Lamsade Work in Progress Rep. Jan 15, 2014

SQL Numerical Value Expressions Over Encrypted Cloud Databases

Sushil Jajodia[1], Witold Litwin[2], Thomas Schwarz[3]

Position Statement[4]

Abstract. Cloud databases may need encryption. Encryption however impairs queries. Evaluating value expressions, especially the numerical ones, may be impossible in practice at present. Fully homomorphic encryption schemes remain impractical. The additively homomorphic Pailler cryptosystem seems practical for addition only value expressions. We propose a scheme derived from this encryption, supporting also other numerical operations in value expressions, e.g., the multiplication. It also allows for the SQL scalar and aggregate functions. The scheme uses encrypted tables that are conceptually the old-fashioned mathematical tables. We present the scheme, discuss its properties and conclude over remaining issues.

1 Introduction

It is the common knowledge that cloud databases often need the client-side encryption. However, the encryption impairs SQL queries, [T3], [D3], [S4]. In particular, queries with value expressions may be impossible over the usually encrypted data, e.g., using AES. To send many data to the client to process them decrypted instead appears often impractical. For numerical SQL value expressions, fully homomorphic encryption schemes allow for all basic arithmetical operations over the encrypted data, i.e., (+,-,*, /, ^). Additively homomorphic ones provide basically for '+' only, [S4]. The former are yet very far from being practical. Among the latter, Pailler cryptosystem scheme seems the closest to be practical [P9], [T3], [S4], [E], [T]. However, SQL queries limited to additions only are by far insufficient in practice. One approach is to decompose a query with a value expression using an operation other than '+' so to perform that operation over the decrypted (plaintext) data, i.e., at the client. There can be also the per-row pre-computation of a plaintext expression over attributes of a table, e.g., of a*b, for the encryption of the computed (dynamic, virtual…) attribute afterwards, [T3]. Yet another approach is to encrypt only the essential data which possibly are also not subject to value expressions [D]. The future of all these proposals remains to be seen.

Below we propose a different approach. We encrypt the cloud stored data susceptible to be subject of a value expression through a novel scheme, based on the Pailler calculation. The encrypted data, on the one hand, are the usual relational ones, e.g., prices of some items or quantity to deliver, stored in tables defined by the conceptual schema. Besides, we also encrypt all the numeric values susceptible to be in the database or to result from a value expression, e.g., every possible price and quantity, and every possible result of price*quantity…. All such values are stored in one or more auxiliary (not in the conceptual schema) tables that we call *scalar function* tables (SFTs). An SFT is conceptually an old-fashioned mathematical table, e.g., $\log_{10} X$ tabulated for X = 1, 2… [W]. It appears in the database as a relational table. The DBA creates it and load to the cloud, prior to the operational use of the database.

Each row in an SFT contains the encryption of a possible datum and of the results of the scalar functions that might apply to the datum. Like a mathematical table, the SFT tabulates enough values for the practical use of the database. It may contain nevertheless many more rows than traditionally in a mathematical table, e.g., millions. As we show this is however not a practical problem for the present storage technology. An SFT is furthermore the basis for, also old-fashioned, rules for rewriting the operations other than the addition in the original value expression. The rewritten expression is calculated encrypted using the additions and the lookups into the SFTs only. For instance, to compute the product $x*y$ of encrypted x and y, we apply the high-school rule $x*y$ = antilog (log x + log y). Both encrypted logs are tabulated in the SFT. Log x is in some row, tabulating the scalar functions over x, with x itself

[1] George Mason University, Fairfax, Virginia, USA, jajodia@gmu.edu
[2] Dauphine University, Paris, France, witold.litwin@dauphine.fr
[3] Univ. Catolica del Uruguay, Montevideo, Uruguay, tschwarz@ucu.edu.uy
[4] Electronic Report. Last update: January 23, 2014.

encrypted in some column, say A, of the row. Log y is similarly in some row with A = y. The value expression log x + log x is calculated encrypted through Pailler encryption formula. The encrypted antilog is found again in A, in the row having the computed sum in some column dedicated to the tabulation of such sums. The whole calculation takes place on the server. When needed, the decryption at the client renders the plaintext $x*y$.

Next section introduces our scheme. It recalls first the Pailler system. On this basis, we show our encryption rules. After a motivating example, we describe the structure of an SFT. Next we show some rewrite rules and discuss our safety model. Section 3 overviews the performance analysis. We conclude in Section 4, where we also discuss the further work.

2 Scalar Function Tables

2.1 Pailler Cryptosystem

Pailler cryptosystem defines an additively homomorphic encryption [P9]. Below we recall only the properties most relevant to our goal. Formally, the values to encrypt/decrypt (plaintext/ciphertext) are integers in Z_n for a specific large n. Let x and y are two plaintexts to encrypt and $\boldsymbol{x}$ and $\boldsymbol{y}$ are their encryptions. Let D denotes the decryption operation, as defined in [P9] or [P]. We have the additively homomorphic encryption since (1) $x + y \bmod n = \mathrm{D}\ (\boldsymbol{x} * \boldsymbol{y} \bmod n^{\mathbf{2}}) \bmod n$. In addition, (2) we can add a ciphertext and a plaintext. The reason is that for any randomly chosen g in Z_{n2}, $x + y \bmod n = \mathrm{D}\ (\boldsymbol{x} * g^y \bmod n^{\mathbf{2}})$. Finally, (3) we can multiply a ciphertext by a plaintext, since $x*y \bmod n = \mathrm{D}\ (\boldsymbol{x}^y \bmod n^{\mathbf{2}}) = \mathrm{D}\ (\boldsymbol{y}^x \bmod n^{\mathbf{2}})$.

The scheme uses (g, n) as the encryption key. This key is public. To encrypt a plaintext x, the user first chooses randomly an integer r in Z_n^*. The ciphertext $\boldsymbol{x}$ is then produced as $\boldsymbol{x} = g^x * r^n \bmod n^2$. An x encrypted twice, may use two different r's or the same r. In other words, the scheme is a probabilistic one. Remarkably, D decrypts x regardless of r. Notice also that if we used r_1 for $\boldsymbol{x}$ and r_2 for $\boldsymbol{y}$, then $\boldsymbol{x} + \boldsymbol{y}$ becomes encrypted using $r_3 = r_1 * r_2$. If we use in particular one r for both ciphertexts, then we have $\boldsymbol{x} + \boldsymbol{y}$ encrypted simply with r^2.

A ciphertext value uses more storage than the plaintext one. Existing implementations basically use 128B or 256B for it. Also the encrypted addition timing is substantially longer. For the final result, one has to add the decryption time as well. We come back to these aspects of the scheme in Section 2.6 and after.

2.2 Our Scheme

For reasons that will appear, we consider that DBA choses r randomly a single while creating the database. There are then two encryption keys: (g, n, r) and (g, n, r^2) used with Pailler formula. We'll talk from now on about r-encryption and r^2-encryption. Both keys are private to the client(s) and DBA. Key (g, n, r) serves for the query encryption, e.g., of values to enter the database. Property (2) and (3) mean no need for the encryption of the query parameters, i.e., constants, in a value expression.

Most of SFT attributes are also r-encrypted. The others are r^2-encrypted. Decryption key is the same for the original and our scheme. It is thus (λ, μ) in notation from [W]. It is also private, of course.

2.3 Motivating Example

Consider a relational table in some cloud database with many goods and their per unit prices, in an attribute called Price for every item. The final price for a client is the unit price multiplied by some quantity, say Qty in other table(s). Let us call TotalPrice this dynamic attribute. We suppose that as often, Price and Qty are basically in many-to-many relationship. This precludes the pre-computing of TotalPrice as in [T3]. The SQL query involves the value expression "Price * Qty as TotalPrice". The query should be processed over the cloud. The Price data are real numbers with two decimal digits but the precision of 10c, e.g., \$43.20. The range is from \$0.10 to \$100.00. The quantities are of type integer, say, 1… 100. The data type of TotalPrice is the one of Price, but ranging \$0.10…10,000.00. All the values involved should remain encrypted.

To process the query, we consider that the cloud database stores somewhere an SFT, say F (<u>A</u>, A', F1, F1'). All these attributes are encrypted. Every value in A and F1 is r-encrypted, while A' and F1' are r^2-encrypted. A in particular, is the primary key where each value, say $\boldsymbol{a}$, is the r-encrypted value a allowed to enter a value expression using an operation other than +. In other words, A tabulates the encrypted numerical domain (data type) for some or all numerical attributes in the cloud database. For each value a allowed in the database for the attributes Price, Qty and TotalPrice, there is thus $\boldsymbol{a}$ in A. Attribute A' contains respectively the r^2-encryption of the same plaintexts. Next, we suppose that each ***b*** in F1 is *r*-encrypted log *a*, while each ***b'*** in F1' is the r^2-encryption of log *a*. This could be log $_e$ *a*, as in MsAccess. Hence each ***a*** is also the *r*-encrypted antilog of *b* (the plaintext of ***b***) and r^2-encrypted antilog of *b'*. In our example, it suffices that the SFT has 100,000 rows. Finally, according to our basic safety model, presented in Section 2.7, whether a column of F is A or F1 or F1' is not known at the cloud.

Consider that the query involves a specific item with Price whose plaintext value happens to be 123. The query also somehow determines the related quantity in the database that is Qty = 45. The value expression processing is as follows. Consider that Price and Qty found in the database are encrypted as 321 and 654 respectively. The cloud getting the query selects at some point F1 from F where A = 321 or A = 654. The result would be two values that must be both in F, by definition. Say the result is (b_1, b_2). The cloud then performs the addition $b_1 + b_2$, defined by (1) above. Say the (encrypted) result is '7654'. The cloud selects A from F where F1' = '7654'. The cloud uses F1', since the result of an addition is here encrypted using r^2, as mentioned in previous section. Say, the result is *a* = '5432". The cloud sends *a* to the client for decryption.

Here, the cloud manipulates F as a relational table. Physically, there are many ways to implement it. One is to have a hash file with a as primary key. Also, a hash index on B may be useful obviously. In our example, the total storage for F with its 100,000 rows should be a couple of MBs. This is clearly negligible for current RAM sizes. Hence, our F could even be in a RAM cache.

Another motivating query could be the request for Total PriceTTC calculated as Price * (Qty – QtyOnHand) * 1.19. Here, QtyOnHand is supposed to be the quantity still available, hence to subtract from the usual one to order (Qty). The constant reflects the 19% VAT. The query is clearly quite typical. With respect to the previous one, it illustrates additional need for (i) the encrypted subtraction and for (ii) the multiplication by a plaintext constant. The latter calculation is straightforward through property (3) above. In contrast none of the discussed properties let us calculate the former. It also illustrates the need for A'. The result, say *x*, of the subtraction is indeed "automatically" r^2- encrypted. To multiply it by Price using property (1), one has to select indeed the row with log *x*. However, A cannot be used for this as it is *r*-encrypted. Unlike A'.

Likewise, the client may wish Total PriceTTC results appearing in descending order. Also, perhaps limited to, say 10 biggest values. Next, one may wish to know the average value and the maximal one. Perhaps this value should be rounded up to its integer part only or to a single decimal digit etc. Also, the client may wish to see only the prices above a threshold, say 123, given as parameter. All these needs are trivially satisfied over plaintext data, using the SQL OrderBy Desc and TOP 10 clauses, AVG (Total PriceTTC) aggregate function with perhaps Int or Round scalar functions or the selection predicate Total PriceTTC > 123. In contrast, the properties of scheme do not provide for these needs over the accordingly encrypted cloud database. The situation is similar with respect to other scalar functions. What follows, complements therefore the Pailler scheme to provide also for the motivating needs.

2.4 SFT Structure

The basic numerical data types for SQL numerical value expressions are positive and negative integer and real numbers. We thus suppose also some usual fixed or floating point real numbers representation and the associated arithmetic over Z_n, [GZ]. We also assume some usual representation of negative values. An SFT is then a table, say again F, but with more attributes than above, i.e., F (A, A', F1, F1', F2, F3…). Here, again, each *a* in the primary key A is some integer or real, *r*-encrypted, i.e., using the encryption key (*g*,*n*,*r*). Same for A' except for r^2-encryption. Each Fi is an *r*-encrypted result of numerical scalar functions that an SQL value expression may invoke over the plaintext *a*, *r*-encrypted in A in the same row. Finally, as already mentioned, A' and F1' contain the r^2-encryption of *a*.

SQL scalar functions are not standardized. Hence, the actual set depends on underlying DBMS. For our purpose, we consider the following Fi's.

F1. Together with F1', these are the already discussed log functions.

F2. The minus function, i.e. F2 (*a*) = minus (*a*) = - *a*.

An SFT tabulates *a* value with some fixed or variable increment and precision over the interval covering the application practical needs. Again, like for the mathematical tables and in the motivating example. These values are basically the only allowed for a numerical attribute subject to a query with a value expression using an operation other than addition. For a fractional increment, the following other scalar functions appear useful:

F3. The Int (a), e.g., Int (2.6) = 2, Int (-2.3) = -3
F4. The Round (a), e.g., Round (2.6) = 3, Round (-2.3) = -2
F5. The Round (a, 1), e.g., Round (2.66) = 2.7
F6. The Round (a, 2),
F7. The Round (a, 3) etc., but we suppose here that the 3-digit precision suffices for most apps.
F8. The Abs (a), if a may be negative and the application needs this function.

We recall again that the values of a known at the cloud are all Pailler encrypted. There is no practical way or perhaps no way at all, to calculate the discussed functions at the cloud for this encryption. The attribute A itself is also implicitly antilog f_1 also called Exp (f_1) function. Basically, for safety, we consider that attribute names in an SFT are meaningless, e.g., *Fi*, as usual for a relation in an encrypted cloud database. For clarity, however, we'll refer to an attribute *Fi* sometimes by the name of the scalar function it tabulates.

2.5 Rewrite Rules for Value Expressions

The basic operations in an SQL value expression are +, -, *, / and ^. The exponentiation ^ is always a positive or zero integer. In some systems, it may also be fractional and/or negative. Our basic rewrite rules are as follows. They are basically applied at the client to positive values (see also Section 2.7). The query format somehow distinguishes the integers representing the plaintext and from those of the ciphertext.

1. For $a + b$. The rewrite is $\boldsymbol{a} + \boldsymbol{b}$ if both are supposed encrypted. This triggers the calculation (1) above at the cloud. Otherwise, if a is a constant, the rewrite is $a + \boldsymbol{b}$. And vice versa. Both cases trigger the calculation (2). The rule extends to $a+b+c$… through the rewrite $(a+b)+c)+$…. and the switch from A' to A at each ')'. The rationale is to calculate the sum using r^2 at most. Otherwise, the sum of i ciphertexts would generate the value encrypted using (g, n, r^i).
2. For -a. The rewrite is minus (a) what we may write from now –a as well, for convenience. Notice that the values manipulated are in the finite field, Z_n. In (pseudo) SQL to calculate –a, the cloud simply executes: select minus from F where A = a.
3. For $a – b$. Following (2), the rewrite is $a + (-b)$.
4. For $a * b$. The rule is A (log a + log b), i.e., exp (log a + log b). Here both logs are searched in F1, but if $\boldsymbol{y}$ is the result of the addition, then A is found as select A from SFT where $F1' = \boldsymbol{y}$**.** The rule extends to a*b*c… through the rewrite as (a*b)*c)… and the switch from F1' to F1 at each ')'. The rationale is the same as for Rule (1).
5. For 1 / a. It becomes exp (**-** (log a)).
6. For a / b. It gets rewritten as exp (log a + (**-** log b)). F1' gets involved as for (4).
7. For a ^ b. The basic rule is exp (exp (log b + log (log a))). It gets amended trivially if b is a fraction or is negative and the related log values are not tabulated. F1' gets again involved as for (4).
8. For $k * a$, where k is a plaintext constant in the query, while a is encrypted. The rewrite is $\boldsymbol{a}^k$. Here, $\boldsymbol{a}$ denotes the encrypted value of a, i.e., the actual content in A. After the decryption, the result is $k * a$ by properties of the scheme.

2.6 Rewrite Rules for Aggregate Functions

The main SQL aggregate functions that need arithmetical calculus are, as one knows, SUM, AVG and VAR. The 1st one by definition is equal to AVG (a) = SUM (a) / COUNT (a). COUNT can obviously be calculated over the encrypted data. The whole expression can be then evaluated also encrypted through rule (6). The rewriting of VAR is also easy to see, although more cumbersome than for AVG. Recall that, by definition, we have indeed VAR (X) = AVG $(x – \text{AVG}(X))^2$ for every $x \in X$.

With the above listed functions only in SFT, the cloud cannot evaluate the expression with the aggregate functions MAX and MIN, e.g., select MAX ($x * y$). The client has to do it as post-processing. The incidence on the overall query processing is beyond the scope of this work. Alternatively, one may tabulate also the plaintext Rank function for A values, ranking these values in ascending or descending order. For safety, the client may choose secretly whether the ranking follows the actual or the reverse order. The actual MAX or MIN has to be reversed on the client accordingly. The cloud may then process the value expression entirely in the cloud. The approach extends to the TOP K function (also called LIMIT).

Observe finally that for a value expression involved in a selection predicate, as discussed in the motivating example, the client may also need to rewrite a clause before sending it out. Consider, e.g., the predicate qty * price = 123. To process it entirely on the cloud, 123 needs to be sent encrypted. Likewise for the predicate qty * price > 123. For the latter, the Rank function is besides necessary. If the order given by Rank is the reversed one, the inequality has to change the side as well.

2.7 SFT Creation, Update and Storage Structure

Obviously, the client must create any SFT and send it to the cloud while the cloud database is being created. As we mentioned, it may be useful to create more than one SFT, e.g., a dedicated one for date related scalar functions. It is an open question how much time the creation process may take in practice at present. As the motivating example shows, the number of values in a typical SFT should be under or in a few millions. A ciphertext uses also more storage, as already mentioned. The storage for a ciphertext in a set of such values may be nevertheless apparently greatly reduced, up to 1/10, [G], [T3]. With or without this optimization, the transfer time of the SFT from the client to the server should be practical. The encryption time may be the longest part of the total. Related values remain a future work.

The SFT (initial) encryption time is linear with the number of values. If it appears excessive, one possibility is to create the SFT with a few functions only, e.g., these above listed and the expected practical minimum of rows. Later on, the client may add attributes (functions). Also, the client may add rows, e.g., to decrease the increment here and there.

As indicated already, we see SFT conceptually as a relational table. All tools for the corresponding storage structures are therefore potentially applicable. Also as mentioned in Section 2.2, we expect SFT to be often cached in RAM or a flash of a cloud database node. A row –based file hashed on A as the primary key seems 1st choice. We also need fast key-based access to most used attributes, F1 especially. Thus, one or more also hash-based inverted files appear useful. If an SFT should be disk-based, a columnar structure seems better choice. Finally, if the database is over several nodes, one may expect SFT at least partly replicated. The replicated part can involve the most used rows or columns. Related details are an open issue at present.

2.8 Safety Model

Our basic safety model is that the cloud database intruder may read the cloud data, but cannot intrude a client node. In particular, the encryption key is (*g*, *n*, *r*) we recall, is private to the clients. As we already mentioned, all the relation and attribute names at the cloud are also supposed encoded by the client so to appear meaningless. Consequently we consider that the query rewrite basically takes place at the client. Finally, we consider that the cloud intruder does not have access to queries and, especially, query log if there is any at the cloud.

A more relaxed safety model may allow the query rewrite at the cloud. This obviously speeds up the client somehow and, probably marginally, the query messaging. Consequently the names of the SFT attributes must be meaningful at the cloud. We then have in counterpart many encrypted values and a known relationship among, e.g. A and A' or A and log A. Rank function also reveals then some information about A. We do not see at present reasons why this model could nevertheless compromise a private key. Likewise, our assumption that (*g*,*n*) is private, unlike in the basic scheme, is conservative as well. The rationale is that, first, it does not make sense to have it public while the only *r* to use, unlike in the original scheme, remains private. But if we made *r* public as well, while restricting it to the same *r* for everyone, the scheme becomes trivially unsafe. It seems possible however to have public such (*g*,*n*,*r*), provided the modification of the original encryption/decryption formulae as in [B]. The result is claimed safe in some sense. Our scheme can be perhaps relaxed consequently. The deeper analysis of all relaxed models remains future goal.

Since SFT basically uses the same *r* for every value within, our encryptions, as well as that in [B], are deterministic. It is a common knowledge that an attribute in a table may then expose uneven distribution of stored values, e.g., of actual prices, if several items are equally priced. A relaxed safety model may consider that the intruder may learn the name of such attribute and disclose some, perhaps secret, info from the spikes. This problem is outside our model. One may nevertheless relax the SFT structure accordingly. This, through multiple ciphertexts using different *r*'s for the same plaintext. The scheme becomes pseudo-probabilistic accordingly. Details of such a variant remain a future work.

3 Performance Analysis

The rewrite rules require each only a few lookups and encrypted additions. This the time for the related operations, i.e., *, /, -,^ should be only a few times longer than for +. Hence it should remain practical, as long as the time for + is considered as such. At present, we could not find however the information on the actual timing of + operation.

A crude evaluation of the SFT row size shows 500B, assuming only A, log, minus and Rank functions, with key-based inversion on F1. Hence, 1GB of storage suffices for 2M rows. This may be, e.g., price range from 0.00 to 20,000.00. Thus RAM cache should suffice for many, maybe most of apps. 1T of storage suffices for 200M rows with 3-digit precision, i.e., the range of values from 0,000 to 200M. Summing up, the storage for SFT does not appear a practical limitation.

4 Conclusion

SFTs allow for SQL value expressions over a cloud database using an additive homomorphic encryption only. The addition, the selected tabulated values and the rewrite rules provide for the arithmetical operations other than the addition, as if the encryption was fully homomorphic. Rapid SFT design and performance analysis does not show any major hurdle. Major benefit is that the query processing, until the final decryption, may take place in the cloud. This result does not seem attained by existing prototypes. While many details we pointed to remain future work, SFTs appear already promising.

More precisely, the future work should first analyze more in depth these details. Next, an actual experimentation appears a necessity. As for CryptoDB, this seems that only way to precisely evaluate the actual processing speed.

Acknowledgements

Discussions with Ken Smith at 2013 Cloud Security Meeting at CSIS (GMU), after the presentation [S3] were at the basis of this work.

References

[B] A. Boldyreva, S. Fehr and A. O'Neill. On Notions of Security for Deterministic Encryption and Efficient Constructions without Random Oracles. Crypto 08. Lecture Notes in Comp. Science, Vol. 5157, 335-359, D. Wagner ed., 2008.

[D4] De Capitani, S., Foresti, S., Samarati, P. Selective and Fine-Grained Access to Data in the Cloud. In: Secure Cloud Computing Jajodia, S. & al eds., Springer, 2014.

[E] Encounter software library. http://plaintext.crypto.lo.gy/article/658/encounter.

[G] Ge Tingjian, Zdonik, S., B. Answering aggregation queries in a secure system model. VLDB 07.

[P9] Paillier, P. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. EUROCRYPT 1999, 223-238.

[P] Pailler Cryptosystem http://en.wikipedia.org/wiki/Paillier_cryptosystem

[S3] Smith, K., Allen, D., Sillers, A., Lan, H., Kini, A. How Practical Is Computable Encryption? http://csis.gmu.edu/albanese/events/march-2013-cloud-security-meeting/04-Ken-Smith.pdf

[S4] Smith, K., Allen, M., D., Lan H., and Sillers, A. Making Query Execution Over Encrypted Data Practical. . In: Secure Cloud Computing Jajodia, S. & al eds., Springer, 2014.

[T] Thep. The homomorphic Encryption Project. https://code.google.com/p/thep/

[T3] Tu, S., Kaashoek, M. F., Madden, S., Zeldovich, N. Processing Analytical Queries over Encrypted Data. VLDB 13.

[W] Mathematical Tables. http://en.wikipedia.org/wiki/Mathematical_table