Securing Outsourced Databases in the Cloud

Dongxi Liu

1 Introduction

Cloud database services, such as Amazon Relational Database Service (RDS) and Microsoft SQL Azure, are attractive for companies to outsource their databases. In cloud database services, a shared platform (e.g., database server hardware and software) is provided to host multiple outsourced databases. By using cloud database services, a client can deploy databases quickly without making the large upfront investment on proprietary hardware and software. Hence, the cloud database services can help companies reduce the total cost of ownership on their database management. Moreover, due to the scalability and elasticity of cloud database services, an enterprise can dynamically increase or decrease the cloud resources allocated to its databases according to its business requirements.

For databases deployed into a cloud database service, the service providers have the privilege to access the databases, since the underlying hardware and software are under their physical control. Hence, the databases in the cloud might be accessed improperly by the service providers accidentally or intentionally. The potential of such improper accesses causes the concern of users about the privacy of their outsourced databases. On the other hand, the underlying software of cloud database services (e.g., hypervisors, operating systems and DBMSs) might be compromised by attackers. At this case, the privacy of the outsourced databases is also at risk of being breached. Though attractive, cloud database services may not be fully exploited if the problem of data privacy cannot be satisfyingly addressed [2].

To protect data in cloud databases, a straightforward approach is to encrypt data before they are stored. By this way, the service providers or attackers can access only meaningless ciphertexts. However, after encryption, the databases may not be efficiently queried. It is not acceptable to decrypt the entire databases before executing

D. Liu (✉)
CSIRO Computational Informatics, Marsfield, NSW 2122, Australia
e-mail: dongxi.liu@csiro.au

S. Nepal and M. Pathan (eds.), *Security, Privacy and Trust in Cloud Systems*,
DOI: 10.1007/978-3-642-38586-5_9,

a query. If a database is large, decrypting the entire database will be unacceptably slow. In addition, if the decryption is done in the cloud, the encrypted database again becomes insecure. Ideally, a query should be executed directly over the encrypted database, producing the encrypted query result, which can only be decrypted by users.

There have been some encryption schemes and systems proposed to facilitate the encryption of databases and their queries [3, 7, 9, 16]. In the following, we describe several requirements for database encryption and query. These requirements are identified according to the role of databases in information systems. In an information system, a database may run for a very long period of time. During this period, the number of records in the database and the stored values may change dramatically. For example, a table containing staff personal information may contain only a few of staff records initially and then thousands of records after several years. And moreover, the staff salaries may increase from a few hundred dollars per week to a few thousand dollars per week.

R1: The native operations in DBMSs, such as SUM and AVG, should be used to support the operations on encrypted data, instead of using user-defined functions, since user-defined functions may not be optimized well by the DBMSs.

R2: The relational data model should be taken to manage encrypted data, and thus the existing DBMSs can be applied without worrying about their physical implementations (e.g., column-oriented DBMSs or row-oriented DBMSs).

R3: The providers of cloud database services should not need encryption keys. Otherwise, the database privacy is not protected against untrusted cloud database services.

R4: The maximum sum of values in one table column should not be predetermined, since it is hard to determine for a long-standing database.

R5: The number and range of values should not be required, since they may increase dramatically when a database runs for a long period of time.

As to be discussed in the next section, the existing works do not satisfy all the requirements. After discussing the exiting works, we will present an approach that satisfies all the above requirements.

2 Related Works

In this section, we introduce the security mechanisms deployed in the cloud database services, and the schemes of querying encrypted databases.

2.1 Security Mechanisms in Cloud Database Services

All cloud database services provide mechanisms to address the security concern of service users. The basic security scheme in cloud database services is access control [4]. Since a cloud database service is shared by multiple users, users must correctly

authenticate themselves to the service to access their own databases. However, the access control mechanisms cannot completely remove the security concern since users have to trust the cloud service providers to correctly implement the access control mechanisms and not to access the databases improperly. In addition, each cloud database service may provide its specific security scheme, as discussed below.

In the Oracle Database Cloud [14], all data are stored by using Oracle's Transparent Data Encryption scheme, which encrypts data stored on disk and in backups. The encryption and decryption of data is performed in the Oracle Database Cloud and transparent to users. The benefit is that users do not have to do extra work to encrypt their databases. However, the encryption keys are managed in the Oracle Cloud, so users must trust the service provider not to decrypt their data improperly. In the system presented later, the keys are managed by users themselves, so the cloud service providers cannot decrypt their data at any time.

The Amazon Relational Database Service (Amazon RDS) provides security mechanisms at the network level [1]. A user can control network access to his database by configuring firewall settings. Moreover, Amazon RDS allows database server instances to run in Amazon Virtual Private Cloud (Amazon VPC), which helps the isolation of database server instances. The databases in Amazon VPC can be accessed by the existing IT infrastructure of an enterprise through encrypted IPsec link.

Similarly, the Microsoft SQL Azure service [13] also controls network access to databases by configuring the SQL Database firewall. The firewall can be configured at the server level or at the database level. The server level firewall controls machines which can build connections with the virtual database servers. The database level firewall controls accesses to certain database instances in the virtual database servers. The communication with the Microsoft SQL Azure service is encrypted with SSL.

2.2 Related Schemes of Encrypting Database

The CryptDB [16] is a system supporting SQL queries over encrypted databases. This system needs the extension of existing DBMSs to support homomorphic operations like SUM and AVG, because the exploited homomorphic encryption scheme [15] performs multiplication on ciphertexts to get the sum of corresponding plaintexts. The existing DBMSs cannot natively support multiplication of values in one table column.

In Ref. [7], a mechanism of supporting aggregate queries is proposed, which is designed only for column-based databases by encrypting multiple values in one table column into one ciphertext. Hence, the mechanism in Ref. [7] is not flexible for data insertion and deletion, since the data to be updated is always packed together with other data not to be updated.

In Ref. [6, 12], a homomorphic encryption scheme is proposed to be efficient and practical. But it needs users to determine the maximum sum of plaintexts, which should not be bigger than the modulus. Otherwise, the scheme is not homomorphic.

That is, if it is used to encrypt values in a table column, the maximum sum of such values must be predetermined and it cannot be bigger than the modulus.

An order preserving encryption scheme has been proposed in Ref. [3]. In this scheme, the *i*th value in the plaintext domain is mapped to the *i*th value in the ciphertext domain, such that the order between plaintexts is preserved between ciphertexts. The scheme [3] can only deal with plaintexts in a finite domain. That is, the number of values in the plaintext domain must be known before using the scheme. The cryptographic analysis of the order preserving encryption scheme is performed in Ref. [5].

The work [2] shows a way of building order preserving polynomials, which are based on the polynomials proposed by Shamir for secret sharing [17]. In this mechanism, the number and range of plaintexts are needed to determine the range of random coefficients in a polynomial. On the other hand, the evaluation results of order preserving polynomials may reveal the distribution of plaintexts, since similar plaintexts are transformed with similar polynomials.

In Ref. [9], an indexing mechanism for range queries is proposed. This mechanism is not strictly order preserving since two different values may be mapped into the same bucket, which is used when checking query conditions. The mechanism can lead to inaccuracy of query results and hence some post-processing is needed to remove unexpected query results.

3 An Architecture of Managing Encrypted Databases

As discussed before, it is desirable that when protecting databases with encryption, the existing DBMSs should be applied without change. For this purpose, we describe an architecture of managing encrypted databases, as shown Fig. 1. In this architecture, enterprise applications are supposed to be built over databases, which are outsourced to a public cloud. Since the public cloud cannot be trusted, the outsourced databases are encrypted for data privacy. Therefore, the database service providers can only access meaningless ciphertexts. Between the encrypted databases and applications is a query proxy, which mediates their communication.

When an application issues a database query, the proxy translates it into a new one that is to be executed over the encrypted database in the cloud. When the query results are returned from the cloud, the query proxy decrypts them and then forwards the decrypted results to the application. For the application, the query result is the same as it directly accesses an unencrypted database. Each query is translated independently by the proxy. Hence, an enterprise can deploy multiple query proxies to process queries in parallel, though there is only one query proxy depicted in Fig. 1.

The query proxy maintains some meta data to perform query translation. The meta data might include databases schemas, encryption keys, and other specific information needed by cryptographic schemes and system management. Encryption keys are surely needed for encrypting or decrypting data. Database schemas are needed when determining the attribute names that are not given explicitly in a query

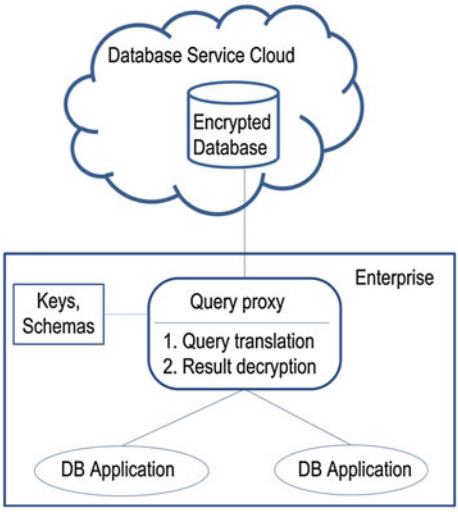


Fig. 1 Architecture of managing encrypted databases

statement, as in the query "*select * from a staff table*". The order-preserving indexing scheme to be introduced later needs the sensitivity of input values to be stored in the query proxy.

The architecture takes a threat model, in which the proxy is deployed into the administrative boundary of the enterprise. Hence, the untrusted database service providers cannot access keys and database schemas maintained in the query proxy. Moreover, the thread model does not allow untrusted service providers to perform plaintext-chosen attacks, since they do not control the query proxy. The prevention of plaintext-chosen attacks at the architecture level is required by order-preserving encryption or indexing schemes [3, 5, 11], since these schemes used for processing range queries leaks order information of plaintexts and hence are vulnerable to plaintext-chosen attacks.

4 Overview: An Approach with Good Usability

We will introduce an approach of managing encrypted databases. This approach comprises an order-preserving indexing scheme, a homomorphic encryption scheme, and how to apply them to encrypt databases and query encrypted databases. The schemes in this approach satisfy all five requirements discussed in the first section, so this approach has good usability in protecting databases in the cloud. In the following, we discuss several types of database queries. The order-preserving indexing scheme and the homomorphic encryption scheme are applied to deal with different query types.

A database query can be an equality query, a range query, an aggregate query or their combinations. An equality query uses the equality comparison as the filtering predicate. For example, the query "*select staffs whose room number is 405*" needs to compare whether the room number of a staff is equal to 405. A range query filters records with inequality comparisons, such as the query "*select staffs who join the company from year 2000 to 2012*". An aggregate query generates a value from a set of data. For example, the query "*select salary average of all staffs*" is an aggregate query. These queries can be combined to perform complex query operations. For example, we can query "*select salary average of staffs* who *join the company from year 2000 to 2012*".

To support the above query types, the approach encrypts a value (e.g., a field in a record) with different cryptographic schemes. For supporting equality queries, a secure hash scheme (e.g., HMAC-SHA1) is used to encrypt the value, such that the ciphertexts of equal values are still equal. To deal with range queries and the aggregate queries using MIN and MAX, the order-preserving indexing scheme is applied. Since this scheme preserves the order of plaintexts, the comparison of two ciphertexts can determine the order of their corresponding plaintexts. For aggregate queries of using SUM and AVG operations, the homomorphic encryption scheme is used to encrypt the value. As a result, the sum or average of ciphertexts can be decrypted to obtain the sum or average of corresponding plaintext values. Note that if a column does not support range queries (e.g., a Boolean column), then the order-preserving indexes of values in this column do not need to be produced; if a column cannot be summed, instead of using the homomorphic encryption scheme, we can use AES to encrypt values in this column.

In addition, database schemas may also contain sensitive information. In the approach, the schemas are anonymised by hashing table names and attribute names when creating an encrypted database. Since a field in a record can be encrypted into multiple ciphertexts, the encrypted records have more fields than the corresponding plaintext records.

5 Order-Preserving Indexing

The order-preserving indexing scheme is used to answer range queries and the aggregate queries using Min and Max over encrypted data. The order-preserving indexing scheme described in this section is proposed in [11].

5.1 Overview

The order-preserving indexing scheme preserves the order between two plaintext values. Formally, the order-preserving index scheme is defined below.

Definition (Order-Preserving Indexing) Suppose k is a secret key, and v_1 and v_2 are two values. If $v_1 < v_2$, then OPI$(k, v_1) < OPI(k, v_2)$, where OPI(k, v_i) means the order-preserving index of value v_i under the secret key k.

That is, by comparing OPI(k, v_1) and OPI(k, v_2), we can know the order of v_1 and v_2. Thus, when order-preserving indexes are stored in encrypted databases, they can be compared by DBMSs when executing queries over encrypted data.

Unlike order-preserving encryption schemes [3, 5], our order-preserving indexing scheme is one-way. That is, from indexes, the original values cannot be recovered even if the key is known. As a result, the order-preserving indexing scheme is simpler to design than order-preserving encryption schemes, since the decryption operation over indexes does not need to be considered.

Our order-preserving indexing scheme is vulnerable to plaintext-chosen attacks, similar to order-preserving encryption schemes. Suppose an adversary can access an oracle to choose arbitrary plaintexts to index. Then, given an index i, the adversary can approximate its plaintext value by the following steps (i.e., binary search).

Step 1: choose an arbitrary value to index
Step 2: compare the index with i
Step 2a: if greater, then choose a smaller value to index
Step 2b: otherwise, choose a bigger value to index;
Step 3: repeat Step 2 until the index is closest to i

To prevent plaintext-chosen attacks, as discussed before, the system architecture is designed to deploy the query proxy in a trusted domain, where the query proxy processes queries issued by database applications.

5.2 An Order-Preserving Indexing Scheme

Before introducing the order-preserving scheme [11], we first define the sensitivity of plaintext values, which indicates the smallest difference between two plaintext values.

Definition (Sensitivity of Plaintexts) Let V be the set of all plaintext values. The sensitivity of V is the minimum element in the set $\{|v_1 - v_2| | v_1 \in V, v_2 \in V \text{ and } v_1 \neq v_2\}$.

By its definition, the sensitivity is always bigger than 0. Though in the above definition, all plaintext values are needed to define their sensitivity. Actually, the sensitivity can be determined by data types or application requirements. For example, if a field contains integers, then the sensitivity is 1; if a field contains even numbers, then the sensitivity can be 2; for a field containing salaries of the form $d_1 d_2 d_3 \cdot d_4 d_5$, where d_i is a digit, the sensitivity can be 0.01.

The order-preserving scheme described in [11] is build over the expression $a^* f(x)^* x + b + noise$, where a and b are real numbers, f is a function that needs

to be instantiated, and *noise* is a random number. For a value v, its index is then computed by $a^* f(v)^* v + b + noise$. Since *noise* is randomly sampled, the indexing scheme is probabilistic. That is, indexing one value twice may generate two different indexes.

To keep the order-preserving property, the following requirements should be satisfied by parameters in the indexing expression $a^* f(x)^* x + b + noise$.

- $a > 0$;
- *noise* is sampled from the range $[0, a^* f(v + sens)^*(v + sens) - a^* f(v)^*(v))$, where $sens$ is the sensitivity of plaintext values;
- $f(x) > 0$ for $x \neq 0$;
- $f(x_1) \geq f(x_2)$ for $x_1 > x_2 \geq 0$ or $x_1 < x_2 \leq 0$.

Note that there is no requirement to plaintext values (i.e., their number, their range and their distribution). The notation $\text{nindex}^{sens}_{[a,b,f]}(v)$ is used to represent the index of value v, where a, b, f and $sens$ are regarded as secret keys of the indexing scheme. The following theorem ensures the order-preserving property of the indexing scheme.

Theorem (Order-Preserving Property) Given the sensitivity *sens* of input values V, for all $v_1 \in V$ and $v_2 \in V$, $if\, v_1 > v_2$, then $\text{nindex}^{sens}_{[a,b,f]}(v_1) > \text{nindex}^{sens}_{[a,b,f]}(v_2)$.

To use the indexing expression $\text{nindex}^{sens}_{[a,b,f]}$, we need to specify the instances of f, which must satisfy the parameter requirements to f. The following are several instances defined and analyzed in [11].

- $f(x) = |x|$;
- $f(x) = x^2$;
- $f(x) = \log_c(d + e^*|x|)$, where $c > 1, d > 1$ and $e > 0$.
- $f(x) = c^* \lfloor |x|/\pi \rfloor + d^* cos(|x| \,\%\, \pi + \pi) + e, where\, d > 0, c \geq 2^* d, e \geq d$, and $\lfloor \_ \rfloor$ and % are the floor and modulo operators, respectively.

These instances of functions $f(x)$ can be composed. For example, by composing the third and fourth ones, we can get $f(x) = \log_c(d + e^* | g^* \lfloor |x|/\pi \rfloor + h^* cos(|x| \% \pi + \pi) + i|)$, where $c > 1, d > 1, e > 0, h > 0, g \geq 2^* h, i \geq h$. Moreover, the composite $f(x)$ still satisfies the parameter requirements of the indexing scheme.

An example of order-preserving indexing is shown in Fig. 2. In the example, the input values are integers from -10 to 10 with the sensitivity 1 and the indexing expression is $16^* \log_7(10 + 18^* |x|)^* \text{x} + 317 + noise$. We can check that the order between input values is preserved among indexes.

The indexing scheme can be applied to index numeric values directly. To index strings, we need to convert strings into the numeric values. A simple idea is that a character in the string is converted into its ASCII encoding. For example, "BC" is converted to 0x4243. However, this simple idea may not work since strings are usually compared in the lexical order. For example, the string "BC" is greater than "ABC". If "BC" is converted to 0x4243 and "ABC" is converted to 0x414243, then 0x4243 is less than 0x414243, which is not correct. To index strings, our indexing scheme needs to know the maximum length of strings that will be compared. If the maximum length of input strings is l and a string has the length n, then $(l - n)$ bytes

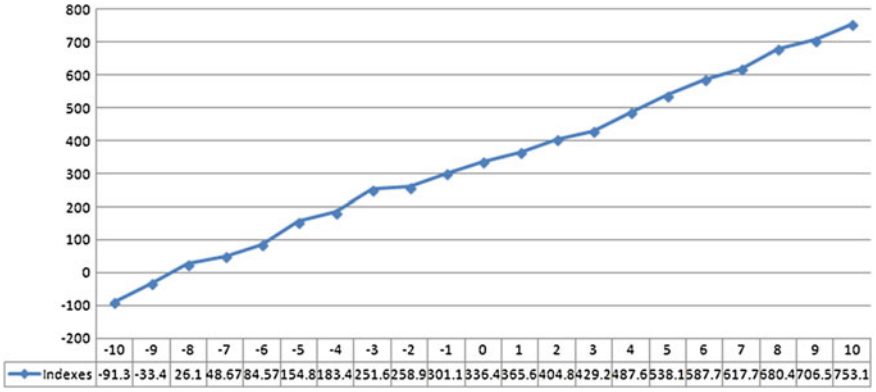


Fig. 2 An Example of order-preserving indexes

of zeros will be padded to the end of the converted integer. For example, if $l = 3$, then "BC" is converted to 0x424300. In addition, the sensitivity of input strings is 1 when they are converted into integers.

5.3 Programmability

The order-preserving indexing scheme allows indexing expressions to be programmed into more complex indexing expressions. In [11], a formal syntax of programming indexing expressions is given. Here, we give intuitive explanation on three programming forms: summation, sequential composition and conditional composition.

- The summation of indexing expressions is represented as $\text{nindex}^{sens}_{[a,b,f]}(v) + \text{nindex}^{sens}_{[a',b',f']}(v)$, which produces the index of v as the sum of $\text{nindex}^{sens}_{[a,b,f]}(v)$ and $\text{nindex}^{sens}_{[a',b',f']}(v)$.
- The sequential composition of indexing expressions is represented as $\text{nindex}^{sens'}_{[a',b',f']}(v); \text{nindex}^{sens}_{[a,b,f]}(v)$, meaning that v is first indexed by $\text{nindex}^{sens}_{[a,b,f]}$, producing an intermediate index, which is then indexed by $\text{nindex}^{sens'}_{[a',b',f']}$, where $sens'$ is the sensitivity of intermediate indexes.
- The conditional indexing expression can be composed in two ways:
  - if v > c then $\text{nindex}^{sens}_{[a,b,f]}(v)$ else $\text{nindex}^{sens}_{[a',b',f']}(v)$, where $\text{nindex}^{sens}_{[a,b,f]}(c) > \text{nindex}^{sens}_{[a',b',f']}(c)$;
  - if v < c then $\text{nindex}^{s}_{[a,b,f]}(v)$ else $\text{nindex}^{sens}_{[a',b',f']}(v)$, where $\text{nindex}^{sens}_{[a,b,f]}(c) < \text{nindex}^{sens}_{[a',b',f']}(c)$.

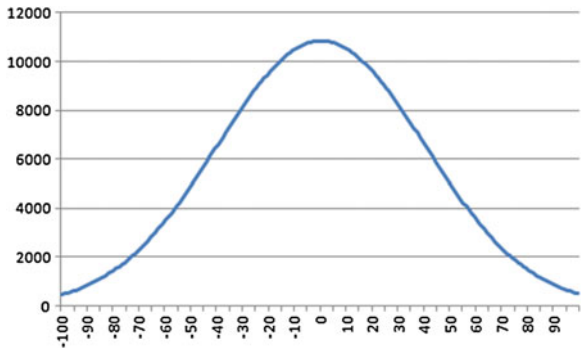


Fig. 3 Plaintexts in Gaussian distribution

The first way means that if v is greater than a constant c, then its index is generated by using the expression $\text{nindex}^{sens}_{[a,b,f]}(v)$, and otherwise, generated by using the expression $\text{nindex}^{sens}_{[a',b',f']}(v)$. The condition $\text{nindex}^{sens}_{[a,b,f]}(c) > \text{nindex}^{sens}_{[a',b',f']}(c)$ ensures that $\text{nindex}^{sens}_{[a,b,f]}(v_1)$ always generates indexes greater than $\text{nindex}^{sens}_{[a',b',f']}(v_2)$ when $v_1 > v_2$, so that the composite indexing expression still satisfies the order-preserving property. Similarly, the second way means that $\text{nindex}^{sens}_{[a,b,f]}(v)$ is used to index v if it is less than c; otherwise, $\text{nindex}^{sens}_{[a',b',f']}(v)$ is used.

Note that these three forms can be mixed in a composite indexing expression. For example, the true branch of a condition indexing expression can be a summation expression, while the false branch can be a sequential expression.

The composite indexing expression contains more secret parameters than its components. Hence, the programmability of indexing expression increases the robustness of the indexing scheme since the forms of indexing expressions are no longer fixed and include more secrets. On the other hand, the programmability of the indexing scheme gives users the capability to unlink the distributions of plaintext values and indexes by indexing plaintexts in different ranges with different expressions. As discussed in [3], it is not secure if the distribution of plaintexts is revealed by the distribution of ciphertexts. In the following, an example borrowed from [11] is used to illustrate how programmability is used to hide the distribution of plaintext values.

Suppose the plaintext values is selected from the range $[-100, 100]$ and their sensitivity is 1. An input value may have 10,000 duplicates. Figure 3 shows the input values in the Gaussian distribution.

Then, the following indexing program is applied to index the plaintext values. By using the conditional composition, the plaintext values are divided into 9 ranges, and processed with different expressions. Figure 4 shows the distribution of the indexes, which is different from the Gaussian distribution of plaintext values.

```
if x > 70 then
   7*(log7(621+12*|x|))*x+1*(*(log12(2030+3*|x|))*x+15683
else if x > 40 then
   17*(log7(1265+8*|x|))*x+7*(log12(621+12*|x|))*x+11706
else if x > 20 then
   25*(log7(6812+78*|x|))*x+17*(log12(1265+8*|x|))*x+8324
else if x => 0 then
   30*(log7(9168+38*|x|))*x+25*(log12(6812+78*|x|))*x+6983
else if x > −20 then
   25*(log7(7523+73*|x|))*x+30*(log12(9168+38*|x|))*x+6983
else if x > −40 then
   20*(log7(8211+31*|x|))*x+25*(log12(7523+73*|x|))*x−6121
else if x > −60 then
   12*(log7(4366+13*|x|))*x+20*(log12(8211+31*|x|))*x−3676
else if x > −80 then
   5*(log7(6723+7*|x|))*x+12*(log12(4366+13*|x|))*x−93
else
   1*(log7(2030+3*|x|))*x+5*(log12(6723+7*|x|))*x−3492
```

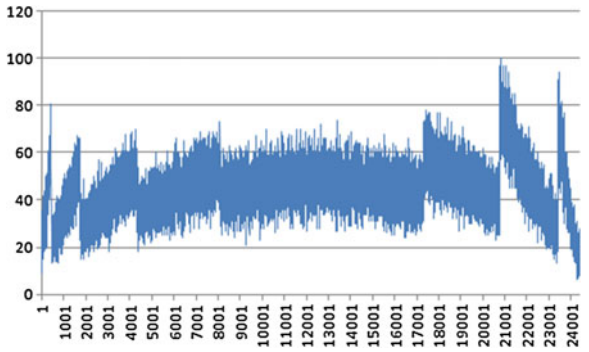


Fig. 4 Distribution of indexes

6 Homomorphic Encryption

Homomorphic encryption allows operations on plaintext values to be performed through operations on ciphertexts. Thus, if table columns in a database are encrypted homomorphically, then the aggregate queries of using SUM and AVG can be directly performed over encrypted table columns by the existing DBMSs.

6.1 Homomorphism

A homomorphic encryption scheme can be fully homomorphic or partially homomorphic. In a fully homomorphic encryption scheme, both additions and multiplications

can be performed over ciphertexts, while in a partially homomorphic encryption scheme, only additions or multiplications can be performed over ciphertexts.

Fully homomorphic encryption is a dream of cryptographic research. Though some fully homomorphic encryption schemes have been proposed [8], there is still no work on the practical applications of these schemes. This is partially caused by the gap between the cryptographic primitives and practical applications. For example, in [18], the plaintext values can only be a bit 0 or 1, while the data in practical applications usually consists of a sequence of bits (e.g., an integer of 32 bits). It is not efficient to encrypt each bit separately. Consequently, the current fully homomorphic encryption schemes are not practical enough to be applied to encrypt databases.

Partially homomorphic encryption can be practical. For example, the widely used RSA is a multiplicatively homomorphic encryption scheme. That is, suppose v_1' and v_2' is the RSA encryption of two plaintexts v_1 and v_2 with the same public key. Then, the decryption of $v_1' * v_2'$ with the corresponding private key is the result of $v_1 * v_2$. However, multiplicatively homomorphic encryption is not useful for performing aggregate queries of SUM and AVG, since these queries need the summation of plaintext values, not their multiplication. Instead, an additively homomorphic encryption scheme is more useful for these aggregate queries.

For an additively homomorphic encryption scheme, it is desirable for queries that the sum of plaintext values can be decrypted from the sum of corresponding ciphertexts. Thus, in order to get the sum of one table column, the values in the encrypted column can be added by the existing DBMSs. Some additively homomorphic encryption schemes do not satisfy this requirement. For example, in the homomorphic encryption scheme [15], the sum of plaintext values is obtained through the multiplication of ciphertexts. This encryption scheme is used by the database encryption systems [7, 16], where the existing DBMSs have to be extended to deal with aggregate queries involving SUM.

The modern encryption algorithms are usually built over finite algebraic structures enforced by using the modulo operation. However, this finiteness requirement is harmful for homomorphic encryption over databases. For example, an additively homomorphic encryption scheme is proposed in [12]; in this scheme, if the sum of plaintexts is greater than the modulus, then scheme is no longer homomorphic. Consequently, it is hard to use this scheme in database encryption, since it is hard to determine the maximum sum of values in a table column for a long-standing database.

6.2 A Homomorphic Encryption Scheme Without Modulus

We have proposed a generic scheme of defining homomorphic encryption without using modulo operations [10]. The scheme supports both additive and multiplicative homomorphism and allows values from an infinite algebraic structure to be encrypted. Here, we introduce one instance of this scheme.

Let *Enc* be the encrypting operation, *Dec* the decrypting operation and $K(n)$ the key. Then, given a value v, the encryption $Enc(K(n), v)$ will generate a ciphertext $(c_1, \ldots, c_n)$, which consists of n subciphertexts $c_1, \ldots,$ and c_n. The parameter n in a key indicates the number of subciphertexts to be generated. In decryption, the operation $Dec(K(n), (c_1, \ldots, c_n))$ will return v. Given another value v', let $Enc(K(n), v') = (c_1', \ldots, c_n')$. Then, the scheme ensures $Dec(K(n), (c_1 + c_1', \ldots, c_n + c_n')) = v + v'$ for additive homomorphism.

The multiplication of v and v' can be decrypted in two steps from the outer product of $(c_1, \ldots, c_n)$ and $(c_1', \ldots, c_n')$, which is represented as $(c_1 * c_1', \ldots, c_n * c_1', \ldots, c_1 * c_n', \ldots, c_n * c_n')$. At the first step, we perform the decryption $Dec(K(n), (c_1 * c_i', \ldots, c_n * c_i'))$ for $i \leq 1 \leq n$ to produce the intermediate ciphertext $(v * c_1', \ldots, v * c_n')$. At the second step, we get $v * v'$ from $Dec(K(n), (v * c_1', \ldots, v * c_n'))$. Specially, given a real number h, we have $Dec(K(n), (h * c_1, \ldots, h * c_n)) = h * v$.

In this instance, the key $K(n)$ is a list of n tuples of real numbers, $[(k_1, s_1, t_1), \ldots, (k_n, s_n, t_n)]$, where $n \geq 3$, $t_i \neq 0 (1 \leq i \leq n-1)$, $\sum_{i=1}^{n-2} k_i \neq 0$, and $k_n + s_n + t_n \neq 0$. The operation *Enc* encrypts v into $(c_1, \ldots, c_n)$ by the following steps.

- Let $r_1, \ldots, r_{n-1}$ be $n-1$ random numbers;
- $c_i = t_i * k_i * v + s_i * r_{n-1} + t_i * r_i$ for $1 \leq i \leq n-2$;
- $c_{n-1} = k_{n-1} * t_{n-1} * \sum_{i=1}^{n-2} r_i + s_{n-1} * r_{n-1}$;
- $c_n = (k_n + t_n + s_n) * r_{n-1}$.

The operation *Dec* decrypts the ciphertext $(c_1, \ldots, c_n)$ into v by the steps below. If the keys are correct, we can see the random noises in each subciphertexts are counteracted, and hence the correct value v is returned. Unlike the methods in [6, 12], we do not use the modulo operation to remove noises, so the presented encryption scheme can be applied to infinite data ranges.

- $L = \sum_{i=1}^{n-2} k_i$;
- $S = c_n / (k_n + t_n + s_n)$;
- $I = c_{n-1} - S * s_{n-1}$;
- $v = \sum_{i=1}^{n-2} (c_i - S * s_i) / (L * t_i) - I / (L * k_{n-1}^* * t_{n-1})$.

The last step of decryption divides different $c_i - S * s_i$ $(1 \leq i \leq n-2)$ with different secret values $L * t_i$. Thus, if an adversary wants to recover v from ciphertexts in brute-force, then he needs to guess t_i $(1 \leq i \leq n-2)$ for each subciphertexts, in addition to guessing other secrets $s_i (1 \leq i \leq n-1)$, L, $k_{n-1} * t_{n-1}$, and $k_n + t_n + s_n$.

6.3 Composition of Homomorphic Encryption

The generic scheme in [10] allows multiple instances to be defined. These instances can be composed into new instances, which are still homomorphic. Briefly, the composition can be achieved by encrypting each subcipertext from one instance again by using the same or another instance.

Suppose there are two homomorphic encryption instances. The first one has the key $K_1(n)$, the encryption operation Enc_1, and the decryption operation Dec_1, and the second one has the key $K_2(m)$, the encryption operation Enc_2, and the decryption operation Dec_2. For the composition of the first and second instances, a value v will be encrypted into $m*n$ subciphertexts, as shown below.

- $Enc_1(K_1(n), v) = (c_1, \ldots, c_n)$;
- $Enc_2(K_2(m), c_i) = (c_{i1}, \ldots, c_{im})$ for $1 \leq i \leq n$;
- The final ciphertext is $(c_{11}, \ldots, c_{1m,\ldots,}c_{n1}, \ldots, c_{nm})$.

To decrypt the ciphertext $(c_{11}, \ldots, c_{1m,\ldots,}c_{n1}, \ldots, c_{nm})$, the following steps are taken.

- $Dec_2(K_2(m), (c_{i1}, \ldots, c_{im})) = c_i$ for $1 \leq i \leq n$;
- $Dec_1(K_1(n), (c_1, \ldots, c_n)) = v$.

A composed homomorphic encryption scheme is more robust than its component instances. Suppose an adversary wants to recover a plaintext from a ciphertext generated by a composed scheme. Then, in addition to breaking each component instance, he needs to guess how to split subciphertexts, so that each subgroup of subciphertexts can be correctly decrypted by using Dec_2 into correct intermediate subciphertexts, which are then decrypted by Dec_1.

In the architecture of managing encrypted databases, the database service providers cannot know whether a ciphertext is generated by a composed homomorphic encryption scheme, since they cannot access the query proxy. This increases the difficulty for them to perform brute force attacks on stored ciphertexts.

6.4 Examples of Homomorphic Encryption

We use examples to illustrate the homomorphic encryption instance. The following key is supposed to be used.

$$[(6.03, 74.99, 94.17), (-56.60, 13.07, 32.45),$$
$$(76.11, 71.69, 34.48), (29.87, 32.70, 92.80)]$$

This key consists of four tuples, meaning that a value will be encrypted into a ciphertext that has four subciphertexts. A key component can be either a positive real or a negative real. The plaintext values in this example are five reals: 1,384.4, 1,384.4, 345.3, 9,233.9 and 563.21. Using the thomomorphic encryption instance, we get five ciphertexts listed in Table 1, with each having four subciphertexts.

Note that the first two values are encrypted into different ciphertexts, though they are the same. The noises used in the encryption are listed in Table 2, with each row containing the noises for encrypting the corresponding value. They can be used to verify the correctness of the encryption operation.

Table 1 Example of ciphertexts

(858839.59014,	−2536268.23010,	2119402.922028,	14817.6369)
(848724.37914,	−2511656.80840,	3298624.388448,	41558.3676)
(309553.73793,	−600157.07450,	4945540.441476,	49011.4665)
(5331366.24729,	−16941668.30340,	3738747.931116,	8301.4191)
(407243.945071,	−1005986.464300,	2392536.630136,	118441.6584)

Table 2 Example of noises

702.25	102.76	95.37
457.78	791.88	267.48
953.82	922.10	315.45
891.31	531.91	53.43
321.35	569.52	762.32

Applying the decryption operation to the ciphertexts, we can get the correct plaintexts. Moreover, the sum of plaintext values 12,909.21 can be obtained by decrypting the following sum of ciphertexts, which is obtained by adding the corresponding subciphertexts in each ciphertext.

$$(7755727.899571, -23595736.880700, 16494852.313204, 232130.5485)$$

The average of plaintexts 2,581.84 can also be correctly decrypted from the following average of ciphertexts, which is obtained by averaging the corresponding subciphertexts in each ciphertext.

$$(1551145.579914, -4719147.376140, 3298970.4626408, 46426.1097)$$

7 Translation of SQL Queries

A database schema designed by application developers are created differently in an encrypted database. We first describe the table structures in an encrypted database and then introduce how to translate a query from an application into a query that can be executed over the encrypted database.

7.1 Table Structures

A table designed by application developers may include multiple columns. In the presented approach, each column is processed independently. Hence, we take a table

that contains only one column as the example to explain the change of table structures in the encrypted database. Suppose a table *Staff* has been designed for an application with one column *Salary*. When creating such a table in a database, the query proxy hashes the table name, such that the table name is meaningless to the untrusted database service providers.

For the column *Salary*, the proxy actually creates multiple columns in the encrypted table. The number of columns depends on the number of subciphertexts generated by the homomorphic encryption scheme. Assume the homomorphic encryption scheme is configured in the query proxy to generate n subciphertexts for the *Salary* column. Then, there will be $n + 2$ corresponding columns created for the *Salary* column. The names of these $n + 2$ columns are obtained by hashing names *SalaryEqIdx*, *SalaryRngIdx*, $SalaryEnc_1$, …., and $SalaryEnc_n$. In these names, *EqIdx*, *RngIdx* and Enc_i are postfixes appended by the query proxy. Figure 5 shows the *Staff* table structure designed by application developers and the table structure managed by the cloud database service, where the notation $Staff'$ represents the hash of the name *Staff*, and similarly for other hashed names $SalaryEqIdx'$, $SalaryRngIdx'$, $SalaryEnc'_1, \ldots$, and $SalaryEnc'_n$.

Note that the n subciphertexts $SalaryEnc'_1$, …, and $SalaryEnc'_n$ can be stored not necessarily in the order of subciphertexts generated from encryption. For example, we can store the subcipertexts in the order $SalaryEnc'_2$, …, $SalaryEnc'_n$, and finally $SalaryEnc_1\prime$. Moreover, the subcipertexts of one value can be mixed with the subcipertexts of another value in the same record. Thus, the adversary is hard to know whether two subcipehrtexts come from the encryption of one value.

When a salary from the database application is being put into the encrypted table, the proxy produces $n + 2$ values for the corresponding columns $SalaryEqIdx'$, $SalaryRngIdx'$, $SalaryEnc'_1$,…, and $SalaryEnc_n\prime$, by using the hash algorithm like HMAC-SHA1, the order-preserving indexing scheme and the homomorphic encryption scheme. The columns $SalaryEqIdx'$ and $SalaryRngIdx'$ are used to process query conditions involving equality and range comparisons, and when the query conditions are satisfied the values in the n columns $SalaryEnc1'$, …, and $SalaryEncn'$ will be returned to decrypt. Note that if values in a column cannot be added or averaged, we also can use other encryption schemes like AES to encrypt this column.

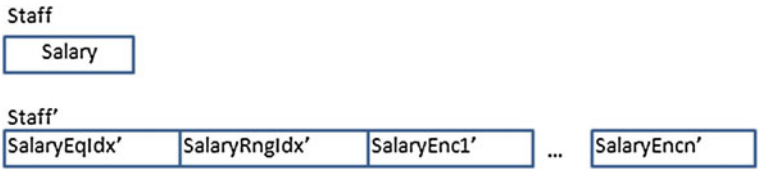


Fig. 5 Change of table structures

7.2 Query Translation

The query translation relies on some meta data. Assume the proxy has the key $K(n)$ for homomorphic encryption, a key k for secure hashing and indexing. An indexing expression is denoted by *Index*(v, s), meaning that v is indexed by *Index* with the sensitivity s. In the following, we assume the sensitivity is *sens*. Note that different columns may use different keys and indexing expressions. The numerical and string data types are represented by Num and String, respectively.

7.2.1 Creation of Databases and Tables

To create a database and a table, the database application can issue the following two statements.

create database *dbname*
create table *tblname* (*colnm Type*,...)

After receiving the above statements, the query proxy translates them into the following ones, which will be executed by the cloud database service. The original schema is recorded by the query proxy in its meta data, where *Hash* represents a secure hash algorithm like HMACSHA1.

create database Hash(*k*,*dbname*)
create table Hash(*k*,*tblname*) (Hash(k, *colnm*+"EqIdx") String, Hash(*k*, *colnm*+"RngIdx") Num,
Hash(k,*colnm*+"Enc1") Num,..., Hash(k,*colnm*+"Enc*n*") Num,…)

The new columns have different data types. The column *colnm*+"EqIdx" have the type String, since its values are always hexadecimal strings generated by the secure hash function. The values of column *colnm*+"RngIdx" are generated by the indexing mechanism and have the numerical type. The columns *colnm*+"Enc1",…, and *colnm*+"Enc*n*" for subciphertexts have the type Num, so that they can be summed or averaged by the DBMSs. Strings can be converted into numeric values before using the homomorphic encryption scheme, or they can be encrypted with other encryption schemes like AES, since it is not meaningful to perform addition operations over strings.

7.2.2 Data Insertion

After a table is created, the database application can put a new record into the table by using the following statement.

insert into *tblname* (*colnm*,...) values (*v*,...)

For this statement, the query proxy translates it into the following one. In the new statement, the value v is hashed, indexed and encrypted before being stored into the encrypted table. The encryption of v using the homomorphic encryption scheme with the key $K(n)$ produces n subciphertexts c_i $(1 \leq i \leq n)$.

insert into Hash(k, tblname)(Hash(k, colnm + "EqIdx"), Hash(k, colnm + "RngIdx"),
Hash(k, colnm + "Enc1"), ..., Hash(k, colnm + "Enc$n$"), ...)
values (Hash(k, v), Index(v, sens), c_1, ..., c_n, ...)

7.2.3 Queries of Data Selection

The data selection queries select one or more columns from a table. The following two forms of query statements can be used to select the column *colnm* or all columns (indicated by *) from the table *tblname* under the condition *cond*.

select *colnm*,... from *tblname* where *cond*
select * from *tblname* where *cond*

The second form can be changed into the first one by replacing * with all column names according to the table schema maintained by the query proxy. For the first form, the query proxy translates it into the following one, where the translation of *cond* into *cond*′ is discussed below.

select Hash(k, *colnm* + "Enc1″), ..., Hash(k, *colnm* + "Enc$n$″), ...
from Hash(k, *tblname*) where *cond*′

In the new query, all subciphertexts must be selected, so that the query proxy can perform the decryption. For the condition *cond*, it is defined over the primitive logical forms *colnm* < *c*, *colnm* = *c*, *colnm* > *c*, where *c* is a constant from the domain of the *colnm* column, by using the logical connectives. When translating the condition *cond*, we replace each primitive logical form with a translated one, as defined below.

The condition *colnm* < *c* is translated into *Hash*(*k*,*colnm*+"RngIdx") < *Index*(*c*,0). Note that *Index*(*c*, 0) is the minimum index of *c*, since no noise is added. The condition *colnm*=*c* is simply translated into *Hash*(*k*,*colnm*+"EqIdx") = *Hash*(*k*,*c*). Assume the sensitivity of values in the *colnm* column is *sens*. Then, *c*+*sens* is the next value of *c*, and *colnm* > *c* is equivalent to the new condition *colnm* ≥ *c* + *sens*, which is translated into *Hash*(*k*,*colnm*+"RngIdx") ≥ *Index*(c+sens,0). Again, *Index*(*c*+*sens*,0) is the minimum index of *c*+*sens*.

In addition, the keywords *order by colnm* and *group by colnm* might be used in queries. They are translated into *order by Hash*(*k*,*colnm*+"*RngIdx*") and *group by Hash*(*k*,*colnm*+"EqIdx"), respectively. That is the ordering comparisons are performed over the columns produced with the order-preserving indexing scheme, and the grouping operation replies on the columns that support equality comparison.

7.2.4 Aggregate Queries: SUM and AVG

The values in a table column can be calculated for their sum and average. The following query statement can be used for this purpose.

select SUM(*colnm*),... from *tblname* where *cond*

select AVG(*colnm*),… from *tblname* where *cond*

In the homomorphic encryption scheme, the sum or average of ciphertexts are performed on each subcipertexts. Hence, the above statements are translated into the following ones.

select SUM(Hash(*k*,*colnm*+"Enc1")),..., SUM(Hash(*k*,*colnm*+"Enc*n*")), …

from Hash(*k*,*tblname*) where *cond′*

select AVG(Hash(*k*,*colnm*+"Enc1")),..., AVG(Hash(*k*,*colnm*+"Enc*n*")), …

from Hash(*k*,*tblname*) where *cond′*

After receiving the sum or average of subciphertexts, the query proxy can decrypt them into the expected sum or average of values in the *colnm* column. The translation of *cond* is the same as that in the data selection queries.

7.2.5 Aggregate Queries: MAX and MIN

The maximum or minimum value in a column might be queried by using the following queries.

select MAX(*colnm*) from *tblname* where *cond*

select MIN(*colnm*) from *tblname* where *cond*

Each of the above queries is translated into two queries.

select MAX(Hash(*k*,*colnm*+"*RngIdx*")) from Hash(*k*,*tblname*) where *cond′*

select Hash(*k*,*colnm*+"Enc1"),..., Hash(*k*,*colnm*+"Enc*n*"), …

from Hash(*k*,*tblname*) where *cond′* and Hash(*k*,*colnm*+"*RngIdx*") = *max*

select MIN (Hash(*k*,*colnm*+"*RngIdx*")) from Hash(*k*,*tblname*) where *cond′*

select Hash(*k*,*colnm*+"Enc1"),..., Hash(*k*,*colnm*+"Enc*n*")), …

from Hash(*k*,*tblname*) where *cond′* and Hash(*k*,*colnm*+"*RngIdx*") = *min*

The first queries determine the maximum index or the minimum index. After getting the maximum index (*max*) or the minimum index (*min*), the query proxy then constructs the second queries to get back the subciphertexts corresponding to *max* or *min*. From the subciphertexts, the maximum or minimum values can be decrypted. Note that we cannot get the maximum or minimum values from the maximum or minimum indexes, since the order-preserving indexing scheme is one-way.

8 Implementation and Evaluation

We implemented a prototype of querying encrypted databases with the SQL Server 2008 as the underlying DBMS. Since the query proxy communicates with the DBMS in standard SQL queries, we can change to a cloud database service just by changing the communication channel (e.g., the IP address of the cloud database science, the port of TCP or UDP, and the secrets for being authenticated).

In this prototype, the database application stores person information in an encrypted database. The person table designed by the application developers has the following schema.

person(id int, name varchar(64), gender varchar(8), birthdate bigint, income numeric(10,2))

A fragment of the encrypted person table is shown in Figure 6 by using the Microsoft SQL Server Management Studio. There are six columns in Figure 6, which are generated from the processing of person incomes, corresponding to the hashes of incomes, their order order-preserving indexes and four subcipehrtexts of encrypting each income. The attribute names are hashed, as shown at the first row in Figure 6. Thus, from this encrypted table, the untrusted database service provider cannot get any meaningful information. For other attributes of the original person table (i.e., id, name, gender and birthdate), they are encrypted with AES, since they cannot be meaningfully added or averaged with the SUM or AVG operations in a query. In addition, the gender attribute can only be "Male" or "Female", and there are no meaningful range queries for this attribute. Hence, the order-preserving indexes are not generated for this attribute.

The order-preserving indexing in the encrypted person table is performed with the following expression. This expression is kept secret in the query proxy. Due to the programmability of the order-preserving indexing scheme, the form of the following indexing expression is not known. It brings difficulty for an adversary to effectively guess the indexing expression even if the adversary happens to know some pairs of plaintexts and indexes.

$$3754.3^{*}\log_{120.2}(513.8 + 77543.32^{*}|(3187.2^{*}\lfloor|x|/\pi\rfloor + 196.2^{*}\cos(|x|\%\pi + \pi) + 26867.3)|)^{*}x + 84648.87)$$

As described above, an income value is encrypted into 4 subciphertexts, so the key should have the form $[(k_1, s_1, t_1), (k_2, s_2, t_2), (k_3, s_3, t_3), (k_4, s_4, t_4)]$. In this evaluation, each k_i or t_i is allowed to have 4 digits, and each s_i to have 8 digits. Thus, the 4 subciphertexts of homomorphic encryption leads to a key space of size 10^{52} (i.e., 10^{52}=10^4*10^8*(10^8*10^4*10^8*10^4*10^8*10^4*10^4), which is the product of the space sizes of L, S, s_1, t_1, s_2, t_2, s_3, t_3, k_3). Other attributes other than income are encrypted with AES 128, which has key space of size 2^{128}.

The performance of querying encrypted databases is tested with respect to data insertion and query on a Dell Latitude E4310 laptop. To test the performance of data insertion, we generate 10,000 person records and insert them into a plain database (PlainDB), where data is not encrypted, and an encrypted database (EncDB), respectively. Figure 7 shows the used time (milliseconds) after inserting every 2,000 records. Compared with the insertion to PlainDB, the insertion of 10,000 records to EndDB takes about more 22.9 % time.

The insertion to EncDB involves four different cryptographic schemes: the hash algorithm (HMACSHA1), the order-preserving indexing scheme, the AES encryption and the homomorphic encryption. Figure 8 shows the time taken by each of these schemes. From this figure, we can see the HMACSHA1 algorithm takes more time than the other three schemes in total. Actually, if we change the order-preserving indexing scheme into a deterministic one by avoiding noises in indexes, then the equality check can be carried out over the indexes, too. At this case, the values in encrypted databases do not need to be hashed, and hence the performance of insertion will be increased.

The query performance is tested on two types of queries. The first type is to select records satisfying some conditions, while the second is an aggregate query using the SUM operation. We use the query below to select records from the encrypted database.

$$\text{select} * \text{from person where income} > \text{min and income} < \text{max}$$

By changing *min* and *max*, we can get five different results, including correspondingly 2,000, 4,000, 6,000, 8,000 and 10,000 records. The time spent on querying PlainDB and EncDB is shown in Fig. 9, from which we can see the performance overhead is linearly increased with the increase of the number of records in the query result. This increase of performance overhead is reasonable, since more records in the encrypted query result need more time to decrypt.

The aggregate query is performed by the following statement, which sums the income of persons satisfying the query conditions.

$$\text{select} * \text{SUM(income) from person where income} > \text{min and income} < \text{max}$$

Cc825d5bbb9aa59f4116303b1cbb4a4fd1c42...	C0ff0c7ac86...	Cc7ec26f81c94e47...	Cd625506ed757c43a...	C2092f10c44be9b4e...	C77df75d77d0f6dbf187
0xB26188E597BA04E117DB4287BF881265...	19772809.11	13049408114.23000	162111443534.48000	79202642383.96000	109932430413.04000
0xB9CE25F77AF8E77EF85E6E338CF4792B...	19781138.13	42946303689.76500	533779391613.01400	259356433621.51000	361969020131.28000
0xEC57E4156A2459DA3C530C1124B95B2E...	19787185.05	29666687091.13000	368732737115.10800	179428528457.21000	250045883035.36000
0xA8DEBC64F8DF4B76CEB747C23B2B51D...	19794019.65	2840338453.90500	35252591912.08200	17422421669.59000	23906376102.24000
0x746FF4D07895818DAFD0F31A00B257D1...	19798704.95	6652840774.45000	82623442536.41600	40665364177.93000	56029104294.80000
0x5A5D689702DF447CF0329628C4EB2C56...	19810175.00	29132989883.55500	362061314136.07000	176131800381.27000	245522905415.76000
0xC9E56B09922F2004B79A71C1F5F89FE45...	19814487.46	37566051837.71000	466873579866.96400	227267403350.93000	316597781046.00000
0x5D66C25692CC79AD0D833FE2265EE705...	19823026.95	21112541829.03500	262402816182.17800	127373261211.88000	177942844899.60000
0xCC78DAF3A2A87A68536C232616098922...	19827396.27	18847491820.73000	234191785910.27200	114158141190.95000	158811693496.64000
0xCC219DC5150E4BBBE2AA1ADA1FF5965...	19839157.75	35856416800.84500	445634418153.96600	216568728060.45000	302196262387.04000

Fig. 6 A fragment of encrypted person table

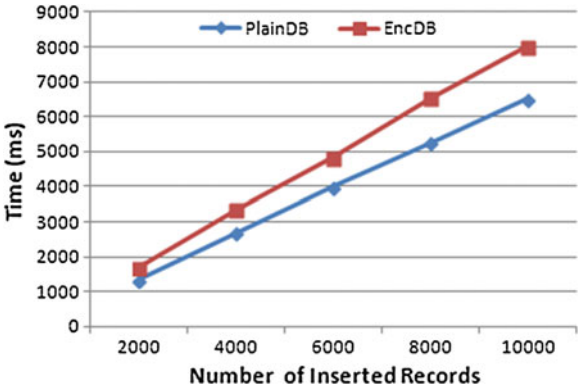
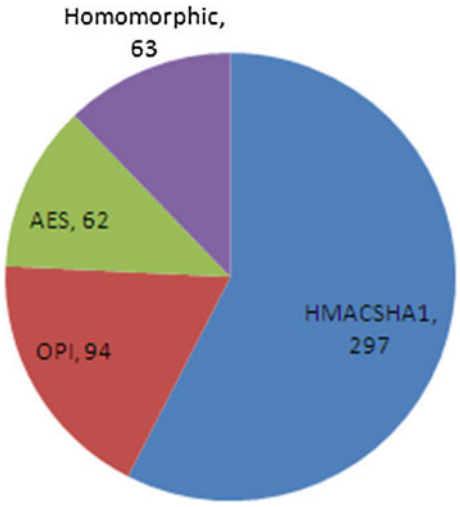


Fig. 7 Performance of insertion

Fig. 8 Performance of different cryptographic schemes



We still let the query return five different results, corresponding to the income sums of 2,000, 4,000, 6,000, 8,000 and 10,000 records. Figure 10 shows the performance result. This result shows that the query time does not increase quickly as that in the selection queries with the increase of records included in the query results. This is because the result of the above aggregate query has only one value (the sum of encrypted income in each person record satisfying the condition) to decrypt, regardless of the number of aggregated records.

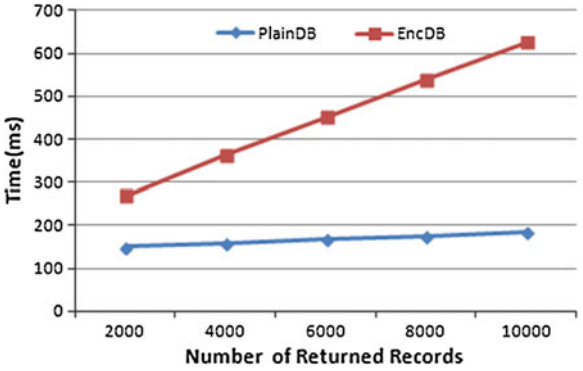


Fig. 9 Query performance for different records number

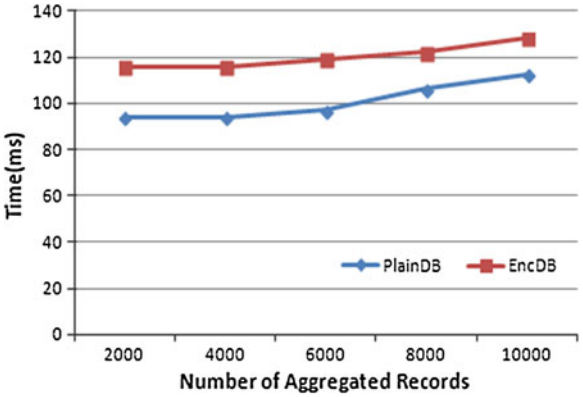


Fig. 10 Performance of aggregate queries

9 Summary

In this chapter, we introduce an approach for database encryption and query. In particular, we introduce an order-preserving indexing scheme and a homomorphic encryption scheme. Compared with the existing order-preserving and homomorphic encryption schemes, the presented schemes are more suitable for long-standing database, since they do not need users to predetermine the number and range of data stored in databases and their maximum sums. We implement a prototype that uses the exiting DBMS (i.e., Microsoft SQL Server 2008) and evaluate its performance. The evaluation shows that the approach incurs acceptable performance overhead.
The approach cannot deal with all SQL queries. For example, it cannot support queries that use conditions involving operations of several columns (e.g., $number*rate>10$). It is an interesting problem of improving the system to make it support more types of SQL queries in future.

References

1. Amazon Relational Database Service. http://aws.amazon.com/rds/
2. Agrawal D, Abbadi A, Emekçi F, Metwally D (2009) Database management as a service: challenges and opportunities. In: Proceedings of the 25th international conference on data, engineering, pp 1709–1716
3. Agrawal R, Kiernan J, Srikant R, Xu Y (2004) Order preserving encryption for numeric data. In: Proceedings of the 2004 ACM SIGMOD international conference on management of data, SIGMOD'04, pp 563–574
4. Bertino E, Sandhu RS (2005) Database security-concepts, approaches, and challenges. IEEE Trans Dependable Secure Comput 2(1):2–19
5. Boldyreva A, Chenette N, Lee Y, O'Neill A (2009) Order-preserving symmetric encryption. EUROCRYPT, pp 224–241
6. Brakerski Z, Vaikuntanathan V (2011) Fully homomorphic encryption from ring-LWE and security for Key dependent messages. CRYPTO 2011, pp 505–524
7. Ge T, Zdonik S (2007) Answering aggregation queries in a secure system model. In the 33rd international conference on very large data bases, pp 519–530
8. Gentry C (2009) Fully homomorphic encryption using ideal lattices. STOC 2009, pp 169–178
9. Hore B, Mehrotra S, Tsudik G (2004) A privacy-preserving index for range queries. VLDB 2004, pp 720–731
10. Liu D (2012) Homomorphic encryption for database querying. Australian Provisional Patent 2012902653 (filed by CSIRO)
11. Liu D, Wang S (2013) Nonlinear order preserving index for ncrypted database query in service cloud environments. Concurrency Comput: Pract Experience (in press)
12. Naehrig M, Lauter K, Vaikuntanathan V (2011) Can homomorphic encryption be practical? In: Proceedings of the 3rd ACM workshop on cloud computing security workshop, CCSW'11, pp 113–124
13. Microsoft SQL Azure Database. http://www.windowsazure.com/en-us/home/features/data-management/
14. Oracle Database Cloud Service. https://cloud.oracle.com
15. Paillier P (1999) Public-key cryptosystems based on composite degree residuosity classes. EUROCRYPT 1999, pp 223–238
16. Popa RA, Redfield CMS, Zeldovich N, Balakrishnan H (2011) CryptDB: protecting confidentiality with encrypted query processing. In: Proceedings of the 23rd ACM symposium on operating systems principles, SOSP'11, pp 85–100
17. Shamir A (1979) How to share a secret. Commun ACM 22(11):612–613
18. van Dijk M, Gentry C, Halevi S, Vaikuntanathan V (2010) Fully homomorphic encryption over the integers. EUROCRYPT 2010, pp 24–43