloud server.

The rest of this paper is organized as follows. Related works is discussed in Section 2. Section 3 gives a brief introduction of the system and attack model. Section 4 presents the encryption model. Section 5 describes how the SQL query works. Section 6 gives security analysis. Section 7 presents the experiments and performance analysis. And Section 8 covers the conclusion.

2. Related Works

Search and queries over encrypted data. The problem of processing queries for the outsourced database is not new. The seminal paper [8] proposed executing SQL query over encrypted data by using bucketization. Since then, a number of techniques related to practical query processing over encrypted data have been proposed, including keyword search queries [9-14] and range queries [15-17]. CryptDB was the first system that can execute the SQL queries over encryption data [5]. Based on CryptDB, Tu designed and implemented a database encryption system that can divide task [18]. Surprisingly, to the best of our knowledge, none of the existing works can solve the SQL query over encrypted Database in Cloud Computing in a secure and efficient manner.

Privacy-preserving. Techniques used to protect sensitive data allow SQL query on encrypted data to be processed just like on plaintext. These methods can provide security by using order-preserving [19], which allows order-relation data items to be established based on their encrypted values without revealing the data itself, and it adopts efficient homomorphic encryption [7]. There are also some other related privacy-preserving methods proposed to ensure security in database [21–23].

Databases as service. There is an increasing interest in database-as-a-service in the cloud [4,20]. Based on the idea, we designed QSDB.

3. System and Encryption Model

3.1 System model

Fig. 1 describes the overall architecture. The trusted client receives queries from applications(1), and transforms the queries(2); it sends the queries to the untrusted cloud server(3), receives results (consisting of encrypted data)(4), decrypts the results by using corresponding keys, and sends the decrypted result to the applications(5). To balance tradeoff between data privacy protection and query processing efficiency, QSDB utilizes three different encryption models to encrypt data.

The system model in this paper involves two different entities: trusted client and untrusted cloud server.

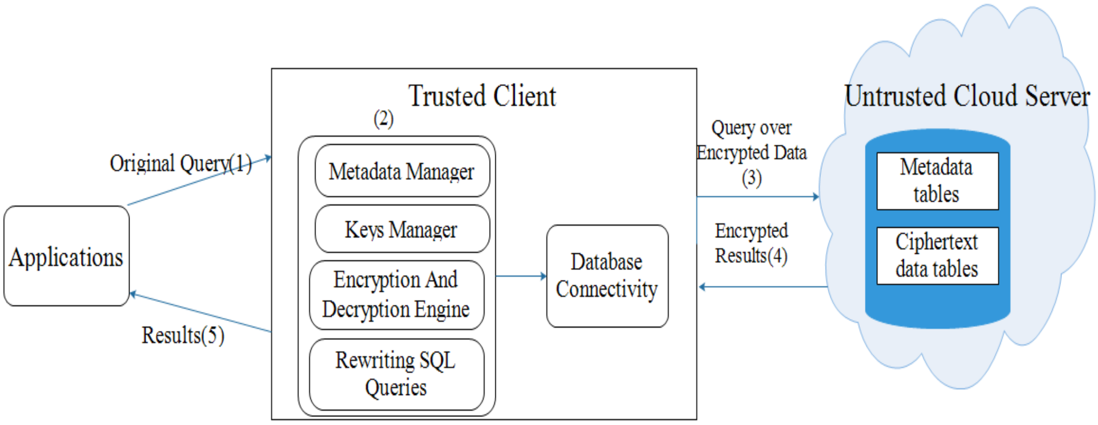


Fig. 1. QSDB architecture

Trusted client. In this system model, trusted client encrypts all data and stores encrypted data at the cloud server. Trusted client also translates the user SQL query to appropriate representation at the cloud server, and performs post-processing of cloud server query results. Its key components contain:

1. Rewriting SQL Query: Applications pose SQL query to the client, and it can parse the SQL statements and analyze the types of queries. Analytic functions should encrypt plaintext data and modify the name of column in the SQL statements according to types of queries, such as to create, select, insert, update, and delete. To illustrate how to rewrite a SQL query, consider the query select name from student, where id = 1, and the rewritten query is select name_EEM from student, where id_EEM = ‘Edaf···df’, column name_EEM corresponds to name, column id_EEM corresponds to id, and ‘Ed···f’ is the EEM (Equivalent encryption model) inner encryption of 1.

2. Encryption And Decryption Engine: Through three encryption models (such as equivalent encryption model, order-preserving encryption model, and homomorphic encryption model), the encryption engine encrypts plaintext data from applications. And the decryption engine decrypts the query results from the cloud server.

3. Keys Manager: There are two challenges in managing the keys. First of all, the keys manager should dynamically generate work keys for the equivalent encryption model. The second challenge is to obtain a key from the metadata table that combines metadata management for order-preserving encryption model and homomorphic encryption model.

4. Database Connectivity: The main function of database connectivity is responsible of connecting to the cloud database. And it can connect to different cloud databases.

5. Metadata Manager: Metadata Manager provides the data storage and access capabilities, which stores the attribute information of plaintext and the keys of order-preserving encryption model and homomorphic encryption model in metadata tables.

Untrusted cloud server. In this system model, an untrusted cloud server is hosted by the service provider who stores the encrypted database. The encrypted database is augmented with additional information which allows certain amount of query processing to occur at the cloud server without jeopardizing data privacy.

1. Metadata tables: Metadata tables store the attribute information of plaintext and the keys of order-preserving encryption model and homomorphic encryption model. For example, the structure of a metadata table is as shown in **Table 1**. The metadata table stores the information of table M1. Table M1 consists of three columns--column C1, column C2 and column C3, the types of columns are int, double and char respectively, and the names and types of columns are without encryption. For numeric data, the metadata tables store the keys of order-preserving encryption model and homomorphic encryption model, but for character data, the two columns are set to null. To protect the information of metadata tables, we encrypt the metadata tables with EEM.

2. Ciphertext data tables: Ciphertext data tables store encrypted data.

Table 1. The structure of a metadata table

Table	Column	Type	OEM Key	HEM Key
M1	C1	int	***	***
M1	C2	double	***	***
M1	C3	char	null	null

In our system, another challenge is to minimize the amount of confidential information revealed to the cloud server. To protect data privacy, sensitive data has to be encrypted before outsourcing. The simple and popular solutions adopted for data privacy are traditional encryption techniques, such as public key encryption and symmetric key encryption. However, traditional database encryption will destroy the data structure of original data, and it cannot support various kinds of SQL operations. We use three encryption models to encrypt privacy data, which can guarantee the security of the system, and also execute SQL query over encrypted database.

A practical QSDB should be efficient and provably secure, and it has the following challenges:

(1) Traditional encryption scheme can only encrypt positive integers and zero over a finite field. Therefore, QSDB can encrypt the floating point numbers without compromising the privacy of sensitive data.

(2)The SQL query can be processed over the encrypted data.

(3)In order to support processing of SQL query, the basic operations (e.g. addition, multiplication, and comparison) on ciphertext need to be constructed.

3.2 Attack model

Our model consists of a trusted client and an untrusted cloud server that interact with each other. The client encrypts the floating point numbers, and the server performs the SQL query over the encrypted data. We consider two types of attackers:

(1) Adversaries can access the database, which can see encrypted data and database structure, and launch ciphertext-only attack.

(2) Adversaries have more knowledge to guess the encryption details. In order to guess the encryption private key, they will try to launch chosen-plaintext attack.

In practical applications, the first type of adversaries is considered as the main threat. Because it is easy for curious adversary to obtain the ciphertext stored in cloud database, it is much harder to get the encryption private key in the cloud database environment. Therefore, the security against the ciphertext-only attack is our basic and practical security goal.

4. Encryption Model

In our system, we design the single-layer encryption and two-layer encryption models, and the main purpose of our design is to protect data privacy and enhance the efficiency of query processing. Let us describe the encryption models that QSDB uses, including equivalent encryption model, order-preserving encryption model, and homomorphic encryption model, as shown in **Fig. 2**. For each encryption model, we explain the security property that QSDB requires from it, its functionality, and how it is implemented.

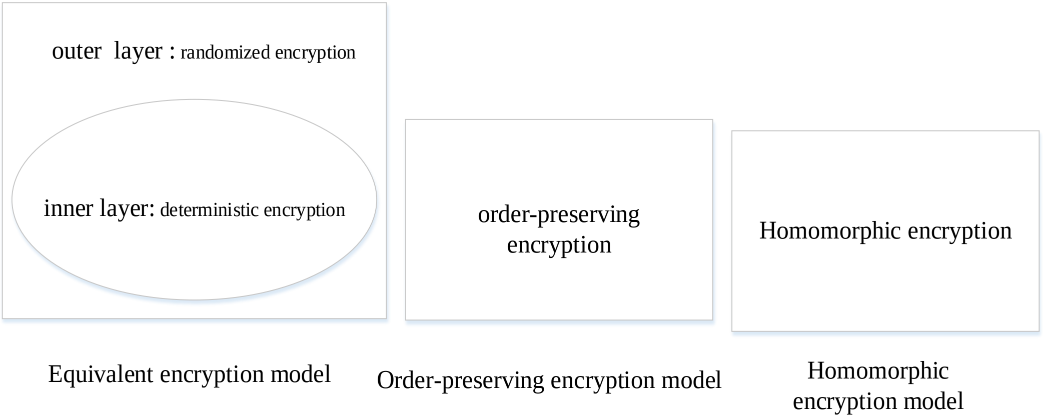

outer layer : randomized encryption
inner layer: deterministic encryption
order-preserving encryption
Homomorphic encryption
Equivalent encryption model
Order-preserving encryption model
Homomorphic encryption model

Fig. 2. Encryption Model

4.1 Equivalent encryption model (EEM)

Equivalent encryption model uses two layers of encryption: the outer layer is randomized encryption, and inner layer is deterministic encryption. In efficient construction of randomized encryption, a block cipher is used, such as AES with a random variable (*salt*). *Salt*, which is used by each data, consists of three parts: master key, column name and row identifier. The master key is a unique key for each user. The row identifier is used to indicate the current number of rows, and this value is not repeatable. Column name is a plaintext column name, which cannot be obtained directly from the database. Salt is generated by the client, and it is not saved in database. Deterministic encryption has a slightly weaker guarantee, yet it allows the cloud server to perform selects with equality predicates, equality JOINS, GROUP BY, COUNT, DISTINCT, etc.

To implement different levels of security strength and computational support, each value in the table is dressed in layer of deterministic encryption as the inner layer, and then, the inner layer is dressed in layer of randomized encryption as the outermost layer, as shown in **Fig. 2**. As the client receives SQL query from applications, it determines whether the layers of encryption need to be removed. The data maintains two encrypted states if selects are without equality predicates; otherwise, the client strips off the outermost layer to allow executing SQL query by sending corresponding outermost layer key to the cloud server. After SQL queries are completed, the inner layer can be re-encrypted immediately back to the outermost layer, and the data maintains the safest state. This property ensures that the QSDB’s security risks problem can be solved.

In our design, deterministic encryption uses a block cipher like AES with ECB mode**.** Master key and column name dynamically generate its key, and a trusted client uses the same key for encrypting values in the same column. We propose the brand new idea for generating keys, and the keys don’t need to be saved. For example, for user’s master key and column *c*, the client uses the key:

$$K_{mk,c,t} = f(MasterKey\ mk, Column\ c), \qquad (1)$$

where *f* is a pseudorandom permutation function. Then, QSDB can perform eq-joins with the same key.

4.2 Order-preserving encryption model (OEM)

Order-preserving encryption model uses single-layer encryption, and it encrypts the floating point numbers through order-preserving encryption. The model is used for numeric data encryption, and it supports range queries over encrypted data. In this strategy, we implement the nonlinear guaranteed order indexing scheme [19] proposed by Liu D. in 2013. Given the sensitivity sens of input values *v*, an order-preserving encryption scheme is a function *F*:

$$nindex_{[a,b,f]}^{sens}(v) = a \times f(v) \times v + b + noise, \qquad (2)$$

where noise is sampled from the range [0, $a*f(v+sens)*(v+sens)$ - $a*f(v)*v$], a>0; v denotes numeric plaintext and belongs to plaintext space; $f(v)$ is a nonlinear function about v, $f(v)>0$ for $v \neq 0$, $f(v1) \geq f(v2)$ for $v1 > v2 \geq 0$ or $v1 < v2 \leq 0$, and $f(v)$, a, b are keys. The scheme shows that the indexing process is irreversible. And this scheme not only keeps the order information in the plaintext, but has also overcome the security vulnerabilities of the linear order preserving indexing scheme proposed in 2012. Therefore, the model can ensure the security of the plaintext data, perform SQL queries over encrypted data directly and reduce the computation cost.

4.3 Homomorphic encryption model (HEM)

Homomorphic encryption model uses single-layer encryption, which adopts homomorphic encryption to encrypt the floating point numbers. Homomorphic encryption is a specially designed encryption scheme that allows direct computation over encrypted data and provides maximum security. To support SUM, AVG and multiplication query, we implemented D. Liu's cryptosystem [7]. The encryption scheme is designed with the following form, where *Enc* is the encryption algorithm of the encryption scheme,

$$Enc(v, k(n)) = (c_1, ..., c_n), \tag{3}$$

v is a floating point numbers to be encrypted, n is the number of sub-ciphertexts, and $k(n)$ is the key. The right side of the equation is the ciphertext, representing the plaintext after the encryption generated n sub-ciphertexts.

The sub-ciphertexts satisfy the following equation:

$$c_i = Value_i(k(n), v) + Noise_i(k(n), R), \tag{4}$$

where $Value_i(k(n),v)$ is a i th function over $k(n)$ and v, and $Noise_i(k(n),R)$ is a i th function over $k(n)$ and R. $Value_i(k(n),v)$ and $Noise_i(k(n),R)$ may have a linear time complexity with respect to n.

The decryption algorithm is defined as:

$$Dec(k(n), (c_1, ..., c_n)) = v.$$

HEM supports homomorphic addition and homomorphic multiplication. Without definition of special functions, the encryption algorithm can significantly reduce computation cost and perform aggregations on floating point values.

5. Executing SQL Query Over Encrypted Database

The client receives SQL query from applications, and it translates the SQL query. Next, we describe how QSDB supports SQL query over encrypted data. QSDB enables the cloud server to execute SQL query on encrypted data almost as if it were executing the same queries on plaintext data, such as table creation (create), insertion (insert), selection (select), updating (update) and deletion (delete). Existing applications do not need to be changed.

5.1 Basic SQL operations

For basic data operations (create, insert, select, update and delete), the QSDB replaces the plaintext with corresponding ciphertext encrypted by EEM, OEM and HEM. In other words, data is encrypted by using various encryption models to support different types of operations. At this point, the cloud server can learn nothing about the data other than the number of columns, rows, and data size. Following are some other notations.
COL – The columns, namely, the set of columns, denoted as COL={col1,col2,…,coln}.
COLID – COLID stores the identity of a column.
num – The number of columns.

5.1.1 Table creation

To illustrate how to translate encrypted SQL operations with three encryption models, consider the SQL operation create table table1 (column_1 numeric, column_2 char), where the data type of column column_1 is numeric, and the column column_1 is extended to column column_1_EEM, column column_1_OEM and column column_1_HEM. The values of column column_1_EEM, column_1_OEM and column_1_ HEM are encrypted by EEM, OEM and HEM respectively. The data type of column column_2 is char, which is encrypted by EEM, and corresponding column name is column_2_EEM (see **Fig. 3**). We describe the detailed table creation process in **Table 2**.

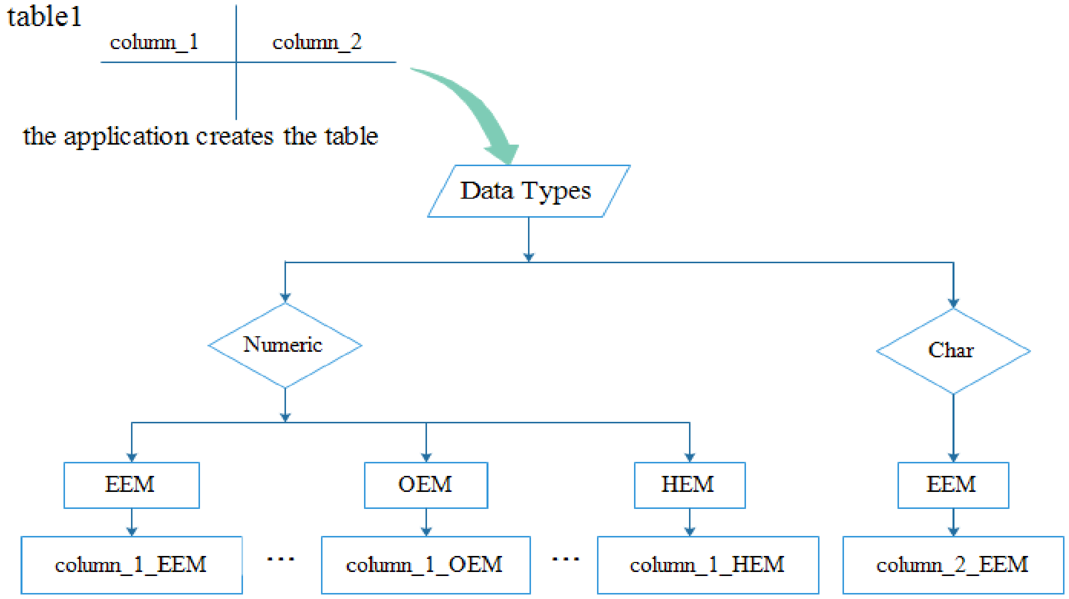


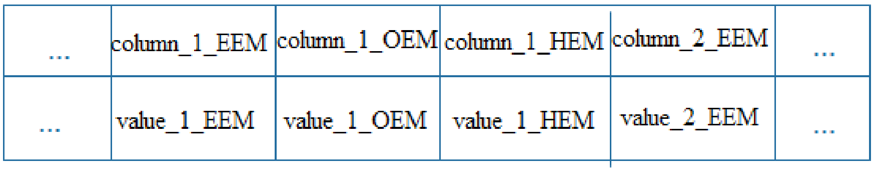
Fig. 3. When the application creates the table, the client uses different encryption model according to data type

Table 2. Algorithm for Table creation

```
1: The client receives a create statement from a user, parses the create statement, obtains
table_name, column_name, value_type, and stores these data in cloud database.
COL = {col1, col2, …, coln} with the identifiers COLID = {COLID|COLID = 1, 2, …, n}.
2: for each column COLID in COL do
        if data type is numeric then
            column_name is extended to column_EEM, column_OEM, and column_HEM;
            EEM(column_EEM) and set value_type to text;
            OEM(column_OEM) and set value_type to double;
            HEM(column_HEM) and set value_type to double;
        else
            if data type is char then
                set column_name to column_EEM;
                EEM(column_EEM) and set value_type to text;
                increase a row identifier as a primary key;
            end if
        end if
    end for
3: return the rewritten create statement.
```

5.1.2 Insertion

If the application issues the query insert into table1(column_1, column_2) values(value_1, value_2), where the data type of column column_1 is numeric, and the value_1 is encrypted by EEM, OEM and HEM respectively. The data type of column column_2 is char, and the value_2 is encrypted by EEM (see **Fig. 4**). Note that the client must request the data types of columns and encryption keys from a metadata table according to corresponding table name. **Fig. 5** describes the detailed flow of inserting process, and we present the detailed inserting process in **Table 3**.



…	column_1_EEM	column_1_OEM	column_1_HEM	column_2_EEM	…
…	value_1_EEM	value_1_OEM	value_1_HEM	value_2_EEM	…

Fig. 4. The encrypted values layout at the cloud database, ciphertexts shown are not full-length

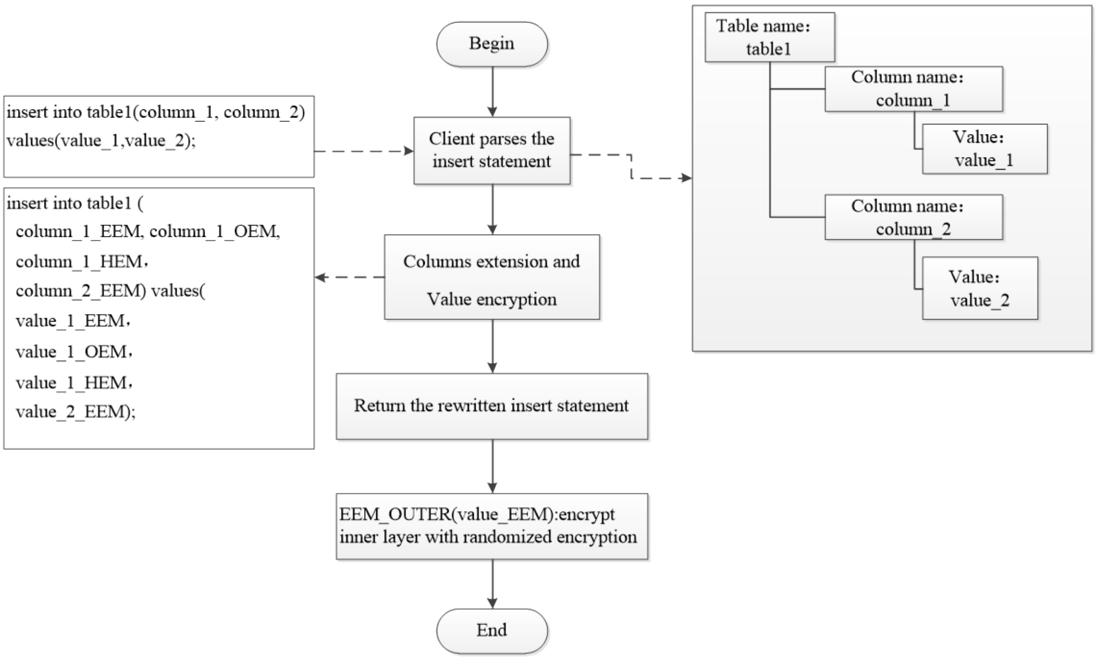

Begin
insert into table1(column_1, column_2) values(value_1,value_2);
Client parses the insert statement
Table name: table1
Column name: column_1
Value: value_1
Column name: column_2
Value: value_2
insert into table1 (column_1_EEM, column_1_OEM, column_1_HEM, column_2_EEM) values(value_1_EEM, value_1_OEM, value_1_HEM, value_2_EEM);
Columns extension and Value encryption
Return the rewritten insert statement
EEM_OUTER(value_EEM):encrypt inner layer with randomized encryption
End

Fig. 5. The detailed flow of inserting process

Table 3. Algorithm for Insertion

```
1: The client receives an insert statement from a user, parses the insert statement, obtains table_name, column_name, and value.
COL = {col1, col2, …, coln} with the identifiers COLID = {COLID|COLID = 1, 2, …, n}, num = 0;
2: obtains the column's data type from the cloud database according to the table_name;
3: for each column COLID in COL do
     if data type is numeric then
        column_name is extended to column_EEM, column_OEM, and column_HEM;
        EEM(value) and the value is value_EEM after encrption;
        OEM(value) and the value is value_OEM after encrption;
        HEM(value) and the value is value_ HEM after encrption;
        num++;
     else
          if data type is char then
             set column_name to column_EEM;
             EEM(value) and the value is value_EEM after encrption;
             num++;
          end if
     end if
   end for
4: return the rewritten insert statement;
5: for i = 1,…,num   do
        EEM_OUTER(value_EEM) with key generation according to the user's master key, the column name and unique row identifier;
   end for
```

5.1.3 Selection

If the next SQL operation is select column_1 from table1 where column_2 = 'value_2', it requires lowering the encryption of column_2 to inner layer. To execute this operation, the client first requests the data types of columns and encryption keys from a metadata table according to corresponding table name, and decrypts the outer layer using the key, which is based on the user's master key; a column name and a row identifier are generated, and a row identifier is unique, column_2_inner = DEC_EEMOUTER($K_{i,j,l}$), where *i* is user's master key, *j* is column_2 name, and *l* is row identifier. The client then issues the query select column_1_EEM from table1 where column_2_inner = value_2_inner, where value_2_inner is the EEM inner encryption of "value_2" with key $K_{mk,c,t}$. Finally, the client decrypts the result from the cloud server by using keys $K_{mk,c,t}$ and $K_{i,j,l}$, obtains the result value_1, and returns it to the application. **Fig. 6** presents the detailed flow of selecting process, and **Table 4** describes the detailed selecting process.

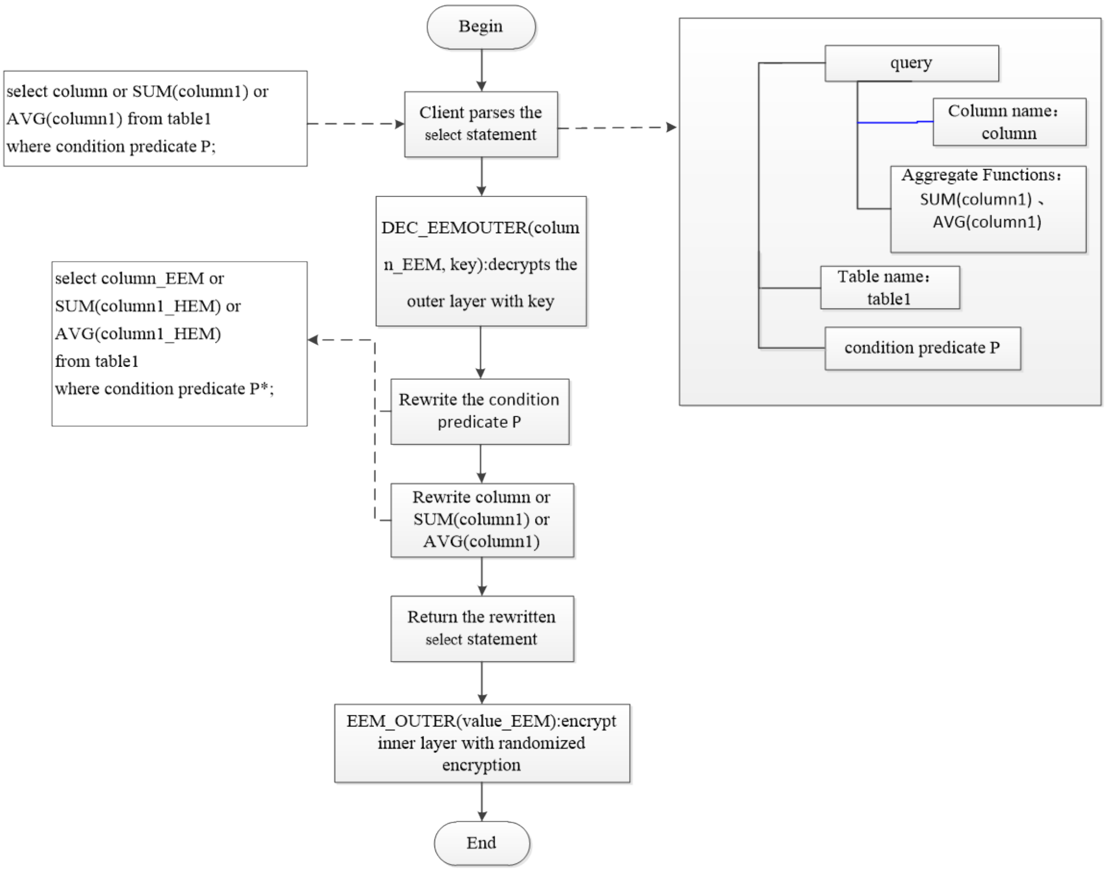

Begin
select column or SUM(column1) or AVG(column1) from table1 where condition predicate P;
Client parses the select statement
query
Column name: column
Aggregate Functions: SUM(column1) 、AVG(column1)
Table name: table1
condition predicate P
DEC_EEMOUTER(column_EEM, key):decrypts the outer layer with key
select column_EEM or SUM(column1_HEM) or AVG(column1_HEM) from table1 where condition predicate P*;
Rewrite the condition predicate P
Rewrite column or SUM(column1) or AVG(column1)
Return the rewritten select statement
EEM_OUTER(value_EEM):encrypt inner layer with randomized encryption
End

Fig. 6. The detailed flow of selecting process

Table 4. Algorithm for Selection

```
1: The client receives a select statement from a user, parses the select statement, obtains table_name, column_name or SUM(column1_name) or AVG(column1_name), condition predicate P;
2: obtains the column's data type and key from the cloud database according to the table name;
3: DEC_EEMOUTER(column_EEM, key) with key generation according to the user's master key, the column name and unique row identifier;
4: if P is equivalent match then
       set column_con to column_conEEM;
       EEM(value_constant) with the key same as the column_con's key;
   else
       if P is range queries then
           set column_con to column_conOEM;
           OEM(value_constant) with the key same as the column_con's key;
       end if
   end if
5: if query is column_name then
   set column_name to column_EEM;
   else
       if query is SUM(column1_name) or AVG(column1_name) then
           set column1_name to column1_HEM;
       end if
   end if
6: return the rewritten select statement;
7: EEM_OUTER(column_EEM) with key generation according to the user's master key, the column name and unique row identifier.
```

5.1.4 Updating

If the application issues the query update table1 set column_1 = value where column_2 > ‘value_2’, it will be interpreted to “update table1 set column_1_EEM = value_EEM（column_1_OEM = value_OEM, column_1_HEM = value_HEM）where column_2_OEM > ‘value_2_OEM’ ”**.** The data type of column column_1 is numeric, the value_1 is encrypted by EEM, OEM and HEM respectively, and the value_2 is encrypted by OEM. **Fig. 7** presents the detailed flow of updating process, and **Table 5** describes the detailed updating process.

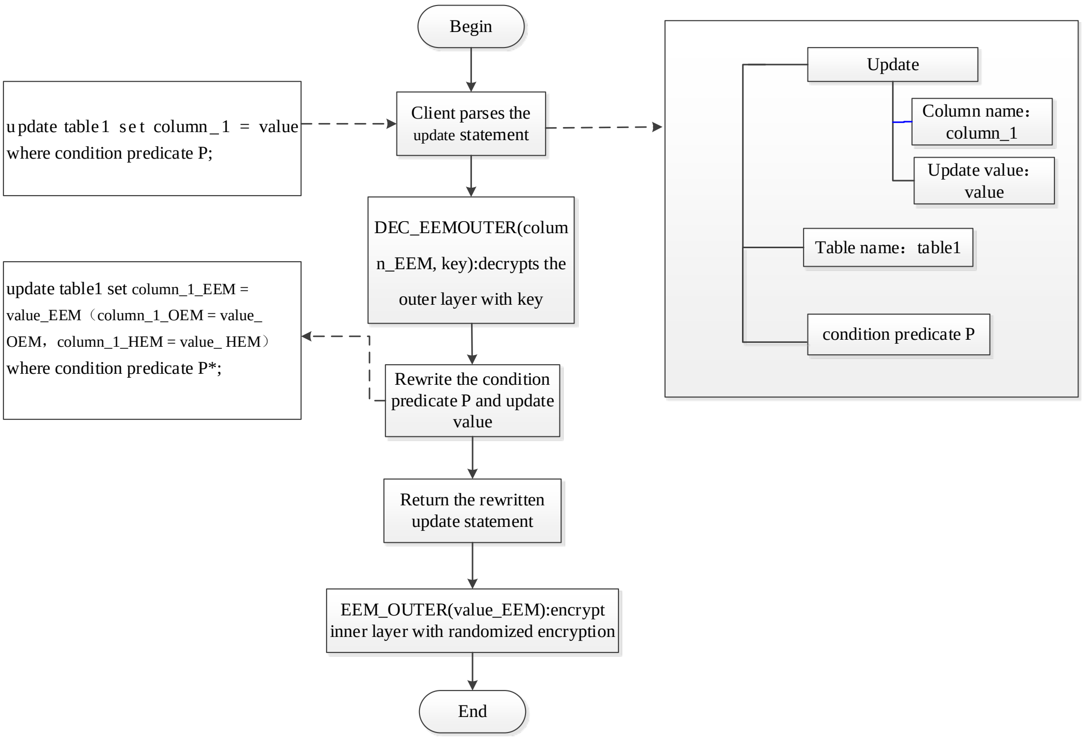

Begin
update table1 set column_1 = value where condition predicate P;
Client parses the update statement
Update
Column name: column_1
Update value: value
Table name: table1
condition predicate P
DEC_EEMOUTER(column_EEM, key):decrypts the outer layer with key
update table1 set column_1_EEM = value_EEM（column_1_OEM = value_OEM，column_1_HEM = value_ HEM）where condition predicate P*;
Rewrite the condition predicate P and update value
Return the rewritten update statement
EEM_OUTER(value_EEM):encrypt inner layer with randomized encryption
End

Fig. 7. The detailed flow of updating process

Table 5. Algorithm for Updating.

```
1: The client receives a update statement from a user, parses the update statement, obtains table_name, column_name, value_update, condition predicate P;
2: obtains the column's data type and key from the cloud database according to the table_name;
3: DEC_EEMOUTER(column_EEM, key) with key generation according to the user's master key, the column name and unique row identifier;
4: if set clause then
if data type is numeric then
      column_name is extended to column_EEM, column_OEM, and column_HEM;
      EEM(value_update) and the value_update is value_update _EEM after encrption;
      OEM(value_update) and the value_update is value_update _OEM after encrption;
      HEM(value_update) and the value_update is value_update _HEM after encrption;
else
       if data type is char then
         set column_name to column_EEM;
         EEM(value_update) and the value_update is value_update _EEM after encrption;
       end if
    end if
end if
5: if where clause then
     deal with condition predicate the same as Table 4;
  end if
6: return the rewritten select statement;
7: EEM_OUTER(column_EEM) with key generation according to the user's master key, the column name and unique row identifier.
```

5.1.5 Deletion

If the next SQL operation is delete from table1 where column_1 < ‘value_1’, it will be interpreted to “delete from table1 where column_1_OEM < ‘value_1_OEM’”, where the value_1 is encrypted by OEM. **Fig. 8** presents the detailed flow of deleting process, and we describe the delete processing in **Table 6**.

Table 6. Algorithm for Deletion.

```
1: The client receives a delete statement from a user, parses the delete statement, obtains table_name, condition predicate P;
2: obtains the column’s data type and key from the cloud database according to the table name;
3: if condition predicate P then
   DEC_EEMOUTER(column_EEM, key) with key generation according to the user’s master key, the column name and unique row identifier;
   deal with condition predicate the same as Table 4;
   end if
4: return the rewritten delete statement;
5: EEM_OUTER(column_EEM) with key generation according to the user’s master key, the column name and unique row identifier.
```

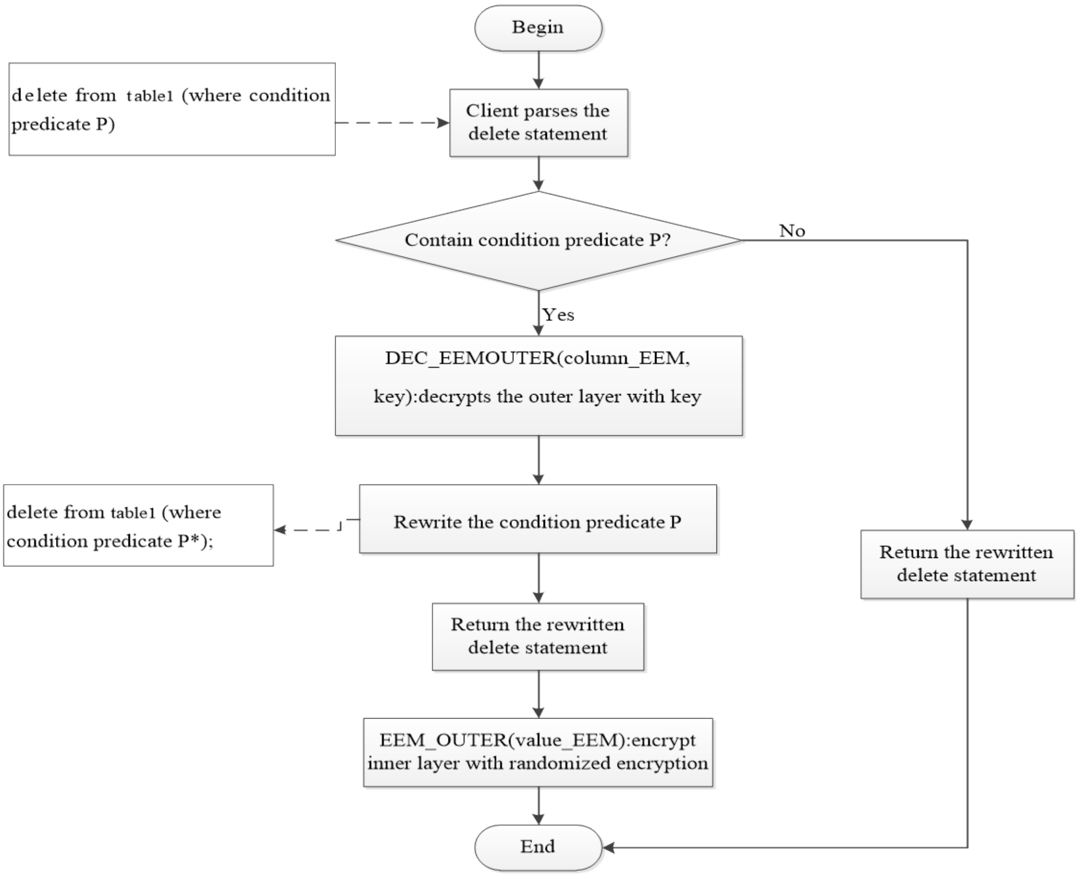

Begin
delete from table1 (where condition predicate P)
Client parses the delete statement
Contain condition predicate P?
No
Yes
DEC_EEMOUTER(column_EEM, key):decrypts the outer layer with key
delete from table1 (where condition predicate P*);
Rewrite the condition predicate P
Return the rewritten delete statement
Return the rewritten delete statement
EEM_OUTER(value_EEM):encrypt inner layer with randomized encryption
End

Fig. 8. The detailed flow of deleting process

5.2 Advanced queries

For range query (such as > and <), the data type of column is numeric, and the client applies OEM to encrypt plaintext data. For example, the application issues the query select column_2 from table1 where column_1 < 'value'; in which column_1 is for range query, the value is encrypted by OEM, and the query will be translated as select column_2_ EEM from table1, where column_1_OEM < 'value_OEM'.

For data x, y (x < y), we have OEM(x) < OEM(y), and OEM(x) means to encrypt x with OEM. Hence, the above translated SQL query will still work in the encrypted cloud database.

The QSDB supports two kinds of joins: eq-joins, in which the join predicate is based on equality, and ra-joins, which involve order checks. To perform eq-joins and ra-joins, the QSDB doesn't need to know in advance the pairs of columns that can be joined, and it can use EEM with the same key for each group of columns that are joined together.

6. Security Analysis

The security of QSDB mainly consists of the following parts: ciphertext-only attack and a particular chosen-plaintext attack. In our model, a trusted client sends the SQL query to an untrusted cloud server via a secure channel, we just store the ciphertext in the untrusted database, and as a result, the adversary can only get some ciphertext, but nothing else.

6.1 Ciphertext-only attack

We prove that our encryption models can be safe enough under ciphertext-only attack.

Equivalent encryption model (EEM). Equivalent encryption model uses two layers of encryption**:** the outer layer is randomized encryption, and inner layer is deterministic encryption. Randomized encryption provides maximum security without functionality in QSDB. It is indistinguishable under an adaptive chosen-plaintext attack, because the scheme is probabilistic, which means that two equal values are mapped to different ciphertexts with an overwhelming probability.

Order-preserving encryption model (OEM). An attacker cannot calculate the value of plaintext from the index value, and the same plaintext has different corresponding index values. Also, the encryption model can fuzz the statistical distributions of plaintext. Without leaking a, b and non-linear function $f(v)$, the scheme cannot reveal the plaintext to an attacker. Therefore, OEM can resist ciphertext-only attack.

Homomorphic encryption model (HEM). Based on R in equation (4), the encryption scheme ensures two encryptions of the same value with the same key have different results, because different random numbers are used in each encryption. In this way, the encryption scheme is robust against chosen-plaintext attack and chosen-ciphertext attack. Because the secret keys cannot be recovered by choosing the pairs of plaintexts and their ciphertexts due to noises in the each ciphertext, and more pairs contain more noises. We assume that an attacker cannot steal $k(n)$ and decryption functions at the same time, and plaintext cannot be recovered. Also, HEM can resist ciphertext-only attack.

6.2 A particular chosen-plaintext attack

We now define the particular chosen-plaintext attack, and we call it “discontinuous chosen-plaintext attack”. According to the definition, an adversary only gets k ciphertexts ($k < n$, n is the number of ciphertexts) in the same column.

If an adversary wants to obtain the ciphertexts of $\{v1, v2, \dots , vl\}$ ($l > n$) in the same column, it will only get $\{Enc(v1), Enc(v2), \dots , Enc(vk)\}$, which only contains k ciphertexts ($k < n$). Because $k < n$, and the encryption scheme ensures that two encryptions of the same value with the same key have different results due to different random numbers in each encryption, even if an adversary knows the decryption algorithm, it cannot guess the encryption privacy key. Then, QSDB can resist the particular chosen-plaintext attack.

7. Experimental Evaluation

We evaluated the performance of QSDB by conducting experiments on our prototype. The prototype was built based on the architecture as shown in **Fig. 1**. We implemented the proposed system with Java language in CentOS Linux operation system, and tested its efficiency and practicality on the real-world data set. The experimental results were obtained with an Intel Xeon CPU E3-1226 Processor (3.3GHz) and 16.0GB RAM, which has 4 processor cores.

To evaluate its performance, two issues are addressed. First of all, a focus of QSDB is the performance of encryption model for batch data encryption. Secondly, it is the performance of QSDB.

7.1 Evaluation of encryption model performance

In our system, we designed the single-layer encryption and two-layer encryption models, and the main purpose of our design is to protect data privacy and enhance the efficiency of query processing. We evaluated the performance of encryption model.

7.1.1 Performance evaluation

The encryption models include EEM, OEM and HEM. We programmed these encryption models and conducted experiment to determine their average execution time. The performance of the encryption model is shown in **Fig. 9** and **Table 7**. From them, we can get the following results:

(1) Microbenchmarks of cryptographic schemes, per unit of data encrypted (one 64-bit character data, one 64-bit floating point data) is measured by taking the average time over many iterations in **Table 7**. The average execution time of EEM is about 12 us, which is around 170 times of that of OEM.

(2) To understand the time cost of three encryption models, we measure the processing time of insert operations, which are encrypted by EEM, OEM and HEM respectively. As **Fig. 9** shows, two encryption models OEM and HEM have very close average execution time, and their average execution time is shorter than that of EEM. In the encryption models OEM and HEM, their time complexity is $O(n)$, thus their average execution time is very close to each other.

From the experimental results, we can see that the cryptographic overhead is relatively small**,** because most of our encryption models are efficient; **Fig. 9** shows their performance. EEM is the slowest, but it is faster than the encryption schemes of CryptDB. Thus, our encryption models are practically efficient and safe, and the system can meet needs of most applications.

Table 7. Execution time of encryption

Encryption model		Encrypt	Decrypt
EEM (8 char)	Inner layer	0.006ms	0.006ms
	Outer layer	0.006ms	0.006ms
OEM (1 double)		0.00007ms	N/A
HEM (1 double)		0.0001ms	0.00004 ms

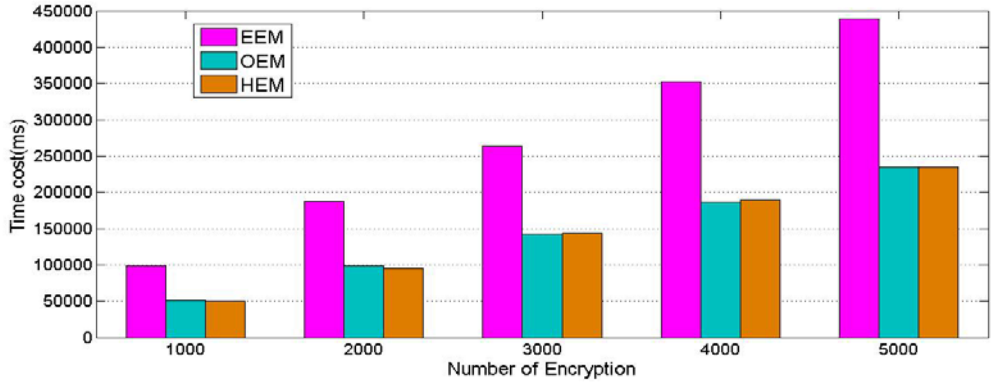


Fig. 9. Encryption model

7.1.2 Evaluation in web applications

The QSDB can be easily deployed in various kinds of database applications. In our evaluation, we used it to build a secure website based on the above implementation details. The webpage programming language is Java, and the web server is Apache. To construct the test platform, we used Java language to construct an application. Moreover, we selected Mysql as the database server.

To evaluate the performance in real web applications, we focus on comparing the execution time of SQL operations and QSDB operations. We tested the basic SQL operations of insert, select, update and delete. The execution time of SQL insert and delete operations by using QSDB is very close to the execution time of operations without encryption, as shown in **Fig. 10** and **Fig. 13**. In **Fig. 11**, with the increase of the number of select operation, the execution time of plaintext is getting closer to that of ciphertext. As shown in **Fig. 12**, we can see that the execution time of SQL update operation by using QSDB is longer than that without encryption. From the experimental results, we can see that the system is efficient and acceptable.

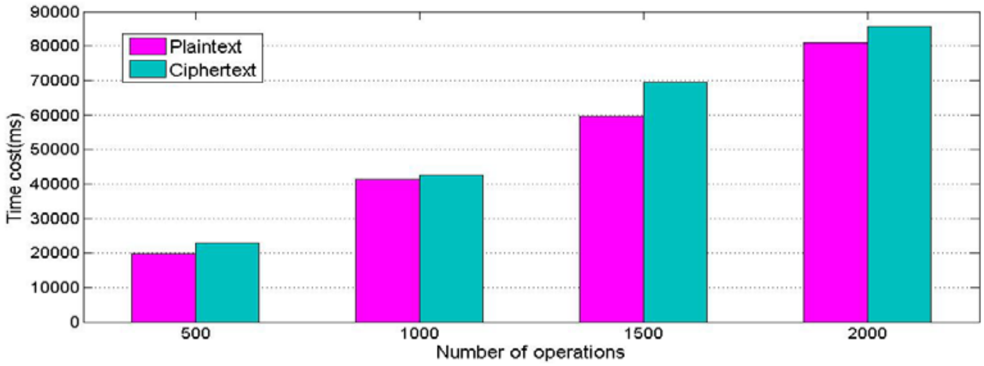


Fig. 10. Insert operation

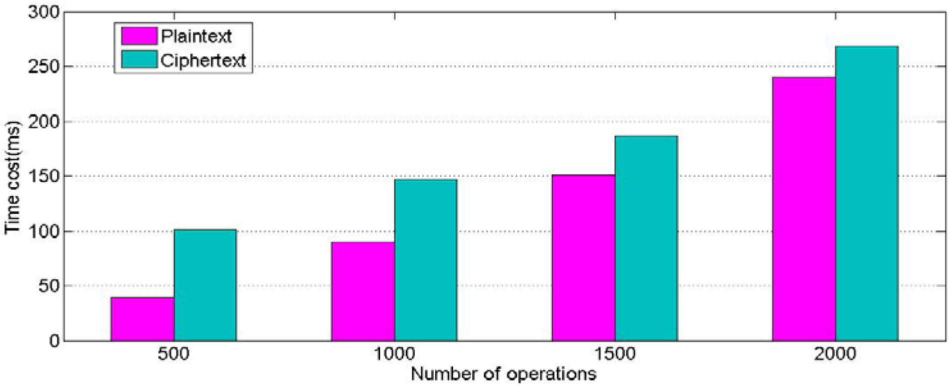


Fig. 11. Select operation

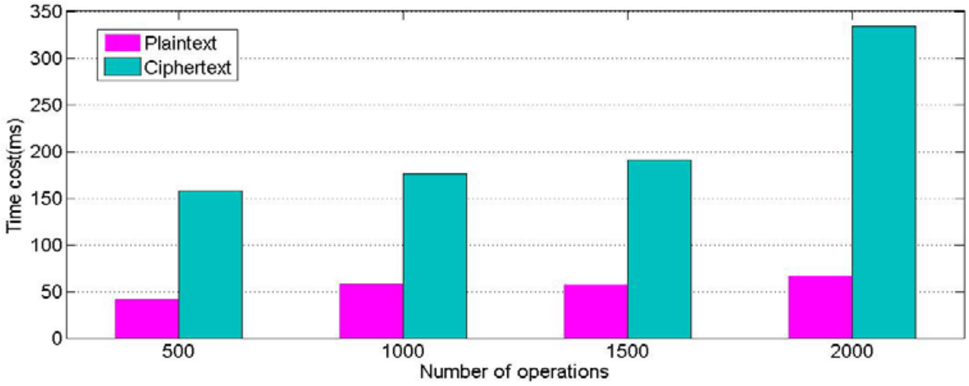


Fig. 12. Update operation

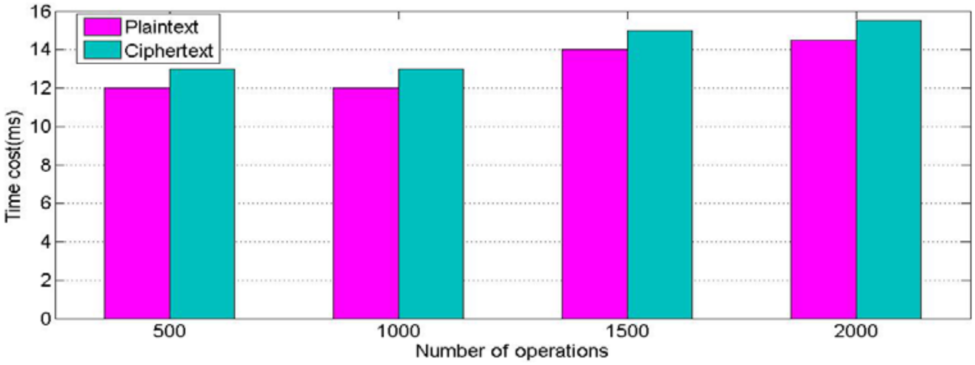


Fig. 13. Delete operation

7.2 Comparison of QSDB with CryptDB

We compared QSDB and CryptDB by using a simple synthetic data set on which different types of operations were executed. Both QSDB and CryptDB were implemented on the same platform, and they used the same machine configurations for the client and the server. CryptDB's idea is to encrypt each data item in one or more onions: that is, each value is dressed in layers of increasingly stronger encryption. Each layer of each onion enables certain kinds of functionality as explained in the previous subsection. For example, the outermost layers, such as RND and HOM, provide maximum security; whereas the inner layers, such as OPE, provide higher functionality.

Fig. 14 shows the results of seven types of SQL queries, and we make a few observations from it. The query processing cost of QSDB is much lower than that of CryptDB. Because QSDB employs two-layer and single-layer encryption models that allow more efficient computation on encrypted data, it reduces the encryption and decryption computation overhead. On the other hand, CryptDB uses one or more than two layers of onions to encrypt data. It is clear that computation based on more onions is generally slower than QSDB's schemes. For example, the range query cost of QSDB is much lower than that of CryptDB, because QSDB uses single-layer encryption model, and CryptDB uses two-layer onions. Thus, QSDB is more efficient than CryptDB.

Fig. 15 shows the space overheads of QSDB and CryptDB as the number of items are changed from 1000 to 100000. We compare the encryptions by using QSDB and CryptDB. They increase the amount of data stored in the DBMS, because they store multiple different encryption fields for the same field, and because ciphertexts are larger than plaintexts in some encryption schemes. Clearly, QSDB has lower space overhead, and the space overhead of CryptDB is more than three times higher than that of QSDB.

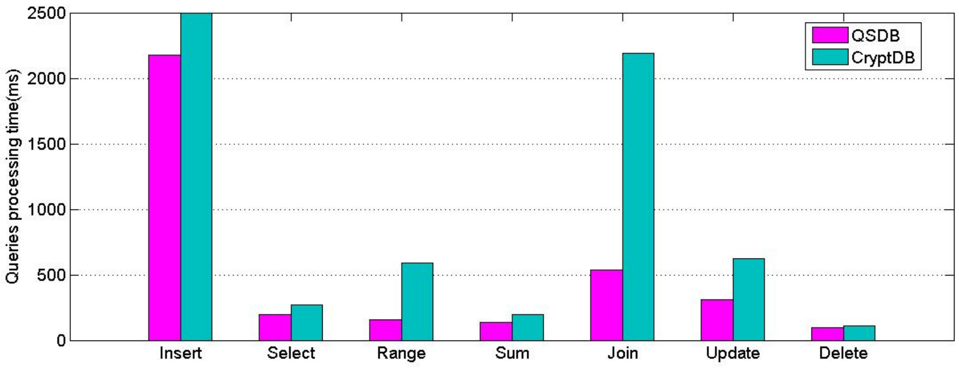


Fig. 14. Execution times of QSDB and CryptDB for the 7 sample queries

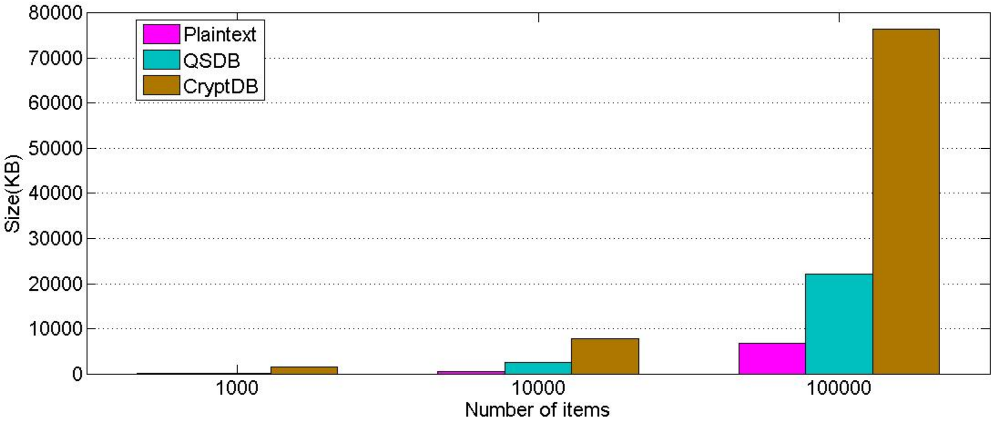


Fig. 15. Space Overheads

8. Conclusion

In this paper, we presented QSDB--a system that provides a practical and strong level of confidentiality for cloud computing environment. The QSDB allows a user to outsource encrypted floating point numbers to a cloud database for storage and processing. For privacy preservation, we propose three encryption models to minimize the information revealed to untrusted cloud server and reduce the encryption and decryption computation overheads. We also describe a new homomorphic encryption algorithm without definition of special functions to cut down the computation overhead. In addition, the system uses different levels of layered encryption, which can enable SQL query to be processed over encrypted data, and split the SQL query into a server query and a client query. The cloud database retains the responsibility to manage the persistence of data. The client gets total privacy, and the cost of cooperating in query execution with the cloud database. Theoretical security analysis and experimental evaluation by using real-world dataset were carried out to demonstrate the efficiency of our proposed scheme for practical application.

However, there are still many challenges in QSDB. Firstly, the QSDB uses three encryption models to encrypt privacy data, which can guarantee the security of the system, and it can also execute SQL query over encrypted database. But a combination of three encryption models can increase the computation cost and space overhead. Secondly, to evaluate the three encryption model’s performance, per unit of data encrypted (one 64-bit character data, one 64-bit floating point data) is measured by taking the average time over many iterations. It is not a best test solution. In the future works, we will try to improve the QSDB to handle these challenges.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (61572263, 61502251 and 61502243), the Project of Natural Science Research of Jiangsu University (14KJB520027，15KJB520027), the Natural Science Foundation of Jiangsu Province (BK20151511), the Natural Science Foundation of Anhui Province(1608085MF127), the Natural Science Foundation of Educational Commission of Anhui Province (KJ2016B17, KJ2015B19), the Postdoctoral Science Foundation of China (2015M581794), and the Postdoctoral Science Foundation of Jiangsu (1501023C).

References

[1] S. Aulbach, T. Grust, D. Jacobs, A. Keper, and J. Rittinger, "Multi-tenant databases for software as a service: schema-mapping techniques," in *Proc. of ACM SIGMOD International Conference on Management of Data*, Vol. 25, 2008, pp. 1195-1206. Article (CrossRef Link)

[2] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska, "Building a database on S3," in *Proc. of ACM SIGMOD International Conference on Management of Data*, Vol. 18, 2008, pp. 251-264. Article (CrossRef Link)

[3] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. of ACM Symposium on Theory of Computing*, Vol. 9, 2009, pp. 169-178. Article (CrossRef Link)

[4] C. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. R. Madden, et al, "Relational Cloud: A Database-as-a-Service for the Cloud," in *Proc. of CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings, pp. 235-240.

[5] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting confidentiality with encrypted query processing," in *Proc. of ACM Symposium on Operating Systems Principles 2011*, SOSP 2011, Cascais, Portugal, October, pp. 85-100. Article (CrossRef Link)

[6] P. Paillier, "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes," in *Proc. of Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques*, Prague, Czech Republic, May 2-6, 1999, Proceeding, Vol. 5, pp. 223-238. Article (CrossRef Link)

[7] D. Liu, "Homomorphic encryption for database querying," WO/2013/188929.

[8] Hacigümüş, Hakan, B. Iyer, C. Li, and S. Mehrotra, "Executing SQL over encrypted data in the database-service-provider model," in *Proc. of ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, June, 2002, pp. 216-227. Article (CrossRef Link)

[9] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proc. of IEEE Symposium on Security and Privacy*, 2012, pp. 44-55.

[10] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. C. Roşu, and M. Steiner, "Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries," *Advances in Cryptology–CRYPTO 2013, Springer Berlin Heidelberg*, 2013, pp. 353-373. Article (CrossRef Link)

[11] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," *Journal of Computer Security*, Vol. 19, 2011, pp. 895-934. Article (CrossRef Link)

[12] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," *Proceedings - IEEE INFOCOM*, Vol. 25, 2011, pp. 829-837. Article (CrossRef Link)

[13] B. Wang, S. Yu, W. Lou, and Y. T. Hou, "Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud," *Proceedings - IEEE INFOCOM*, 2014, pp. 2112-2120. Article (CrossRef Link)

[14] Z. Fu, X. Sun, Q. Liu, L. Zhou, and J. Shu, "Achieving efficient cloud search services: multi-keyword ranked search over encrypted cloud data supporting parallel computing," *IEICE Transactions on Communications*, Vol. E98.B, 2015, pp. 190-200.

[15] B. Dan, and B. Waters, "Conjunctive, Subset, and Range Queries on Encrypted Data," in *Proc. of The Theory of Cryptography Conference*, Vol. 4392, 2006, pp. 535-554. Article (CrossRef Link)

[16] B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu, "Secure multidimensional range queries over outsourced data," Vldb Journal International Journal on Very Large Data Bases, Vol. 21, 2012, pp. 333-358. Article (CrossRef Link)

[17] R. Li, A. X. Liu, A. L. Wang, and B. Bruhadeshwar, "Fast range query processing with strong privacy protection for cloud computing," *Pvldb*, Vol. 7, 2014, pp. 1953-1964. Article (CrossRef Link)

[18] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," *Proceedings of the Vldb Endowment*, Vol. 6, 2013, pp. 289-300. Article (CrossRef Link)

[19] D. Liu, and S. W. †, "Nonlinear order preserving index for encrypted database query in service cloud environments," *Concurrency and Computation Practice and Experience*, Vol. 25, 2013, pp. 1967–1984. Article (CrossRef Link)

[20] D. Agrawal, A. E. Abbadi, F. Emekci, A. Metwally, and S. Wang, "Secure Data Management Service on Cloud Computing Infrastructures," *New Frontiers in Information and Software as Services*, Springer Berlin Heidelberg, Vol. 74, 2011, pp. 57-80. Article (CrossRef Link)

[21] S. Bajaj, and R. Sion, "Trusteddb: a trusted hardware based database with privacy and data confidentiality," in *Proc. of SIGMOD*, 2011.

[22] A. H. M. S. Sattar, J. Li, X. Ding, J. Liu, and M. Vincent, "A general framework for privacy preserving data publishing," *Knowledge-Based Systems*, Vol. 54, 2013, pp. 276-287. Article (CrossRef Link)

[23] J. Li, Z. Liu, X. Chen, F. Xhafa, X. Tan, and D. S. Wong, “L-EncDB: A lightweight framework for privacy-preserving data queries in cloud computing,” *Knowledge-Based Systems*, Vol. 79, 2015, pp. 18-26.Article (CrossRef Link)

Guo-Xiu Liu, born in 1982. PhD at the School of Computer Science and Technology, Nanjing University of Posts and Telecommunications. Her current research interests include data security and privacy protection, and cloud computing security.

Geng Yang, born in 1961. Professor and PhD supervisor with the School of Computer Science and Technology, Nanjing University of Posts and Telecommunications. His current research interests include computer communication and networks, parallel and distributed computing, cloud computing security, and information security.

Hai-Wei Wang，born in 1989. Master at the School of Computer Science and Technology, Nanjing University of Posts and Telecommunications. His current research interests include data security and privacy protection, and cloud computing security.

Hua Dai, born in 1982. PhD and Associate professor with the School of Computer Science and Technology, Nanjing University of Posts and Telecommunications. His current research interests include database security, distributed data management and security.

Qiang Zhou，born in 1978. received his Ph.D. degree in the School of Computer Science and Technology from Nanjing University of Posts and Telecommunications in 2014. He is currently an Associate Professor at Chuzhou University. His research interests include wireless sensor networks, parallel and distributed computing, and information security.