International Journal of Security and Its Applications
Vol. 9, No. 5 (2015), pp. 195-204
http://dx.doi.org/10.14257/ijsia.2015.9.5.20

Efficient Database Encryption Scheme for Database-as-a-Service Environment

Hankyu Joo

Hallym University
hkjoo@hallym.ac.kr

Abstract

Computing-as-a-service is gaining ground. Clients may use the service without purchasing the system supporting the service. Database-as-a-Service (DBaaS) is an important area of computing-as-a-service. DBaaS allows clients to use an expensive database management system without purchasing it. In a DBaaS environment, database Tables are stored on servers belonging to a service provider, and hence, they must be encrypted in order to ensure data confidentiality. However, the encryption introduces performance degradation in the execution of queries over encrypted data. The execution of range queries, in particular, undergoes severe performance degradation. Several encryption schemes to alleviate this problem have been proposed. However, most of these schemes leak other information in addition to order information. In this study, a new database encryption scheme for DBaaS is proposed. The proposed scheme enables the execution of range queries without severe performance degradation and without leakage of information other than order information.

Keywords: *database-as-a-service, confidentiality, database encryption, order preserving encryption*

1. Introduction

Nowadays, computing as a service is gaining ground. Clients may use the service without purchasing the system supporting the service. One of the areas in computing as a service is Database-as-a-service (DBaaS). With DBaaS, clients may use an expensive database management system without purchasing the system [1].

In DBaaS, clients store their data Tables on servers that belong to the service providers. These servers are not under the control of clients, and hence, there is no guarantee of confidentiality of the client data that is stored on the servers without a proper protection scheme. Clients may encrypt the sensitive data to ensure data confidentiality. However, the encryption of data introduces performance degradation in the execution of queries over encrypted data.

When a column of a database Table is encrypted, exact queries can be executed without severe performance degradation by maintaining the index on the encrypted column. The client may encrypt the search condition and send a query of the encrypted condition to the server.

However, the execution of range queries undergoes severe performance degradation. In order to execute a range query, the server must decrypt each encrypted item to determine whether the item is in the range. The index is maintained on the encrypted data, and hence, it may not be used for performance enhancement.

Several approaches to alleviate the performance degradation have been proposed. In order to alleviate the problem, the server must be able to perform the order comparison without decrypting the encrypted items. In order to enable the server to perform order comparison without decryption, Order-Preserving Encryption (OPE) schemes have been proposed. Some of the proposed OPE schemes allow servers to perform an order

ISSN: 1738-9984 IJSIA

comparison without decryption; however, they leak other information in addition to order information. Other schemes require the server to perform operations that are not database operations. The server must support such operations.

In this study, a database encryption scheme that does not result in severe performance degradation is proposed. The proposed scheme leaks only order information and uses only database operations of the server.

The remainder of this paper is organized as follows: Related work is discussed in Section 2. Our proposed database encryption scheme is described in Section 3. The proposed scheme is evaluated in Section 4. The conclusion of the study is presented in Section 5.

2. Related Work

A large amount of research has been conducted in the field of database encryption. Database encryption has encountered severe performance degradation in the execution of range queries. Research in database encryption has mainly focused on alleviating performance degradation. OPE was proposed for database encryption by Agrawal, *et al.*, [2] to alleviate performance degradation. Boldyreva, *et al.*, [3] proposed a new security definition for order-preserving encryption called *indistinguishability under ordered chosen-plaintext attack* (IND-OCPA). The introduction of OPE by [2] was followed by numerous studies in the field [3-6]. OPE allows the encryption function to preserve the numerical ordering of the plaintexts. Another similar scheme called order-preserving encoding has also been proposed [7]. An order-preserving encoding scheme maintains tree-structured order information on the server.

Studies such as [7-9] indicate that most of the proposed OPE schemes leak other information in addition to order information. Another scheme [7] requires operations that are not database operations to be performed at the server. Joo [10] proposed a basic approach to the database encryption scheme in which an order column is added for each encrypted column. According to Kolesnikov and Shikfa [11], even an ideal encryption system with order information leaks other information.

3. Proposed Scheme

In this section, the proposed database encryption scheme is described. In Section 3.1, the assumptions on which the design of the scheme is based are listed. The encryption scheme is described in Section 3.2. The algorithm for range query is presented in Section 3.3 and the algorithm for insertion is discussed in Section 3.4.

3.1. Assumptions

The design of a new scheme is based on the following assumptions:

- In DBaaS, all database Tables, including those containing confidential information, are stored on the server of the service provider. The server is a database server, and hence, it can be accessed only by database operation interfaces.
- The cost of storage is not high. This implies that an increase in the Table size does not result in excessive cost.
- In many applications, database Tables are initialized with a large number of items. After the initialization, query operations are extremely frequent. The insertion of items is required; however, it is not a frequent operation. Extensive Table modification may occur periodically.
- Indexing is used to improve the query speed.

3.2. Database Encryption Scheme

The proposed scheme uses an extended Table. When a Table is created and stored on a server, each column having confidential data (x) of a database Table is encrypted and renamed (eX), and an additional order column (oX) is added. The encryption may be performed by using any existing secure encryption algorithm such as Advanced Encryption Standard (AES) [12]. The order column (oX) is used to maintain the order of the confidential data column (x). Let us assume that an item m has a value x_m in column x and oX_m in column oX. Further, an item n has a value x_n in column x and oX_n in column oX. Then, if $x_m < x_n$, $oX_m < oX_n$.

For example, let us consider an *emp* Table, Table 1. The Table contains four columns: *ID*, *Name*, *Title*, and *Salary*. The columns, *ID* and *Salary*, contain confidential information.

Table 1. Plaintext emp Table

ID	Name	Title	Salary
650213-1234567	Hankyu Joo	Manager	14,000
690313-2345678	Sangmin Han	Programmer	12,000
700225-1234567	Jaewook Choi	Programmer	15,000
….	….	….	….

In order to ensure security, the data in the columns, *ID* and *Salary*, are encrypted. The corresponding column names are renamed as *eID* and *eSalary*, as shown in Table 2. The columns, *oID* and *oSalary*, with order information of *ID* and *Salary*, respectively, are also added. The extended Table is named *eEmp* and is stored on the server.

Table 2. Encrypted eEmp Table

eID	Name	Title	eSalary	oID	oSalary
eX12Klm	Hankyu Joo	Manager	tyK3xDs	200	600
jMcD38h	Sangmin Han	Programmer	wQ1b0ld	700	300
y08b1xK	Jaewook Choi	Programmer	d2IpRz9	900	900
….	….	….	….	….	….

Indexes are created for the columns, *eID*, *eSalary*, *oID*, and *oSalary*.

The encryption and order calculation is performed on the computer of the client. When a Table is created, the order column has a large interval. Although x_n and x_m are consecutive in order, oX_n - oX_m is not one but a large number such as 100. Indexes are used whenever necessary.

3.3. Range Query for Encrypted Tables

For unencrypted database Tables, the following range query may be performed. The column x may be indexed to improve the query speed.

```
Select a
from tb
where x >= m and x <= n
```

If the column x contains confidential information, this column is encrypted as the column eX and the encrypted Table is renamed as *eTb*. In the encrypted Table, the same format of range query results in severe performance degradation. In order to alleviate the performance degradation, the query should be changed as follows:

```
Select a
from eTb
where oX >= minLarge(eTb, eX, oX, m) and
      oX <= maxSmall(eTb, eX, oX, n)
```

For example, the following query may be executed to obtain the names corresponding to a salary between *12,000* and *15,000* in Table 1. The column *Salary* may be indexed to improve the query speed.

```
Select Name
from employee
where Salary >= 12,000 and Salary <= 15,000
```

If the column *Salary* is encrypted as the column *eSalary*, and if the column *oSalary* maintains the order of *Salary* as seen in Table 2, the query should be modified as follows.

```
Select Name
from eEmp
where oSalary >= minLarge(eEmp, eSalary, oSalary, 12,000)
          and
      oSalary <= maxSmall(eEmp, eSalary, oSalary, 15,000)
```

The functions *minLarge* and *maxSmall* are executed on the computer of the client. The client communicates with the server to obtain the necessary information by using only the query *select*. The function *minLarge(eTable, field, order, value)* is used to obtain the *order* of an item whose decrypted *field* has the least value greater than or equal to *value* in *eTable*. The algorithm for function *minLarge(eTable, field, order, value)* is described in Table 3. The function *maxSmall(eTable, field, order, value)* is used to obtain the *order* of an item whose decrypted *field* has the largest value smaller than or equal to *value* in *eTable*. The algorithm for function *maxSmall(eTable, field, order, value)* is described in Table 4.

Table 3. MinLarge Algorithm

```
int minLarge(TableName eTable, ColumnName field,
ColumnName order, Comparable value) {
  int min = select MIN(order) from eTable;
  int max = select MAX(order) from eTable;
  return internalMinLarge(eTable, field, order, min, max,
    value);
}
int internalMinLarge(TableName eTable, ColumnName field,
ColumnName order, int min, int max, Comparable value) {
  Comparable dF;
  int eF, oX;
  if (min >= max) return min;
  int mid = (min+max)/2;
  (eF, oX) = select field, order
                 from eTable
                 where order >= mid
                 order by order
                 limit 1;
  dF = decrypt(eF);
  if (value == dF) return oX;
  else if (value < dF) return internalMinLarge(eTable,
     field, order, min, mid-1, value);
  else return internalMinLarge(eTable, field, order,
     oX+1, max, value);
}
```


The function *minLarge* obtains the minimum (*min*) and maximum (*max*) values in the column *order* of *eTable*. With these values, *minLarge* calls the function *internalMinLarge*. The function *internalMinLarge(eTable, field, order, min, max, value)* is a recursive function to determine the *order* of an item whose decrypted *field* has the least value greater than or equal to *value*. The *internalMinLarge(eTable, field, order, min, max, value)* searches only those items whose *order* is between *min* and *max*. When *internalMinLarge* is called, an item whose column *order* has the least value greater than or equal to *mid* is selected from *eTable*. The selected item is checked to determine whether its decrypted *field* is the same as *value*. If the decrypted *field* of the item is the same as *value*, the *order* of the item is found and returned. If the decrypted *field* is smaller than *value*, *internalMinLarge* is called with the range between *min* and *mid*. If the decrypted *field* is larger than *value*, *internalMinLarge* is called with the range between *mid* and *max*. After successive recursive calls, *min* becomes greater than or equal to *max*. At that point, *min* has the *order* of the item whose *field* is least greater than *value*.

Table 4. MaxSmall Algorithm

```
int maxSmall(TableName eTable, ColumnName field,
ColumnName order, Comparable value) {
  int min = select MIN(order) from eTable;
  int max = select MAX(order) from eTable;
  return internalMaxSmall(eTable, field, order, min,
max,
    value);
}
int internalMaxSmall(TableName eTable, ColumnName field,
ColumnName order, int min, int max, Comparable value) {
  Comparable dF;
  int eF, oX;
  if (min >= max) return max;
  int mid = (min+max)/2;
  (eF, oX) = select field, order
                from eTable
                where order <= mid
                order by order DESC
                limit 1;
  dF = decrypt(eF);
  if (value == dF) return oX;
  else if (value < dF) return internalMaxSmall(eTable,
     field, order, min, oX-1, value);
  else return internalMaxSmall(eTable, field, order,
     mid+1, max, value);
}
```

The function *maxSmall* works in a similar manner as the function *minLarge*. The function *maxSmall* obtains the minimum (*min*) and maximum (*max*) values in the column *order* of *eTable*. With these values, *maxSmall* calls the function *internalMaxSmall*. The function *internalMaxSmall(eTable, field, order, min, max, value)* is a recursive function to determine the *order* of an item whose decrypted *field* has the largest value smaller than or equal to *value*. The *internalMaxSmall(eTable, field, order, min, max, value)* searches only those items whose *order* is between *min* and *max*. When *internalMaxSmall* is called, an item whose column *order* has the largest value smaller than or equal to *mid* is selected

from *eTable*. The selected item is checked to determine whether its decrypted *field* is the same as *value*. If the decrypted *field* of the item is the same as *value*, the *order* of the item is found and returned. If the decrypted *field* is smaller than *value*, *internalMaxSmall* is called with the range between *min* and *mid*. If the decrypted *field* is larger than *value*, *internalMaxSmall* is called with the range between *mid* and *max*. After successive recursive calls, *min* becomes greater than or equal to *max*. At that point, *max* is the *order* of the item whose *field* has the largest value smaller than *value*.

3.4. Insertion into Encrypted Tables

When a new item is inserted into an unencrypted database Table, the following insert statement must be executed, where *Tb* is the Table name, *a* and *x* are column names, and *aV* and *xV* are values corresponding to the columns, *a* and *x*:

```
Insert Into Tb (a, x)
Values (aV, xV);
```

When a new item is inserted into an encrypted Table, a value for the column *order* must be calculated. The value of the column *order* for the new item is the average value of the two neighboring order values. Let us assume that the encrypted Table is *eTb*, *eX* is the encrypted column corresponding to *x*, and *oX* is the column *order* corresponding to *x*. The following insert statement must be executed:

```
Insert Into eTb (a, eX, oX)
Values (aV, encrypt(xV), newOx(eTb, eX, oX, xV));
```

The algorithm *newOx* in Table 5 is used to calculate a new value of *order* corresponding to the *value* that is supposed to be encrypted. The function *newOx* is executed on the computer of the client. The client communicates with the server to obtain the necessary information by using only the query *select*.

Table 5. NewOx Algorithm

```
int newOx(TableName eTable, ColumnName field,
ColumnName order, Comparable value) {
  int lox, sox;
  lox = minLarge(eTable, field, order, value);
  sox = maxSmall(eTable, field, order, value);
  if (lox == sox) newOx = lox;
  else if (lox == sox+1) {
     adjust(lox);
     newOx = (lox + sox) / 2;
  } else {
     newOx = (lox + sox) / 2;
  }
  return newOx;
}
void adjust(TableName eTable, ColumnName order, int
lox) {
  tox = an item whose next order is not current
order+1;
  for (all items with order >= lox && order <= tox)
      increase order by 1;
}
```

The function *newOx(eTable, field, order, value)* is used to calculate the new value of *order* corresponding to the *value* that is supposed to be encrypted. The function *newOx*

uses *minLarge* and *maxSmall* functions that are described in Tables 3 and 4, respectively. The *order* of the item, which has the smallest value of *order* that is larger than or equal to the new item with *value*, is selected from *eTable* and saved as *lox*. The *order* of the item, which has the largest value of *order* that is smaller than or equal to the new item with *value*, is selected from *eTable* and saved as *sox*. The new order value (*newOx*) is the average value of *lox* and *sox*. The function *adjust* is called when it is not possible to have a new *order* because the existing order interval is 1.

If multiple *order* columns are added, the number of times that the function *newOx* is called is equal to the number of columns added.

For example, let us suppose that an item *(690812-2011369, Kyoung Kim, Secretary, 12,500)* must be added to Table 1. In order to insert the item, the following insert statement must be executed.

```
Insert Into emp (ID, Name, Title, Salary)
Values (690812-2011369, Kyoung Kim, Secretary, 12,500);
```

After the insertion, the Table *emp* is modified as shown in Table 6.

Table 6. Modified emp Table

ID	Name	Title	Salary
650213-1234567	Hankyu Joo	Manager	14,000
690313-2345678	Sangmin Han	Programmer	12,000
690812-2011369	Kyoung Kim	Secretary	12,500
700225-1234567	Jaewook Choi	Programmer	15,000
….	….	….	….

In order to insert the item into an encrypted Table as shown in Table 2, the following insert statement must be executed.

```
Insert Into eEmp (eID, Name, Title, eSalary, oId, oSalary)
Values (encrypt(690812-2011369), Kyoung Kim, Secretary,
encrypt(12,500), newOx(eEmp, eID, oID, 690812-2011369),
newOx(eEmp, eSalary, oSalary, 12,500));
```

If the above insert statement is executed, an item *(Dj8L1s, Kyoung Kim, Secretary, q9PlnT, 800, 450)* will be added to Table 2, thus generating Table 7.

Table 7. Modified eEmp Table

eID	Name	Title	eSalary	oID	oSalary
eX12Klm	Hankyu Joo	Manager	tyK3xDs	200	600
jMcD38h	Sangmin Han	Programmer	wQ1b0ld	700	300
Dj8L1s	Kyoung Kim	Secretary	q9PlnT	800	450
y08b1xK	Jaewook Choi	Programmer	d2IpRz9	900	900
….	….	….	….	….	….

Periodically, a large number of items may be modified. When this occurs, the Table may be retrieved to the client side, and the *order* columns may be recalculated by using the same approach as used in Table initialization.

4. Evaluation

The encryption of a column uses an existing secure encryption algorithm, and hence, the proposed scheme leaks only order information. This scheme uses only standard query operations to access the server and does not require any changes at the server. In the absence of order information, the performance of a range query for an encrypted column is O(n) slower than that for a plaintext column, where n is the total number of items in the Table. In the proposed scheme, the addition of the order column to the encrypted columns results in the performance of the range query for the encrypted column to be O(log_n) slower than that for a plaintext column. In the proposed method, the insertion operation may be performed in O(log_n). If many items are inserted for a specific range of values, the order column may be restructured. By inspecting the order column, clients may learn about its status.

5. Conclusion

In this study, a new scheme for database encryption is proposed. The proposed scheme may be implemented only on the client side without any changes to the server. It does not leak any information except the order of the encrypted field. The proposed method may execute a range query on the encrypted column without resulting in severe performance degradation. The proposed scheme requires additional columns in the encrypted Table. However, the additional storage for these columns is not expensive in a DBaaS environment. In this method, periodic inspection of the order column is necessary and possible restructuring of the column may be required. However, database insertion is not a frequent operation and restructuring may only be necessary when the original Table undergoes a major update.

ACKNOWLEDGEMENTS

This research was supported by Hallym University Research Fund, 201504 (HRF-201504-012).

References

[1] H. Hacigumus, B. Iyer and S. Mehrotra, "Providing database as a service," Proceedings of 18th International Conference on Data Engineering, **(2002)** February 26–March 1, San Jose, California.
[2] R. Agrawal, J. Kiernan, R. Srikant and Y. Xu, "Order preserving encryption for numeric data," ACM SIGMOD, **(2004)**, pp. 563-574.
[3] A. Boldyreva, N. Chenette, Y. Lee and A. O'Neill, "Order-preserving symmetric encryption," EUROCRYPT, LNCS 5479. **(2009)**, pp. 224-241.
[4] A. Boldyreva, N. Chenette and A. O'Neill, "Order-preserving symmetric encryption revisited: Improved security analysis and alternative solutions," CRYPTO 2011, LNCS 6841, **(2011)**, pp. 578-595.
[5] D. H. Yum, D. S. Kim, J. S. Kim, P. J. Lee and S. J. Hong, "Order-preserving encryption for non-uniformly distributed plaintexts," WISA 2011, LNCS 7115, **(2012)**, pp. 84-97.
[6] D. Liu and S. Wang, "Nonlinear order preserving index for encrypted database query in service cloud environments," Concurrency and Computation: Practice and Experience, vol. 25, **(2013)**, pp. 1967-1984.
[7] R. A. Popa, F. H. Li and N. Zeldovich, "An ideal-security protocol for order-preserving encoding," IEEE Symposium on Security and Privacy, **(2013)**, pp. 463-477.
[8] X. Liangliang, O. Bastani and I. Yen, "Security Analysis for an Order-Preserving Schemes," Technical Report UTDCS-06-10, University of Texas at Dallas, **(2010)**.
[9] X. Liangliang, I. Yen and D. Lin, "Security Analysis for an Order-Preserving Schemes," Technical Report UTDCS-01-12, University of Texas at Dallas, **(2012)**.
[10] H. Joo, "Practical Database Encryption Scheme for Database-as-a-Service," Advanced Science and Technology Letters, Security, Reliability and Safety, vol. 93, **(2015)**, pp. 34-39.
[11] V. Kolesnikov and A. Shikfa, "On The Limits of Privacy Provided by Order-Preserving Encryption," Bell Labs Technical Journal, vol. 17, no. 3, **(2012)**, pp. 135-146.
[12] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)," FIPS PUB, vol. 197, **(2001)**.

Author

Hankyu Joo, received his B.Sc. degree in Computer Science from Hallym University, Korea, in 1988. He received his M.S. and Ph.D. degrees in Computer Science and Engineering from Arizona State University in 1994 and 1998, respectively. He worked for the Electronics and Telecommunications Research Institute, Korea, from 1999 to 2000. He is a professor of Computer Engineering at Hallym University. His research interests include information security and software engineering.