A User Privacy Protection Technique for Executing SQL over Encrypted Data in Database Outsourcing Service

T.B.P. Hue[1], D.N. Thuc[1], T.B.D. Thuy[1], Isao Echizen[2], and S. Wohlgemuth[2]

[1] University of Science, VNU - HCMC
227 Nguyen Van Cu, District 5, Ho Chi Minh City, Vietnam
{ptbhue,ndthuc,dtbthuy}@fit.hcmus.edu.vn
[2] National Institute of Informatics
2-1-1 Hitotsubashi, Chiyoda-ku, Tokyo, Japan
iechizen@nii.ac.jp, sven.wohlgemuth@me.com

Abstract. The fact that the data owners outsource their data to external service providers introduces many security and privacy issues. Among them, the most significant research questions relate to data confidentiality and user privacy. Encryption was regarded as a solution for data confidentiality. The privacy of a user is characterized by the query he poses to the server and its result. We explore the techniques to execute the SQL query over the encrypted data without revealing to the server any information about the query such as the query type or the query pattern, and its result. By implementing all the relational operators by using the unique selection operator on the server-side database with a constant number of elements in each time of selection, our proposal can defeat against the statistical attacks of the untrusted server compromising data confidentiality and user privacy. Experimental evaluation demonstrates that our proposal less affects the system's performance and is applicable in the real world.

Keywords: Database outsourcing, database encryption, user privacy, access pattern privacy, access privacy.

1 Introduction

Amount of data held by organizations is increasing quickly and it often contains sensitive information. Management and protection of such data are expensive. An emerging solution to this problem introduces a new paradigm called *database as a service* (DAS), in which the database of an organization is stored at an external service provider. The advantages of DAS are cost savings and service benefits. There are three main entities in the DAS scenario (Fig. 1): (1) *Data owner:* individual or organization that is the subject of the data made available for controlled external use (2) *User*: individual or organization that requests data from the server (3) *Server*: organization that receives the data sent from the data owners and makes it available for distribution to users.

In DAS scenario, however, sensitive data, which is now stored on a site that is not under the direct control of the data owner, can be put at risk. Moreover, the data request of user can be revealed to the untrusted server to violate the privacy of the

C. Douligeris et al. (Eds.): I3E 2013, IFIP AICT 399, pp. 25–37, 2013.

user. Therefore, the data confidentiality and user privacy need to be taken into account. To ensure data confidentiality, the data owner needs to hide the database's content before outsourcing it to the service provider. We also know that the privacy of a user is characterized by the query he poses to the server and its result. It is necessary to protect both the query and its result from the unauthorized parties (such as untrusted server) to protect the user privacy.

Encryption was often considered as a solution for data confidentiality ([2], [4], [5], [6]). The order preserving encryption scheme supported the equality and range queries over the encrypted data [4]. Other work on privacy homomorphism illustrated techniques for performing arithmethic operations (+, -, x, /) on encrypted data ([5], [6]). Hacigümüs et al. [2] proposed storing, together with the encrypted database, additional indexing information. By using the created index, the server could execute the queries over the encrypted data. There were four steps to process a query (Fig. 2): (1) the query Q posed by a user was translated by the query processor at the client site to its server-side representation Q^S (2) Q^S was sent to the server and was executed over the encrypted database (3) the result (in encrypted form) was sent to the client; it was decrypted and filtered out those tuples not satisfying the query condition (4) the final result was sent to the user by the client. All of the above-mentioned work revealed to the untrusted server the query type of users, which was the useful information for the server to predict the query being requested. Using these query execution techniques, the clients transfer to the server the same query patterns when the users pose the same query requests. The untrusted server may perform statistical attacks and exploit these query patterns. By correlating known public information with the frequent query patterns, together with the query types, the server can infer the users' trend of information, or more critical the users' trend of sensitive information, which violates the privacy of users [3].

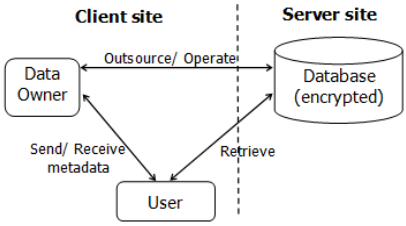


Fig. 1. Diagram of DAS

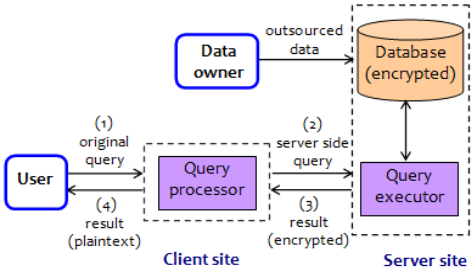


Fig. 2. Query execution process [2]

In this paper, we explore the techniques for executing SQL queries over the encrypted data without revealing to the server any information about the query or its result. Our proposal can defeat against the statistical attacks of the untrusted server compromising both data confidentiality and user privacy.

The remainder of this paper is organized as follows: section 2 presents the query execution solution for protecting the user privacy; section 3 presents the security analysis; section 4 is the experimental evaluation of our proposals; section 5 concludes the paper.

2 Our Proposed User Privacy Protection Technique

In this section, we propose a query execution technique which can protect the privacy of user. We adopt the database storage model and condition transformation technique proposed in [2].

2.1 Storage Model

For each relation r with the schema $R(A_1, A_2, \ldots, A_n)$, we store on the server an encrypted relation r^S with the schema $R^S(t^S, A_1^S, A_2^S, \ldots, A_n^S)$ where t^S stores an encrypted string that corresponds to a tuple in relation r, each A_i^S is a corresponding index to the attribute A_i that will be used for query processing at the server (Fig. 3). We can use any block cipher technique such as AES, RSA, Blowfish, etc., with the key size 128 bits. If there is unique user (also the data owner) in the system, we use one key for encrypting the whole database; otherwise, we use multiple keys which are managed by a key management mechanism [8, 9, 10]. The index is created based on the mapping function $Map_{R.Ai}(v)$, which will be defined as the following:

- The *partition function* which partitions the attribute's domain of values into disjointed partitions: $partition(r.A_i) = \{p_1, p_2, .., p_k\}$.
- The *identification function* $ident_{R.Ai}(p_j)$ which assigns an identifier to each partition p_j of attribute A_i.
- The *mapping function* which maps a value v in the domain of attribute A_i to the identifier of the partition to which v belongs: $Map_{R.Ai}(v) = ident_{R.Ai}(p_j)$, where p_j is the partition that contains v.

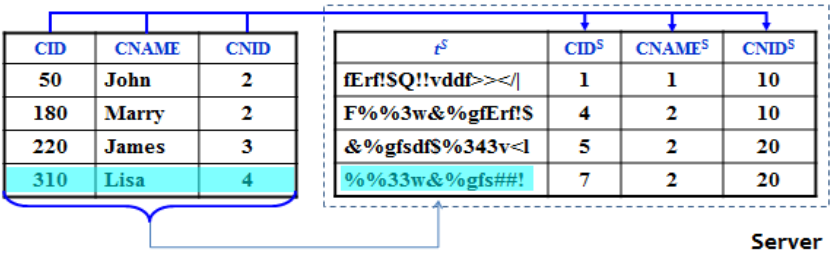


Fig. 3. Data storage model [2]

We use the operator D that maps the encrypted representation to its corresponding unencrypted representation. That is, $D(r^S) = r$. For differentiating the execution of an operation on the client site or on the server site, we denote the "S" in superscript form beside the operator with the suggestion to execute this operator on the server site. For example σ^S denote the selection operator is executed (on the encrypted data) at the server site. We denote R^+ for the set of all the attribute of r: $R^+ = \{A_1, \ldots, A_n\}$.

In our proposal, the principles of all the relational operators on each database management systems are unchanged. Besides the notation of the normal relational operators, such as $\prod$, σ, $\bowtie$, $\cup$, $\cap$, -, $\leftarrow$ which stand for the projection, the selection, the join, the union, the intersection, the set difference, the assignment respectively as

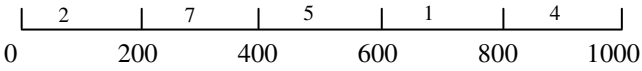
defined in [1], we use the additional ones: τ denotes for the sorting operator, and γ denotes for the grouping and aggregation operator.

2.2 Condition Transformation

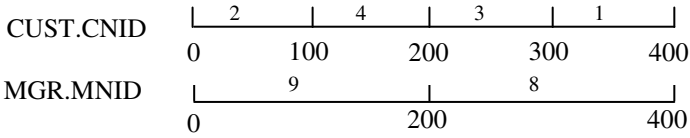
The condition mapping function, Map_{cond}, translates a condition from a user's query to its corresponding one over the server-side representation [2]. This section we only refer to random mapping of two popular types of condition, which will be used in the next section. For the more consideration, please refer to [2]. We will use these two relations for illustrations:

CUST (CID, CNAME, CNID)
MGR (MID, MNAME, MNID)

Attribute = Value: $Map_{cond}(A_i = v) \Rightarrow A_i^S = Map_{Ai}(v)$. For example, $Map_{cond}(CID = 250) \Rightarrow CID^S = 7$.



Attribute$_1$ = Attribute$_2$: $Map_{cond}(A_i = A_j) \Rightarrow \bigvee_{\varphi} (A_i^S = ident_{Ai}(p_k)$ và $A_j^S = ident_{Aj}(p_l))$, where φ is $p_k \in$ partition (A_i), $p_l \in$ partition(A_j), $p_k \cap p_l \neq \varnothing$.



For example, $Map_{cond}(CUST.CNID = MGR.MNID) \Rightarrow (CUST^S.CNID^S = 2 \wedge MGR^S.MNID^S = 9) \vee (CUST^S.CNID^S = 4 \wedge MGR^S.MNID^S = 9) \vee (CUST^S.CNID^S = 3 \wedge MGR^S.MNID^S = 8) \vee (CUST^S.CNID^S = 1 \wedge MGR^S.MNID^S = 8)$.

2.3 Solution for Protecting Access Pattern Privacy

Principles. Our solution was based on three principles: (1) all the relational operators (from the client query) are implemented by doing only the selection operator over the server-side database (2) we select (n + m) elements in each time of selection over the server-side database, where n and m are the parameters which determine the security level of our proposed system (3) minimizing the work done at the client side.

Conforming to the principle 1, the untrusted server cannot recognize the type of the query that is being requested. Respond to whatever the query type required by the user, the server simply does the selection. The algorithm Select_NTimes is used by the client for dispatching the selection request(s) to the server and receiving the result in encrypted form. The number of elements requested in each time depends on the total number of index values being requested and the values of n and m.

Principle 2 prevents the server from doing the statistical attacks to learn the frequent query pattern. In the case the users request the same query, the sets of

elements the corresponding client request from the server in each time of selection are different with the high probability. The appropriate values of n and m should be suggested by the data owner or an expert in the field involved. We will analyze the security of our proposed techniques in section 3. Principle 3 keeps the spirit of database outsourcing service in which most of work should be done by the server.

Solution for Selecting Data from the Server. All the relational operations over the encrypted database will be implemented by using the Select_NTimes algorithm. Note that GetRand(n, I) is the function for getting randomly n elements from the set I while card(I) is the function for returning the cardinality of the set I.

Algorithm Select_NTimes(r(R), A, I, n, m) For selecting from the encrypted relation $r^S(R^S)$ of relation r(R) the tuples with the value at the attribute A^S belonging to the set of values I, $I \subseteq Ident(R.A)$

```
T = ident(r.A) – I; R = ∅
While card(I) > 0
Begin
        N = ∅
        If card(I) <= n then
            Begin
                L = I
                If card(I) < n then
                        N = GetRand(m + n – card(I), T)
                Else if card(I) = n then
                        N = GetRand(m, T)
                I = ∅
            End
        Else // card(I) > n
             Begin
                If card(I) <= n+m then
                Begin
                        L = GetRand(n, I);
                        N = GetRand(m, T)
                End
                Else // card(I) > n+m
                        L = GetRand(n+m, I)
                        I = I – L
             End // card(I) > n
        Z = L ∪ N
        R1 ← σ^S_{A^S ∈ Z}(r^S)
        R2 ← σ^S_{A^S ∈ L}(R1)
        R ← R ∪ R2
End While
Return R
```

The Select_NTimes algorithm operates in the following manner. Let T be the set of values of A^S except the values in I. If the cardinality of the requested set I is less than n, the client adds to I the values in T in order to have a set Z having the cardinality (n+m) in each time of selection. In the case the cardinality of I is equal to n, the client conforms to principle 1 to add m values in T more. In the case the cardinality of I is greater than n but less than (n+m), the client to get randomly n values from I and add together with m ones getting randomly in T for each time of selection over r^S. If the cardinality of I is greater than (n + m), the client flexibly selects (n +m) values randomly from I for each time of selection over r^S. When receiving the result returned from the server, the client should remove the spurious tuples for saving the cost for decrypting them (see Fig. 4).

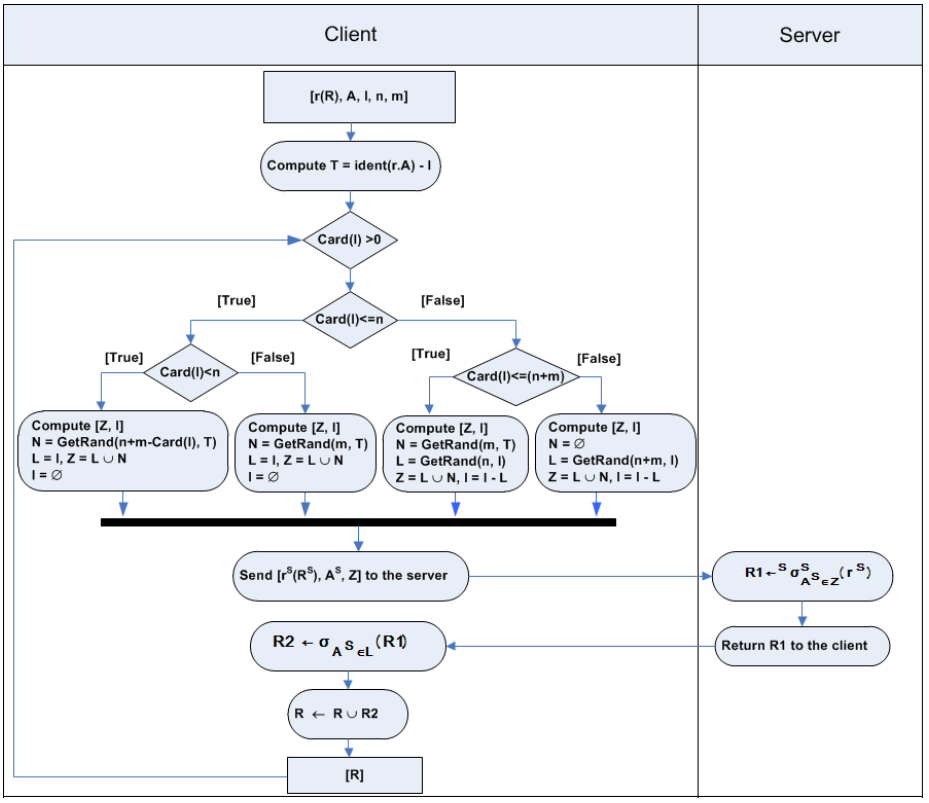


Fig. 4. Select_NTimes algorithm

We suggest decrypting the result one time after selecting from the server all the satisfied rows. By this way, we save the time of using resource of the client. However, it requires the client to store the result before decrypting all of them. An alternative way is to apply the client side operation to the tuples arriving over the answer stream as soon as they arrive without the need to store them.

We knew that sorting the input data is necessary for implementing operations such as join, union, intersection, duplicate elimination. Every tuple belonging to a single group of γ_L will be in a single group of $\gamma^S_{L'}$ computed by the server. The client only needs to consider tuples in a single group of $\gamma^S_{L'}$ when computing the groups corresponding to γ_L. The sort operator τ_L on the tuples having been grouped by the server can also be implemented efficiently using the merge-sort algorithm. These are the reasons why we design the algorithm, named Select_NTimes_Grouped, which has the same function as Select_NTimes except that the result set is grouped according to the specified attributes after every selection done by the server. Algorithm Select_NTimes_Grouped(r(R), A, I, n, m, L) is used for selecting from encrypted relation $r^S(R^S)$ corresponding to the relation r(R) the tuple(s) having the value(s) at attribute A^S belonging to the set of values I. The returned result is grouped by L^S, the correspondence of L.

By using Select_NTimes_Grouped algorithm, the operations that need the input data to be grouped or sorted (such as join, grouping and aggregation, sort, duplicate elimination) be implemented efficiently.

Implementation of Relational Operators

- **Selection operator**

Algorithm Selection (r(R), A, C, n, m) For selecting from relation r(R) the tuple(s) having the value(s) at attribute A satisfying condition C

$Map_{cond}(C) \Rightarrow A^S \in I$;
result ← Select_NTimes(r(R), A, I, n, m)
return σ_C(D(result))

We explain the above implementation using the example: $\sigma_{CID = 500}$ (CUST). $Map_{cond}(CID = 500) \Rightarrow CID^S \in \{5\}$, which means I = {5}. Suppose that the parameters for attribute CID of relation CUST are n = 2 and m = 2, according to the algorithm Select_NTimes, because card(I) = 1 < n (n = 2), the client will choose (m+n – card (I)) = (2+2 -1) = 3 elements randomly from ident(CID) – I = {2, 7, 5, 1, 4} – {5} = {2, 7, 1, 4}. Suppose that 2, 1 and 4 are selected, which means N = {2, 1, 4}. The client requests the set L ∪ N = {5, 2, 1, 4} rather than requests only I = {5}. The client then decrypts the result and filters out the tuples satisfying the condition C.

- **Join operator**

Algorithm Join (r(R), t(T), C, n, m) For returning the result of $r \bowtie_C t$, C is a θ- join condition
I, J contain all possible partitions of A_i, A_j that exists at least one pair of them may provide some values of A_i and A_j that can satisfy the condition C: $A_i \theta A_j$.
result1 = Select_NTimes_Grouped(r(R), A_i, I, n, m, A_i)
result2 = Select_NTimes_Grouped(t(T), A_j, J, n, m, A_j)
result = σ_C(D(result1 $\bowtie_{Map_{cond}(C)}$ result2))

return result

For instance, with the join condition C: CUST.CNID = MGR.MNID, Map_{cond}(CUST.CNID = MGR.MNID) $\Rightarrow$ (CUST^S.CNID^S = 2 $\wedge$ MGR^S.MNID^S = 9) $\vee$ (CUST^S.CNID^S = 4 $\wedge$ MGR^S.MNID^S = 9) $\vee$ (CUST^S.CNID^S = 3 $\wedge$ MGR^S.MNID^S = 8) $\vee$ (CUST^S.CNID^S = 1 $\wedge$ MGR^S.MNID^S = 8)

For using our proposed JOIN algorithm, we identify that A_i is CNID, A_j is MNID, I = {2, 4, 3, 1}, J = {9, 8}. The client selects all the rows from the relation r^S satisfying the condition 'CNID^S in I', which resulted in result1. The client also selects all the rows from the relation t^S satisfying the condition 'MNID^S in J', which resulted in result2. The client executes the join operation between result1 and result2 with the join condition is Map_{cond}(C), which resulted in result. The client continues executing the decryption operator on result and selecting the rows satisfying the condition C.

- **Projection operator**

Algorithm Projection (r(R), n, m, L) For returning the projection of r(R) on the projection attribute(s) L

$A \in R^+$; I = ident(A)
result = Select_NTimes(r(R), A, I, n, m)
return $\prod_L$(D(result))

The projection operator cannot be implemented on the server because each tuple of r is encrypted together into a single string in the t^S attribute of r^S. After selecting all the rows of the relation r, the client decrypts the result and performs the projection.

- **Grouping and aggregation operator**

Algorithm Group_Aggregation(r(R), L, n, m) For returning the values of aggregation functions operating on each group
L = $L_G \cup L_A$, L_G contains attributes on which the grouping is performed; L_A corresponds to a set of aggregation operations.
I = ident(A_i); $A_i \in L_G$
result = Select_NTimes_Grouped(r(R), A_i, I, n, m, L_G)
return γ_L(D(result))

The grouping and aggregation operation is denoted by $\gamma_L(r)$ where $L = L_G \cup L_A$. L_G is the list of attributes on which the grouping is performed while L_A is the set of aggregation operations. The server does not perform any aggregation corresponding to L_A. It returns all the rows of the relation r in responding to the client's request using Select_NTimes_Grouped algorithm. This result has been grouped by the server on the corresponding group of L_G. The client decrypts this result and performs the grouping operation and computing the aggregation functions specified in L_A.

- **Sort operator**

Algorithm Sort (r(R), L, n, m) For sorting the tuples of r(R) by L
I = ident(A_i); $A_i \in R^+$
result = Select_NTimes_Grouped(r, A_i, I, n, m, L)
return τ_L(D(result))

The sorting operator is implemented similarly to the grouping operator. The client firstly selects all the rows of the relation r using the Select_NTimes_Grouped algorithm. This result has been grouped by the server on the encrypted attributes of those in L. The client then decrypts the result, performs the sorting operation on the attributes in L. If the mapping functions of the attributes in R^+ are all order-preserving, the grouping operation operated on each part of the result returned by the Select_NTimes_Grouped should be replaced by a corresponding sorting operation for saving the cost at the client. The reason is that the result returned by the server is presorted within the partition. Sorting the result is a simple local operation over a single partition.

The following three set operators must be executed on the two compatible relations. These operators are implemented by the same manner. They cannot be executed by the server because on the encrypted form of the relations r and t, it is impossible to tell whether or not a given tuple satisfies the current operator. The client firstly selects all the rows of two relations r and t using the Select_NTimes_Grouped algorithm, except the union operator (without duplicate elimination) using the Select_NTimes algorithm. The client then decrypts the results and performs the corresponding operation.

- **Set operators**

Algorithm Difference (r(R), t(T), n, m) For returning the difference between r(R) and t(T)
$A \in R^+$; $B \in T^+$; A and B have the same domain value
I = ident(A); J = ident(B)
result1 = Select_NTimes_Grouped(r(R), A, I, n, m, A)
result2 = Select_NTimes_Grouped(t(T), B, J, n, m, B)
return D(result1) – D(result2)

Algorithm Algorithm Union (r(R), t(T), n, m) For returning the union of r(R) and t(T)
$A \in R^+$; $B \in T^+$; A and B have the same domain value.
I = ident(A); J = ident(B)
result1 = Select_NTimes(r(R), A, I, n, m, A)
result2 = Select_NTimes(t(T), B, J, n, m, B)
return D(result1) $\cup$ D(result2)

Algorithm Intersect (r(R), t(T), n, m) For returning the intersection between r(R) and t(R)
$A \in R^+$; $B \in T^+$; A and B have the same domain value.
I = ident(A); J = ident(B)
result1 = Select_NTimes_Grouped(r(R), A, I, n, m, A)
result2 = Select_NTimes_Grouped(t(T), B, J, n, m, B)
return D(result1) $\cap$ D(result2)

3 Security Analysis

Our proposed system use two parameters n and m. n is used for preventing the server from predicting the query type of the user based on the number of values to be requested. For example, when requesting an exact match query, the number of requested values is 1, which contrasts to the projection query with a larger number of requested values. m is used with the purpose of making noise to prevent the server from predicting the query type in the case the cardinality of the requested set is small (card(I) < n+m).

There is the trade-off between the security level and the communication and computation cost in our proposed system. The higher the value m is, the harder for the server to predict exactly the query is being executed. When the value of m is large enough, the probabilities of being selected of all the values in the considering domain are similar to each other, which creates difficulty for the server to predict the query type or the query pattern of users. However, the high value of m affects the performance of the system.

In the case the cardinality of I is greater than (m+n), each time of selection the client choose one set in N sets of values for selecting from server: $N = C_{card(I)}^{n+m}$. For preventing the case the server finding the intersection of the requested sets for predicting the query pattern, the value of N must be large enough. The higher the value of N, the more secure the system is. N becomes maximum if (n+m) approximates to card(I)/2.

Every attribute to which there may be have the query relates to should be set the values of n and m. For security reason, all the selection conditions done on an attribute should use the same values of n and m.

4 Experimental Evaluation

We present the experimental evaluation of our proposal. We implemented our proposed query execution method and the one suggested by Hacigümüs et al. [2], called Hacigümüs, and compared the query execution time between them.

By utilizing TPC-H benchmark [7], we generated two relations containing information about customers and mangers: CUST (CID, CNAME, CNID) and MGR (MID, MNAME, MNID). These attributes mean customer's identity, customer's name and customer's nation identity. The attributes of MGR relation have the same meanings as ones in CUST. We generated 150000 rows for the relation CUST, with the CID ranged from 1 to 150000. We generated 1000 rows for the relation MGR. The nation identity attributes (CNID and MNID) ranged from 1 to 25.

Our experiments were carried out on an Intel© Core2 Duo Processor P8700 2.53GHz, 4GB RAM. Relevant software components are Windows 7 as the operating system, SQL Server 2005 as the database management system and Microsoft Visual Studio C++ 2008 as the programming language. We used the equi-width technique to partition the domain of attributes CID, CNID, MID and MIND. The domains of

attributes CID and MID were partitioned into fragments, each fragment contained 49 integer values. The domain of attribute CNID was partitioned into 5 fragments while the domain of attribute MNID was partitioned into 3 fragments. We considered four queries: one exact match selection, one range selection, one join and one projection.

Q1: SELECT * FROM CUST WHERE CID = 500; Q2: SELECT * FROM CUST WHERE CID >= 500; Q3: SELECT * FROM CUST, MGR WHERE CNID = MNID; Q4: SELECT CID, CNAME FROM CUST;

For the query Q1, the condition after mapping was $CID^S = 10$. Firstly, we executed it 5 times using the execution process proposed by Hacigümüs et al. [2] and recorded the execution time. Secondly, by using our proposed access pattern privacy protection techniques, we executed Q1 with the value of n was 2 and the value of m run from 2 to 6, and computed the average execution time. We also repeated it 5 times. Thirdly,

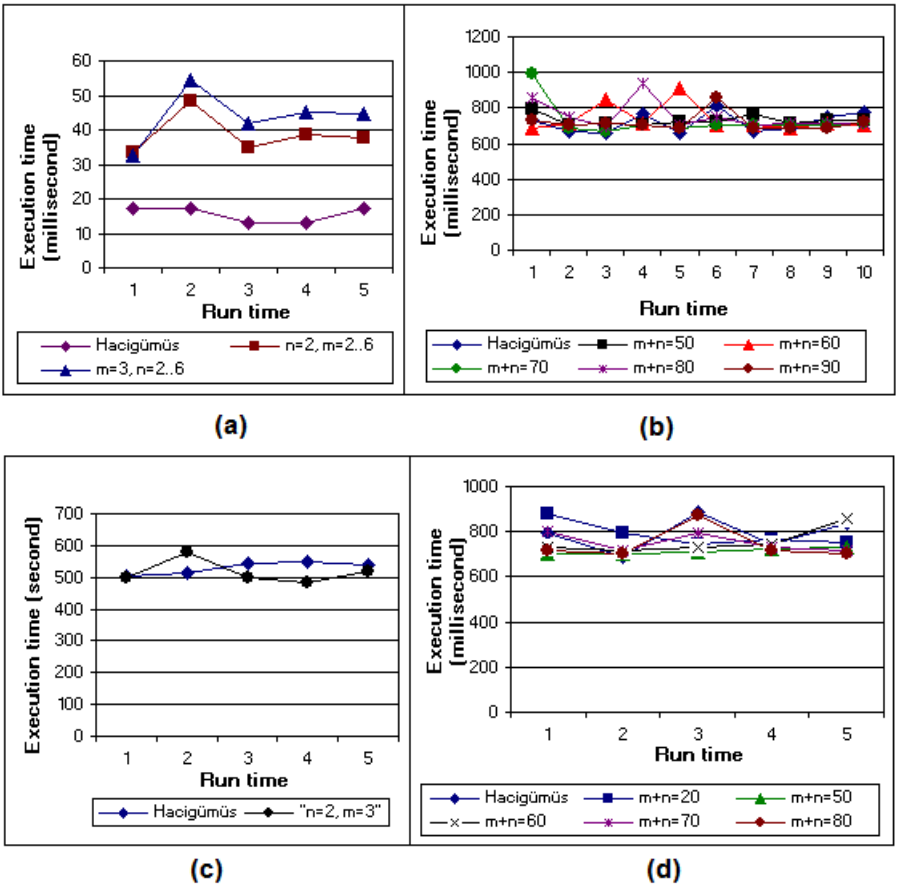


Fig. 5. Execution time when using our access pattern protecting solution comparing with that when using techniques of Hacigümüs el al.: (a) Q1 (b) Q2 (c) Q3 (d) Q4

we did the same things with Q1 as the second execution time with the value of m was 3 and n run from 2 to 6. For each pair of n and m, we run the query 10 times with different random sets of selection values according to the Select_NTimes algorithm. Fig. 5 (a) demonstrates that there is the small difference between the execution time of the two latter running times. The amount of execution time of these two running times is certainly higher than that of Hacigümüs et al. because the higher number of selection values were selected from the server.

For the query Q2, the condition after mapping was the set containing 290 values. We executed Q2 using Select_NTimes algorithm with the cardinality of the selection sets of values in each time of execution the selection (that was m+n) varied by 50, 60, 70, 80, 90. For each value of (m+n), we run the query 10 times with different random sets of selection values. Fig. 5 (b) demonstrates that the execution time when using our proposed execution technique is the same as that when using Hacigümüs' one. The differences in the execution time between the values of (m+n) are low during 10 times of running the experiment.

Executing Q3 by using Select_NTimes_Grouped algorithm (with n= 2 and m = 3) costs the same amount of time as that by using Hacigümüs' one, Fig. 5 (c). The result of the join operator contains 599988 rows which need more than 8 minutes to produce.

The result of the query Q4 contains all the rows of the CUST relation (150000 rows). We executed Q4 using Select_NTimes algorithm with the cardinality of the selection sets of values in each time of execution the selection (that was m+n) varied by 20, 50, 60, 70, 80. For each value of (m+n), we also run the query 10 times with different random sets of selection values. What we see on the Fig. 5 (d) is that the differences in execution time between the values of (m+n) are small, and the execution time of our proposed techniques is the same as that of Hacigümüs' one.

5 Conclusion

In this paper, we analyze the existing solutions for protecting data confidentiality and user privacy in DAS. The recent and well-known proposal of Hacigümüs el al. [2] is expressive but cannot defeat against the statistical attacks of the untrusted server, which may violate the data confidentiality and the user privacy. We propose the simple but robust technique for executing the relational operators over the encrypted database which can protect both the data confidentiality and the user privacy. Experimental evaluation demonstrates that our proposal less affects the system's performance and is applicable in the real world.

References

1. Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems, 4th edn. Addison-Wesley (2004) ISBN 0-321-12226-7
2. Hacigümüs, H., Iyer, B.R., Li, C., Mehrotra, S.: Executing SQL over encrypted data in the database-service-provider model. In: SIGMOD, pp. 216–227 (2002)

3. Sion, R.: Towards Secure Data Outsourcing. In: Handbook of Database Security, pp. 137–161 (2008)
4. Agrawal, R., Kierman, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. In: Proc. of ACM SIGMOD 2004, France (2004)
5. Boyens, C., Gunter, O.: Using online services in untrusted environments – a privacy-preserving architecture. In: Proc. of the 11th European Conference on Information Systems (ECIS 2003), Italy (2003)
6. Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L.: Authenticated Index Structures for Aggregation queries in Outsourced Databases, Technical Report BUCS-TR-2006-011 (2006)
7. TPC-H. Benchmark Specification, `http://www.tpc.org`
8. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Over-encryption: Management of Access Control Evolution on Outsourced Data. In: VLDB, pp. 123–134 (2007)
9. Atallah, M.J., Frikken, K.B., Blanton, M.: Dynamic and efficient key management for access hierarchies. In: Proc. of the ACM CCS, Alexandria, VA, USA (2005)
10. Hue, T.B.P., Luyen, G.N., Kha, N.D., Wohlgemuth, S., Echizen, I., Thuc, D.N., Thuy, T.B.D.: An Efficient Fine-grained Access Control Mechanism for Database Outsourcing Service. In: Proc. of the Int. Conf. on Information Security and Intelligent Control (ISIC 2012), pp. 67–71. IEEE Computer Society Press, Taiwan (2012) ISBN 978-4673-2586-8