Efficient Execution of Aggregation Queries over Encrypted Relational Databases

Hakan Hacıgümüş[1], Bala Iyer[2], and Sharad Mehrotra[3]

[1] IBM Almaden Research Center, USA, hakanh@acm.org
[2] IBM Silicon Valley Lab., USA, balaiyer@us.ibm.com
[3] University of California, Irvine, USA, sharad@ics.uci.edu

Abstract. Encryption is a common method to assure privacy of stored data. In many practical situations, decrypting data before applying logic compromises privacy. The challenge is to come up with logic transformation techniques and result-mapping methods so that the exact result of applying logic to data-in-the-clear is obtained by applying the transformed logic to encrypted data and mapping the result produced. In the scope of relational aggregation queries and in the presence of logical predicates, we show how to support needed transformations and mappings.

1 Introduction

Increasingly, large companies are outsourcing their IT department and sometimes their entire data center [2]. Privacy of the data stored in these *service* environments is a growing concern. We will give a working definition for the *database-as-a -service* (DAS) model [7,5], and focus on the certain aspects of the privacy challenge in the model. Specifically, we explore techniques to support aggregation in relational databases on encrypted data without decryption in the presence of logical predicates.

In order to explain our methods in a limited space, we use the following simple but general query form:

```
SELECT <grouping attributes>, <aggregation function>
 FROM <relations> WHERE <predicates> GROUP BY <grouping attributes>
```

`<aggregation function>` refers to an SQL aggregation function (`SUM`, `COUNT`, `AVG`, `MIN`, `MAX`) with an arithmetic expression as the parameter. `<predicates>` may include logical comparisons.

Our techniques exploit a specialized encryption method, privacy homomorphism (PH for short), that allows basic arithmetic ($+,-,\times$) over encrypted data. The primary contributions of this paper include: **(1)** The first application of PH to aggregation queries in relational databases including extensions to make it applicable, **(2)** Formal techniques to transform SQL aggregation queries to execute over encrypted tables, and **(3)** performance studies based on a real queries against a real database to validate the ideas.

The DAS model: The DAS model [7][5], is an instantiation of the computing model involving trusted clients, who store their data at an untrusted server that are administrated by the service provider. The challenge is to make it impossible for the system provider to correctly interpret the data. The data is owned by clients. The clients only have limited computational power and storage, and they rely on the server for the mass computational power and storage. The server exposes mechanisms for the clients to create and manage the client databases at the server.

Data originates from the client. We propose that authorized clients be given needed encryption key(s). The data is encrypted by the client before it is sent to the server for inclusion in a table. Data is always encrypted when it is stored on, or processed by the server. At no time is the encryption key given to any administrator, thus data cannot be decrypted by the server. Queries against data-in-the-clear, originate from the client. Algorithms, based on the metadata known to the client, decompose the query into client and server queries. The server query is sent to the server to be executed against encrypted data. Processing algorithms are designed such that the results of the original query are obtained if the client decrypts and further processes the answers of the server query using the decomposed client query.

Furthermore, data privacy is assured under the conditions that the client does not share the encryption keys, the metadata or the unencrypted data with any party who might be an adversary and the server is considered as an adversary.

2 Aggregation over Encrypted Data

2.1 Background on Privacy Homomorphisms

Definition of PH: Assume $\mathcal{A}$ is the domain of unencrypted values, $\mathcal{E}_k$ an encryption function using key k, and $\mathcal{D}_k$ the corresponding decryption function, i.e., $\forall a \in \mathcal{A}$, $\mathcal{D}_k(\mathcal{E}_k(a)) = a$. Let $\tilde{\alpha} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ and $\tilde{\beta} = \{\beta_1, \beta_2, \ldots, \beta_n\}$ be two (related) function families. The functions in $\tilde{\alpha}$ are defined on the domain $\mathcal{A}$ and the functions on $\tilde{\beta}$ are defined on the domain of encrypted values of $\mathcal{A}$. $(\mathcal{E}_k, \mathcal{D}_k, \tilde{\alpha}, \tilde{\beta})$ is defined as a privacy homomorphism if $\mathcal{D}_k(\beta_i(\mathcal{E}_k(a_1), \mathcal{E}_k(a_2), \ldots, \mathcal{E}_k(a_m))) = \alpha_i(a_1, a_2, \ldots, a_m) : 1 \leqslant i \leqslant n$. Informally, $(\mathcal{E}_k, \mathcal{D}_k, \tilde{\alpha}, \tilde{\beta})$ is a privacy homomorphism on domain $\mathcal{A}$, if the result of the application of function α_i on values may be obtained by decrypting the result of β_i applied to the encrypted form of the same values.

Given the above general definition of PH, we next describe a specific homomorphism proposed in [8] that we will use in the remainder of the paper. We illustrate how the PH can be used to compute basic arithmetic operators through an example.

• The key, $k = (p, q)$, where p and q are prime numbers, is chosen by the client who owns the data.
• $n = p \cdot q$, p and q are needed for encryption/decryption and are hidden from the server. n is revealed to the server. The difficulty of factorization forms the basis of encryption.
• $\mathcal{E}_k(a) = (a \bmod p, a \bmod q)$, where $a \in \mathbb{Z}_n$. We will refer to these two components as the p *component* and q *component*, respectively.

• $\mathcal{D}_k(d_1, d_2) = d_1 qq^{-1} + d_2 pp^{-1} \pmod{n}$, where $d_1 = a \pmod{p}, d_2 = a \pmod{q}$, and q^{-1} is such that $qq^{-1} = 1 \pmod{p}$ and p^{-1} is such that $pp^{-1} = 1 \pmod{q}$. **(1)**
• $\tilde{\alpha} = \{+_n, -_n, \times_n\}$, that is addition, subtraction, and multiplication in mod n.
• $\tilde{\beta} = \{+, -, \times\}$, where operations are performed componentwise.

Example: Let $p = 5, q = 7$. Hence, $n = pq = 35, k = (5, 7)$. Assume that the client wants to **add** a_1 and a_2, where $a_1 = 5, a_2 = 6$. $\mathcal{E}(a_1) = (0, 5), \mathcal{E}(a_2) = (1, 6)$ (previously computed) are stored on the server. The server is instructed to compute $\mathcal{E}(a_1) + \mathcal{E}(a_2)$ componentwise (i.e., without decrypting the data). The computation $\mathcal{E}(a_1)+\mathcal{E}(a_2) = (0+1, 5+6) = (1, 11)$. The result, $(1, 11)$ is returned to the client. The client decrypts $(1, 11)$ using the function $(d_1 qq^{-1} + d_2 pp^{-1}) \pmod{n} = (1\cdot 7\cdot 3+11\cdot 5\cdot 3) \pmod{35} = 186 \pmod{35}$, which evaluates to 11, the sum of 5 and 6.[1] The scheme extends to **multiplication** and **subtraction**.

2.2 Extensions to PH

The basic PH scheme above works for modular addition, subtraction, and multiplication of integers only, and needs to be extended in several directions for it to be useful in SQL query processing. SQL arithmetic requires arbitrary expression evaluation using non-modular addition, subtraction, multiplication, and *division* functions on *signed* integers and *floating point* data types. We discuss some of these extensions below (full discussion can be found in [6]).

Negative numbers can be dealt by offsetting the range of numbers. To see the need for this, recall that arithmetic is defined on modulo n in PH. For example, the numbers 33 and -2 are indistinguishable when we represent them in modulo 35. That is, 33 (mod 35) ≡ -2 (mod 35). Let v_{min} be the smallest negative and v_{max} the largest positive number representable on a machine. We map the range of numbers $[v_{min}, v_{max}]$ to a new range $[0, (v_{max} - v_{min})]$, n is chosen to be greater than $v_{max} - v_{min}$. A number x is mapped to a shifted value x' in the new range as $x' = x - v_{min}$. After decryption, the client corrects the answer and maps, in a straightforward manner, back to the values in the original domain.

Preventing Test for Equality: Picking n such that $n > v_{max} - v_{min}$ (as we did above) enables the server to test for equality. Say x and y are two numbers encrypted as (x_p, x_q) and (y_p, y_q), respectively. Let $z = x * y$, which implies in the encrypted domain, $(z_p, z_q) = (x_p, x_q) * (y_p, y_q)$. The server could start adding (x_p, x_q) to itself every time checking the equality between the sum and (z_p, z_q). When the equality is satisfied, the server learns the unencrypted value of y. Thus, $(z_p, z_q) = \underbrace{(x_p, x_q) + (x_p, x_q) + \ldots + (x_p, x_q)}_{y\ times}$.

We plug this exposure by adding random noise to the encrypted value. We encrypt an original value x as follows; $\mathcal{E}(x) = (x \pmod{p} + R(x) \cdot p, x \pmod{q} + R(x) \cdot q)$, where $R(x)$ is a pseudorandom number generator with seed x. $R(x)$ value is generated during every insertion/update by the client. This prevents equality testing for the server and the server cannot remove the noise

[1] n is selected in such a way that results always fall in $[0, n)$; for this example $[0, 35)$.

without knowing p and q. The noise is automatically removed at the client upon decryption. In the presence of noise, the following decryption function should be used in place of equation (1): $\mathcal{D}_k(d_1, d_2) = (d_1 \bmod p)qq^{-1} + (d_2 \bmod q)pp^{-1} \pmod{n}$ This equation is true because noise had been added in multiples of p for the first and in multiples of q in the second term. The modulo of each (mod p) and (mod q) term removes the added noise.

Another benefit of introducing the noise is that p and q components are no longer stored in modulo p and q, respectively. It makes it additionally difficult for the server to guess their values.

3 Selecting Tuples over Encrypted Data

3.1 Aggregation Queries without Predicates

Consider an aggregation query that computes the total compensation of employees: that is `SUM(salary + commission)` from an `employee` relation. Let $employee^S$ be the encrypted server side representation of the `employee` relation. The relation is encrypted at the row level by treating a record as a bit string that is encrypted as a unit. $employee^S$, besides storing the resulting ciphertext as a special field, also contains fields $salary_p^h$ and $salary_q^h$ that store the values *salary* (mod p), and *salary* (mod q); i.e., together the two fields encode $\mathcal{E}^{PH}(salary)$, where $\mathcal{E}^{PH}$ is a PH used to encrypt *salary*. Similarly, $commission_p^h$ and $commission_q^h$ fields represent *commission* using the PH strategy. The original query can be evaluated by computing the aggregation componentwise at the server using the following query:

SELECT SUM(salary$_p^h$+ commission$_p^h$) as s1, SUM(salary$_q^h$+ commission$_q^h$) as s2
FROM employee^S

The client can decrypt the result by computing: $s1 \bmod p * q * q^{-1} + s2 \bmod q * p * p^{-1} \pmod{n}$.

3.2 Handling Logical Comparisons

To support logical comparisons over encrypted data, we differentiate between equality and inequality operators. Consider an attribute A_i on which equality test needs to be performed (e.g., as part of a equi-join or selection operation). If we encrypt the attribute value using a deterministic encryption algorithm, such as AES [1], and store the encrypted field value at the server, equality can be directly tested since $\forall$ domain values $v_i, v_j, v_i = v_j \Leftrightarrow \mathcal{E}_k(v_i) = \mathcal{E}_k(v_j)$, where $\mathcal{E}_k$ is a deterministic encryption algorithm with key k.

For inequality comparisons we utilize the strategy proposed in [5]. Consider a relation `employee (eid,ename salary,city,did)` an instance of which is shown in Table 1. Suppose we wish to retrieve `eid` of employees who make more than $60K$. To evaluate conditions such as $salary > \$60K$, a *coarse index* is stored at the server. Such an index is derived by first partitioning the domain of salary into a set of partitions (or buckets) over the domain of salary (assumed

Table 1. Relation *employee*

eid	ename	salary	city	did
23	Tom	70K	Maple	10
860	Mary	60K	Maple	55
320	John	23K	River	35
875	Jerry	45K	Maple	58
870	John	50K	Maple	10
200	Sarah	55K	River	10

Table 2. Partitions

employee.salary	
Partitions	**ID**
[0,25K]	59
(25K,50K]	49
(50K,75K]	81
(75K,100K]	7

Table 3. Relation $employee^S$: encrypted version of relation *employee*

							$salary^h$	
etuple(encrypted tuple)	eid^{id}	$salary^{id}$	$city^{id}$	did^{id}	$city^f$	did^f	$salary^h_p$	$salary^h_q$
=*?Ew@R*((¡¡=+,-...	2	81	18	2	?Ew...	@R*...	7	27
b*((¡¡(*?Ew@=l,r...	4	81	18	3	?Ew...	=+,...	18	17
w@=W*((¡¡(*?E:,j...	7	59	22	4	¡(*...	¡(*...	2	23
¡(* @=W*((¡?E;,r...	4	49	18	3	?Ew...	E:,...	3	2
(¡(@=U(¡S?/,6...	4	49	18	2	?Ew...	@R*...	8	7
ffTi* @=U(¡?G+,a...	7	49	22	2	¡(*...	@R*...	13	12

to be between $[0, 100K]$ below). For example, $partition(employee.salary) = \{[0K, 25K], (25K, 50K], (50K, 75K], (75K, 100K]\}$. Associated with each partition is its identity determined by an *identification* function *ident* that could be derived using, for example, a one-way hashing technique. A particular assignment of identifiers to 4 salary partitions is shown in Table 2. For instance, $ident_{employee.salary}([0, 25K]) = 59$. A value in the domain can be mapped using the partitioning to its corresponding partition. For example, the salary of `Tom` in the above table maps to partition 81; that is, $Map_{employee.eid}(70K) = 81$. This mapping is used as a *coarse* index at the server in order to support comparison operators over the encrypted data. For example, to test if a tuple satisfies the condition $salary > 60K$, we can test the condition `salary`^{id} `= 81 OR salary`^{id} `= 7` at the server. If the tuple satisfies the condition, its encrypted representation is returned to the client that can decrypt the results to filter out false positives.

4 Query Processing over Encrypted Data

Having developed basic methods to compute aggregations and compare values, we now turn our attention to techniques to evaluate SQL queries over encrypted data. We begin by first formally specifying how relational data is stored at the server. We will then discuss the techniques to map a query into the server side representation.

4.1 Storage Model

Let R be a relation with the set of attributes $\tilde{R} = \{r_1, \ldots, r_n\}$. R is represented at the server as an encrypted relation R^S that contains an attribute

$etuple = \langle \mathcal{E}^t(r_1, r_2, \ldots, r_n) \rangle$, where $\mathcal{E}^t$ is the function used to encrypt a row of the relation R. R^S also (optionally) stores other attributes based on the following classification of the attributes of R:

- *Aggregation attributes* ($A_j \in \tilde{R} : 1 \leqslant j \leqslant j' \leqslant n$): are attributes of R on which we expect to do aggregation. For each A_j, R^S contains an attribute A_j^h that represents the encrypted form of corresponding original attribute A_j with PH, thus $A_j^h = \mathcal{E}^{PH}(A_j)$, where $\mathcal{E}^{PH}$ is a PH.
- *Field-level encrypted attributes* ($F_k \in \tilde{R} : 1 \leqslant k \leqslant k' \leqslant n$): are attributes in R on which equality selections, equijoins, and grouping might be performed. For each F_t, R^S contains an attribute $F_k^f = \mathcal{E}^f(F_k^f)$, where $\mathcal{E}^f$ is a deterministic encryption used to encode the value of the field F_k.
- *Partitioning attributes* ($P_m \in \tilde{R} : 1 \leqslant m \leqslant m' \leqslant n$): are attributes of R on which general selections/joins might be performed. For each P_m, R^S contains an attribute P_m^{id} that stores the partition index of the base attribute values, thus $P_m^{id} = Map_{R.P_m}(P_m)$.
- *Embedded attributes* ($E_\ell \in \tilde{R} : 1 \leqslant \ell \leqslant \ell' \leqslant n$): are attributes in $\tilde{R}$ that are not in any of the above three categories. These attributes are, most likely, not accessed individually by queries for either selections, group creation, or aggregation. They need not be encrypted separately. Their values can be recovered after the decryption operation on the encrypted row (i.e., *etuple*) is executed on the client site.

Given the above attribute classification, the schema for the relation R^S is as follows:

$$R^S(etuple, P_1^{id}, \ldots, P_{m'}^{id}, F_1^f, \ldots, F_{k'}^f, A_1^h, \ldots, A_{j'}^h)$$

Table 3 shows a possible instance of the server side representation of the the *employee* relation given in Table 1. In the mapping, we assumed that partitioning attributes are $\{eid, salary, city, did\}$, field-level encrypted attributes are $\{city, did\}$, and aggregation attributes are $\{salary\}$. Note that for a relation, the categories may overlap.

4.2 Approach Overview

Given a query Q, our problem is to decompose the query to an appropriate query Q^S on the encrypted relations R^S such that results of Q^S can be filtered at the client in order to compute the results of Q. Ideally, we would like Q^S to perform bulk of the work of processing Q. The effectiveness of the decomposition depends upon the specifics of the conditions involved in Q and on the server side representation R^S of the relations involved. Consider, for example, a query to retrieve sum of salaries of employee in $did = 40$. If *did* is a field-level encrypted field, the server can exactly identify records that satisfy the condition by utilizing the equality between the client-supplied values and the encrypted values stored on the server. In such a case, aggregation can be fully performed on the *salary* attribute of the selected tuples exploiting the PH representation. If, on the other hand, the condition were more complex, (e.g., `did > 35 AND did < 40`), such a query will be mapped to the server side by mapping the *did* to the corresponding partitions associated with the *did* field that cover the range of values from 35 to 40. Since the tuples satisfying the server side query may be a superset of the actual answer, aggregation cannot be completely performed at the server. Our strategy is to separate the qualified records into those that certainly satisfy the query conditions, and those that may satisfy it - the former can be aggregated at the server, while the latter will need to be transmitted to the client, which on decrypting, can filter out those that do not, and aggregate the rest. The strategy suggests a natural partitioning of the server side query Q^S into two queries Q_c^S and Q_m^S as follows:

• **Certain Query (Q_c^S):** that selects tuples that *certainly* qualify the conditions associated with Q. Results of Q_c^S can be aggregated at the server.

• **Maybe Query (Q_m^S):** that selects *etuple*s corresponding to records that *may* qualify the conditions of Q but it cannot be determined for sure without decrypting. The client decrypts these *etuple*s, and then selects the ones that actually qualify and performs the rest of the query processing.

To finalize the computation, the client combines results from these queries to reach the actual answers. We next discuss how a client side query Q is translated into the two server side representations Q_c^S and Q_m^S.

4.3 Mapping Conditions

The principal issue in decomposing the query Q into its server side representations Q_c^S and Q_m^S is to map the conditions specified in Q to corresponding conditions on the server side representation. We first consider how an individual condition C_k of Q is mapped. Our mapping function $Map_{cond}(C_k)$ consists of two components: $Map_{cond}^c(C_k)$ and $Map_{cond}^m(C_k)$. Map_{cond}^c maps C_k to a server side condition such that every tuple satisfying $Map_{cond}^c(C_k)$ *certainly* satisfies C_k, while Map_{cond}^m maps C_k into a condition that qualifies tuples that *maybe* satisfies C_k. Together, the two conditions identify (a superset of) tuples that satisfy the original condition C_k. Naturally, $Map_{cond}(C_k) = Map_{cond}^c(C_k) \vee Map_{cond}^m(C_k)$. We will use the following notation to describe how the conditions in the original query are mapped to their server side representations.

Let R be a relation, $R.A_i$ be a partitioning attribute of R, let $\{p_1, p_2, \ldots, p_n\}$ be the set of partitions associated with with $R.A_i$, and v be a value in the domain of $R.A_i$. We define the following mapping functions on the partitions associated with A_i: $Map_{R.A_i}^{>}(v) = \{ident_{R.A_i}(p_k) \mid p_k.high \geqslant v\}$, and $Map_{R.A_i}^{<}(v) = \{ident_{R.A_i}(p_k) \mid p_k.low \leqslant v\}$, where $p_k.low$ and $p_k.high$ are the lower and the upper boundary of the partition p_k, respectively.

Attribute = Value: We can evaluate the condition by testing equality between the field level encrypted values of the attribute A_i and the value v given in the condition, i.e., $A_i^f = \mathcal{E}_k(v)$. The result is exactly the set of records that satisfy the condition since, for a deterministic encryption, $A_i^f = \mathcal{E}_k(A_i)$ and $A_i = v \Leftrightarrow \mathcal{E}_k(A_i) = \mathcal{E}_k(v)$ Thus,

$$Map_{cond}(A_i = v) \equiv Map_{cond}^c(A_i = v) \equiv A_i^f = \mathcal{E}_k(v)$$

Attribute $<$ Value: We utilize partitioning attributes to map the condition. Since the query condition may fully contain some of the partitions and partially overlap with the others, Map_{cond} function will have both Map^c_{cond} and Map^m_{cond} components.

$$Map_{cond}(A_i < v) \equiv Map^c_{cond}(A_i < v) \vee Map^m_{cond}(A_i < v)$$

$$Map^c_{cond}(A_i < v) \equiv \bigvee_{p_j \in P_{C_k} \wedge p_j.high < v} A_i^{id} = ident(p_j)$$

$$Map^m_{cond}(A_i < v) \equiv \bigvee_{\substack{p_\ell \in P_{C_k} \wedge p_\ell.low \leqslant v \\ \wedge p_\ell.high \geqslant v}} A_i^{id} = ident(p_\ell)$$

where $P_{C_k} = \{p_t \mid p_t \in Map^{<}_{A_i}(v)\}$

Attribute1 $=$ Attribute2: For this case, we again exploit the field level encrypted attributes of an encrypted relation. We can test the equality of two attribute values directly over their encrypted values, as $A_i = A_j \Leftrightarrow \mathcal{E}_k(A_i) = \mathcal{E}_k(A_j)$ due to deterministic encryption.[2] As a result, Map_{cond} includes only Map^c_{cond} component for this type of condition. Thus,

$$Map_{cond}(A_i = A_j) \equiv Map^c_{cond}(A_i = A_j) \equiv A_i^f = A_j^f$$

Attribute1 $<$ Attribute2: To evaluate the condition, we need to test the order of the values of two attributes mentioned in the condition. Since the encryption algorithms, which may be used for field level encrypted attributes and for aggregation attributes do not preserve the order of the original data, they may not be used for the test. Therefore, we use partitioning attributes to evaluate the condition.

The condition is mapped by considering all pairs of partitions of A_i and A_j that could satisfy the condition. The pairs that have overlap (either fully or partially) are subject to Map^m_{cond} function. The other pairs, which do not overlap while satisfying the condition, are subject to Map^c_{cond} function. Formally the mapping is defined as follows:

$$Map_{cond}(A_i < A_j) \equiv Map^c_{cond}(A_i < A_j) \vee Map^m_{cond}(A_i < A_j)$$

$$Map^c_{cond}(A_i < A_j) \equiv \bigvee_{\phi} \big(A_i^{id} = ident_{A_i}(p_m) \wedge A_j^{id} = ident_{A_j}(p_n)\big)$$

$$Map^m_{cond}(A_i < A_j) \equiv \bigvee_{\varphi} \big(A_i^{id} = ident_{A_i}(p_k) \wedge A_j^{id} = ident_{A_j}(p_\ell)\big)$$

where ϕ is $p_m \in partition(A_i)$, $p_n \in partition(A_j)$, $p_n.low > p_m.high$ and φ is $p_k \in partition(A_i)$, $p_\ell \in partition(A_j)$, $p_\ell.low \leqslant p_k.high$.

[2] We make an assumption that the same key is used to encrypt the attributes A_i and A_j

```
Input: Composite condition $W$ of original query $Q$
 1 for each atomic condition $C_i$ in $W$
     Compute mapped condition $C_i'$ : $Map_{cond}(C_i) \equiv Map^c_{cond}(C_i) \vee Map^m_{cond}(C_i)$
 2 Build mapped composite condition $W'$ with $C_i$'s
 3 Convert $W'$ into DNF
 4 Define $D^c$ as set of disjuncts having only $Map^c_{cond}$
 5 Define $D^m$ as set of disjuncts having $Map^m_{cond}$
 6 Form query $Q^S_c$ with $D^c$ in WHERE clause
 7 Form query $Q^S_m$ with $D^m$ in WHERE clause
```

Fig. 1. Condition mapping algorithm

4.4 Query Decomposition

Once all conditions are translated according to the mappings given above, we need to identify the parts of the conditions that will be evaluated by Q^S_c and Q^S_m. To separate the conditions, we first map each condition by using the mapping functions given above. Then, we convert the resulting conditions into disjunctive normal form (DNF) and split the disjuncts into two classes:

• **Certain disjuncts:** These are the disjuncts that contain *only* Map^c_{cond} functions. Note that tuples satisfying these disjuncts *certainly* satisfy the conditions associated of the original query Q.

• **Maybe disjuncts:** These disjuncts contain Map^m_{cond} functions (they may also contain Map^c_{cond} functions). Tuples that satisfy these disjuncts may or may not satisfy the conditions specified with the original query Q.

The above classification suggests a natural splitting of the server side query into two parts: Q^S_c and Q^S_m. Q^S_c is formed with the *certain disjuncts*, which only contain Map^c_{cond} functions and Q^S_m is formed with the *maybe disjuncts*, which contain Map^m_{cond} functions. Algorithmic steps of this procedure is given in Figure 1.

For Q^S_c, the `GROUP BY` attributes in the `SELECT` clause are replaced by their field-level encrypted attributes and the aggregation is replaced with the corresponding aggregation over the PH representation of the attribute. The result of Q^S_c will be the encrypted representation of the group value along with the PH encrypted value of the corresponding aggregation. The client can decrypt the group values and the encrypted aggregations. For Q^S_m, the `SELECT` clause is replaced by the selection of the *etuple*s that will be sent to the client. The client will need to decrypt the *etuple*s to determine those that satisfy the conditions associated with query Q. Subsequently, the client will perform the corresponding `GROUP BY` and aggregations.

The client can determine the final result of the aggregation query by merging the results of the individual computations of the two queries. The mechanism to merge the results of the two queries depends upon the specific aggregation function associated with the query.

4.5 Handling Other Aggregation Functions

We note that `COUNT`, by itself, does not involve arithmetic and hence does not pose any additional difficulty due to PH. `AVG` function can be implemented as `SUM / COUNT`.

The `MIN` and `MAX` functions cannot be directly supported using PH. The PH does not preserve the order of the original data. It is already established in [8, 3,4] that if a PH preserves order then it is insecure. Hence, we have devised different mechanisms to compute minimum and maximum values as follows.

We note that, the domain partitioning strategy can also be used to support the `MIN` and `MAX` functions. Since the order of the partitions is known to the client, the client can exactly identify and request the partition(s) that may contain the minimum and maximum values. After receiving the *etuple*s in the requisite partitions, the client can decrypt and find the exact values of `MIN` and `MAX` function by only evaluating within those partitions.

4.6 Example

In this section, we explain our strategy by walking through the steps of the query translation discussed above using an example query over the *employee* and *manager* tables. Sample population of *employee* table is given in Table 1 and partitioning scheme of *salary* attribute of *employee* is given in Table 2. Consider the following query, which has composite condition W : $city$ =‘$Maple$’$\wedge salary < 65K \wedge emp.did = mgr.did$ consists of three atomic conditions, namely, C_1 : $city$ =‘$Maple$’, C_2 : $salary < 65K$, C_3 : $emp.did = mgr.did$;

```
SELECT SUM(salary) FROM employee, manager
WHERE city=‘Maple’ AND salary < 65K AND emp.did=mgr.did
```

Let us now generate the server side representation of the original query by following the algorithm steps given in Figure 1.

1. We first map each atomic condition by identifying Map^c_{cond} and Map^m_{cond} parts. Hence, the conditions are mapped as follows:
   For C_1: $Map_{cond}(city =' Maple') \Rightarrow Map^c_{cond}(city =' Maple') \Rightarrow city^f = \mathcal{E}('Maple')$
   For C_2: $Map_{cond}(salary < 65K) \Rightarrow Map^c_{cond}(salary < 65K) \vee Map^m_{cond}(salary < 65K)$
   $Map^c_{cond}(salary < 65K) \Rightarrow salary^{id} = 49 \vee salary^{id} = 59$
   $Map^m_{cond}(salary < 65K) \Rightarrow salary^{id} = 81$
   For C_3: $Map_{cond}(emp.did = mgr.did) \Rightarrow Map^c_{cond}(emp.did = mgr.did)$
   $\Rightarrow emp.did^f = mgr.did^f$

2. Thus, mapped composite condition W' is formed as:
   $W' : city^f = \mathcal{E}('Maple') \wedge (salary^{id} = 49 \vee salary^{id} = 59 \vee salary^{id} = 81)$
   $\wedge emp.did^f = mgr.did^f$
3. Now we can convert W' into DNF:

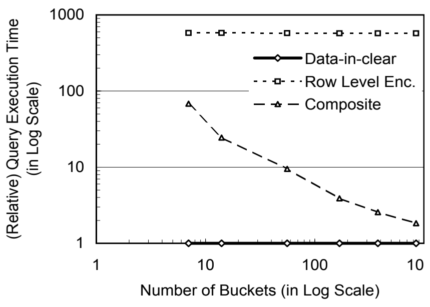
$$\begin{aligned} W': \quad & \underbrace{(city^f = \mathcal{E}('Maple')}_{Map^c_{cond}(C_1)} \wedge \underbrace{salary^{id} = 49}_{Map^c_{cond}(C_2)} \wedge \underbrace{emp.did^f = mgr.did^f)}_{Map^c_{cond}(C_3)} \longrightarrow \mathbf{D_1} \\ \vee\ & \underbrace{(city^f = \mathcal{E}('Maple')}_{Map^c_{cond}(C_1)} \wedge \underbrace{salary^{id} = 59}_{Map^c_{cond}(C_2)} \wedge \underbrace{emp.did^f = mgr.did^f)}_{Map^c_{cond}(C_3)} \longrightarrow \mathbf{D_2} \\ \vee\ & \underbrace{(city^f = \mathcal{E}('Maple')}_{Map^c_{cond}(C_1)} \wedge \underbrace{salary^{id} = 81}_{\underline{Map^m_{cond}(C_2)}} \wedge \underbrace{emp.did^f = mgr.did^f)}_{Map^c_{cond}(C_3)} \longrightarrow \mathbf{D_3} \end{aligned}$$

4. From step 3, the set D^c, only having Map^c_{cond}, is defined as $\{D_1, D_2\}$.
5. The set D^m, having Map^m_{cond}, is defined as $\{D_3\}$.
   Now we can form the server side representation of the original query by forming two queries: Q^S_c and Q^S_m as follows:
6. Q^S_c: `SELECT SUM`^{PH}`(salary`^{h}`) FROM employee`^{S}`,manager`^{S}
   `WHERE city`^{f}`=`$\mathcal{E}$`('Maple') AND (salary`^{id}`=49 OR salary`^{id}`=59)`
   `AND emp.did`^{f}` = mgr.did`^{f}
7. Q^S_m: `SELECT employee`^{S}`.etuple,manager`^{S}`.etuple`
   `FROM employee`^{S}`, manager`^{S}` WHERE city`^{f}`=`$\mathcal{E}$`('Maple') AND salary`^{id}`=81`
   `AND emp.did`^{f}` = mgr.did`^{f}

Q^S_c evaluates and returns the aggregation, `SUM(salary)`, on encrypted relation. Q^S_m selects tuples, which may satisfy the original query condition. In our example, these correspond to the first two tuples of the *employee*^{S} relation (see Table 3). The query returns the corresponding *etuple*s to the client. Upon decryption, the client can figure that, the first tuple (which has $salary = 70K$) does not satisfy the query and should be eliminated. The second tuple, however, which has $salary = 60K$, satisfies the query condition. The client finalizes the computation by combining the answer returned by Q^S_c, and those condition-satisfying tuples returned by the second query, Q^S_m.

4.7 Experimental Evaluation

To evaluate our strategy we used Query #1 from the TPC-H benchmark. The TPC-H tables were appropriately mapped to the server such that the query evaluation could exploit all the mechanisms to compute over encrypted data discussed in earlier sections. Figure 2 compares the performance of TPC Q#1 for three strategies: (1) executing Q#1 directly over original TPC-H data in the clear; (2) executing Q#1 by simply selecting the corresponding *etuples* without any aggregation at the server as presented in [5]. Tuples are decrypted and aggregated at the client; and (3) composite strategy that corresponds to our strategy where aggregation evaluation is pushed to encrypted data as much as possible. The first strategy forms the base line of comparison of results while the other two highlight the advantage of our scheme. Figure 2 plots the relative query response. As expected, strategy 2 performs poorly due to the dominant cost of decryption and increased communication. The composite strategy, however, shows significantly less overhead even with small number of buckets. As the number of buckets increases, lesser number of buckets partially overlap with the query range, hence more aggregation work is done at the server and the



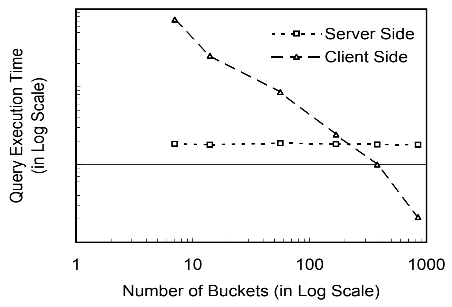


Fig. 2. Comparison of the query processing schemes

Fig. 3. Server side vs. Client side query performance for composite scheme

network traffic as well as decryption cost at the client reduces. Figure 3 shows the client and server components of the query response times. The server side query execution time remains steady with the number of buckets. Client side query, however, significantly benefits from the increasing number of buckets, which suggests less number of *etuple*s returned and decrypted. As a result of declining decryption cost, client side query shows much improved performance.

5 Conclusions

The privacy of information stored in the databases is an issue of increasing importance. In this paper, we have shown how to execute SQL aggregations efficiently over encrypted data, a significant advance in privacy of data subject to SQL processing. To achieve this, we have developed an enhanced encrypted data storage model and introduced formal query implementation techniques to translate original aggregation queries to a form that can directly be executed over the encrypted data.

References

1. AES. Advanced Encryption Standard. *National Institute of Science and Technology, FIPS 197*, 2001.
2. ComputerWorld. J.P. Morgan signs outsourcing deal with IBM. Dec. 30, 2002.
3. J. Domingo-Ferrer. A new privacy homomorphism and applications. *Information Processing Letters*, 6(5):277–282, 1996.
4. J. Domingo-Ferrer. Multi-applications smart cards and encrypted data processing. *Future Generation Computer Systems*, 13:65–74, 1997.
5. H. Hacıgümüş, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over Encrypted Data in Database Service Provider Model. In *Proc. of ACM SIGMOD*, 2002.
6. H. Hacıgümüş. Privacy in Database-as-a-Service Model. *Ph.D. Thesis, Department of Information and Computer Science, University of California, Irvine*, 2003.
7. H. Hacıgümüş, B. Iyer, and S. Mehrotra. Providing Database as a Service. In *Proc. of ICDE*, 2002.
8. R. L. Rivest, L. M. Adleman, and M. Dertouzos. On Data Banks and Privacy Homomorphisms. In *Foundations of Secure Computation*, 1978.