VRS-DB: Preserve Confidentiality of Users' Data using Encryption Approach

Jitendra Rauthan, Kunwar Vaisla
Uttarakhand Technical University Dehradun
India
jsrauthan1@gmail.com
hod_cse@itgopeshwar.ac.in

ABSTRACT: *We tackle stability problems within a cloud data-base frame that uses the Database as a Service display. In this a version, a data proprietor transactions its advice to a cloud data-base pro company. To offer advice stability, fragile data is authenticated from the Data-Owner until it's moved into the Service Provider. Existing security ideas, whatever the situation, are simply largely homomorphic as in all those had been designed to empower one definite type of calculation to be achieved on encrypted data. These recent plans cannot be integrated to respond real usable questions which have surgeries of varied kinds. We advocate and dissect a secure question dealing with frame (VRS-DB) on societal tables along with also an arrangement of basic directors on encoded data that enable information inter-operability, allowing a broad selection of SQL queries to become ready from the Service-Provider on data that is authenticated. We show that our security conspire is protected from 2 forms of threats also that it's basically proficient.*

Keywords: VRS-DB, Encryption, Decryption, Proxy Server, Privacy, Security, Confidentiality, Sensitive Data

Received: 1 May 2018, Revised 29 May 2018, Accepted 18 June 2018

DOI: 10.6025/jio/2018/8/3/98-113

1. Introduction

Developments in the cloud computing, currently towards plenty investigation paintings at technical improvement the cloud database systems that deploy the database-as-a-service version (DBaaS). Industrial cloud database offerings, consisting of Amazon's RDs1 and Microsoft's Square Azure2, are also available. Underneath the DBaaS version, a statistics proprietor uploads the data and information to a carrier company, that hosts superior machines and complex information software system to method queries on behalf of the Data-Owner. The Service-Provider therefore delivers storage, calculation and supervision services. Here various benefits of outsourcing information services, adore extremely ascendable, in addition adaptable calculation for handling buxom assignments. Likewise, with multitentacled cloud information databases will importantly cut back the full value of possession.

To give information security, touchy information ought not be uncovered to the Service-Provider. The basic practice is to scramble the information before putting away it at the Service-Provider. The Service-Provider accordingly gives a solid store

house stockpiling and organization administrations, (for example, reinforcement and recuperation). To process questions, the encoded information ought to be dispatched back to the Data-Owner, which needs to process the touchy information without anyone else. The capable calculation administrations given by the Service-Provider is generally lost.

Keeping in mind the end goal to use the calculation assets of the Service-Provider in question preparing, a couple of secure inquiry handling frameworks, for example, CryptDB [5] and MONOMI [6], have been produced. A shortcoming of these frameworks is that every datum operation (e.g., correlation or expansion) is upheld by a specific encryption conspire. These plans are by and large not information interoperable, i.e., the yield of an administrator can't be utilized as contribution of another in light of the fact that diverse administrators utilize distinctive encryption techniques. In this manner, existing methodologies give restricted local backings to complex questions that include different sorts of administrators.

For example, CryptDB can just help 4 out of 22 TPC-H questions without fundamentally including the Data-Owner or broad precomputation in inquiry handling. Trusted DB [2] and Cipherbase [1] adopt an equipment strategy to give information security. Since particular equipment is required, these methodologies are by and large costlier than programming approaches, which can be actualized utilizing off-the-rack machines. A point by point talk of the above frameworks can be found in [7].

Our framework, VRS-DB, adopts another strategy that depends on the Secure Multiparty Computation (SMC) display [8]. With mystery sharing, each plain esteem is decayed into a few offers and each offer is kept by one of various gatherings. While no gathering can recuperate the plain esteems by its own offers, a convention executed by the gatherings can be characterized to register a deterministic capacity of the qualities. VRS-DB gives an arrangement of basic administrators to help SQL inquiry handling. A recognizing highlight of VRS-DB is that the bolstered administrators all work on mystery offers thus they all work on the same scrambled space. VRS-DB in this manner gives information interoperability, which permits an extensive variety of complex inquiries to be communicated and handled. For instance, all TPC-H inquiries can be locally handled by VRS-DB. In addition, our conventions for executing the different administrators are basically productive.

2. Literature Surveys

There exist many approaches tackling the challenge of data security. One common way is to encrypt data before uploading to the cloud server. For instance, Oracle database 11g provides Transparent Data Encryption to encrypt disk-resident data, as a service of reliable data storage and administration. However, as the data is not preserved after conventional encryption, query processing on encrypted data is impossible and the computational power of cloud service provider is thus lost. To execute queries, Data-Owner has to adopt a Decrypt-Before-Query(DBQ) approach, that is, shipping all ciphertext to Data-Owner, decrypting them before executing queries.

On the other hand, fully homomorphic encryption(FHE)[10] techniques are developed to realize computation on encrypted data. Despite its theoretical significance, such encryption scheme is too computationally expensive for practical use, not to mention data-intensive queries on the cloud. For example, a recent implementation of FHE[2] lets us process 180 AES blocks in around 18 minutes using 3.7GB RAM, whose running time is far from satisfactory for industry use.

Several partially homomorphic encryption schemes are proposed to process particular types of operations on encrypted data, such as OPES[3](which supports homomorphic comparison encryption) and RSA(which supports homomorphic multiplication encryption). However, a simple selection query such as "uni-cost * quantity < budget" cannot be processed on encrypted data by piecing different encryption functions together.

TrustedDB[4] and Cipherbase are hardware-based secure processors, which use secure hardware components to store private keys of sensitive data during processing. However, since security is dependent on the specific hardware, such secure processors cannot scale up as clusters easily and have to be replaced in case of hardware failure, which makes them not economically scalable.

CryptDB[5,11] is a well-known system for processing encrypted data, using an onion encryption approach, implemented upon MySQL. CryptDB uses different levels of encryptions to support different security strength and computational support. For example, it uses RSA to allow equality check on encrypted data, and uses OPES to achieve ordering preserving comparison. Under CryptDB, data security is gradually released to a level to achieve appropriate computation requirement. However, some

operators of CryptDB are not secure against chosen plaintext attack, and it's capable of a limited range of secure operations.

Within this paper, we focus in an algorithmic way to solving the secure database issue. Since our answer doesn't rely on special hardware, so this may be executed with inexpensive off-road machines.

3. Technique Overview

Inside this area we clarify our information version, both the device structure, security grade, and also the inquiry version.

3.1 Encryption Procedure

VRS-DB encrypts sensitive data with a secret sharing method per column. The Data-Owner maintains the secret keys of columns while Service-Provider maintains the encrypted value of sensitive columns, plaintext of non-sensitive columns together with some crypto helper columns.

To encrypt data with secret sharing, the Data-Owner first generate two secret numbers, g and n. n is the product of two big random prime numbers $\rho 1$, $\rho 2$, while g is a positive number that is co-prime with n. We define

$$n = \rho 1 \rho 2$$

$$\phi(n) = (\rho 1-1)(\rho 2-1)$$

Consider a sensitive column $C1$ to be encrypted. Data-Owner generates a pair of column keys $<w, z>$ for column $C1$ randomly. Also, for each row (rw) of column $C1$, Data-Owner generates a distinct row identifier (r-id) randomly. We require that $0 < r`, w, z < n$.

We use $[v]$ to denote a sensitive value to be encrypted, v_e to denote the ciphertext after encryption, v_k to denote the item key of $[v]$.

The item key is generated by

$$v_{k=gen(rw, <w,z>)=wg^{(rw.z \bmod \Phi(n))} \bmod n.}$$

The encrypted value is computed by

$$v_e = \mathcal{E}([v], v_k) = [v], v_k^{-1} \bmod n.$$

To recover plaintext from ciphertext, we compute

$$[v] = D(v_e, v_k) = v_e v_k \bmod n.$$

R-id is encrypted by addictive homomorphic encryption $E()$.

The encryption procedure is illustrated in Figure 2. It shows how sensitive data is transformed into encrypted values. To reveal the plaintext, Data-Owner only needs to store the column keys, while Service- Provider maintains the bulk encrypted data.

3.2 Secure Operators

Data interoperability is one of the most important qualities of VRS-DB. Table 1 shows a list of primitive secure operators implemented in current version of VRS-DB.

Notice that VRS-DB supports data operability in different modes. For instance, the × operator, can be applied in three scenarios: operands are both encrypted (BE Mode); one operand is encrypted, the other is a constant (OE Mode); one operand is encrypted, the other is plain (OEP Mode). Besides r-id, column T (with encrypted values of random number) and column Y (with encrypted values of 1's) help achieve secure operation.

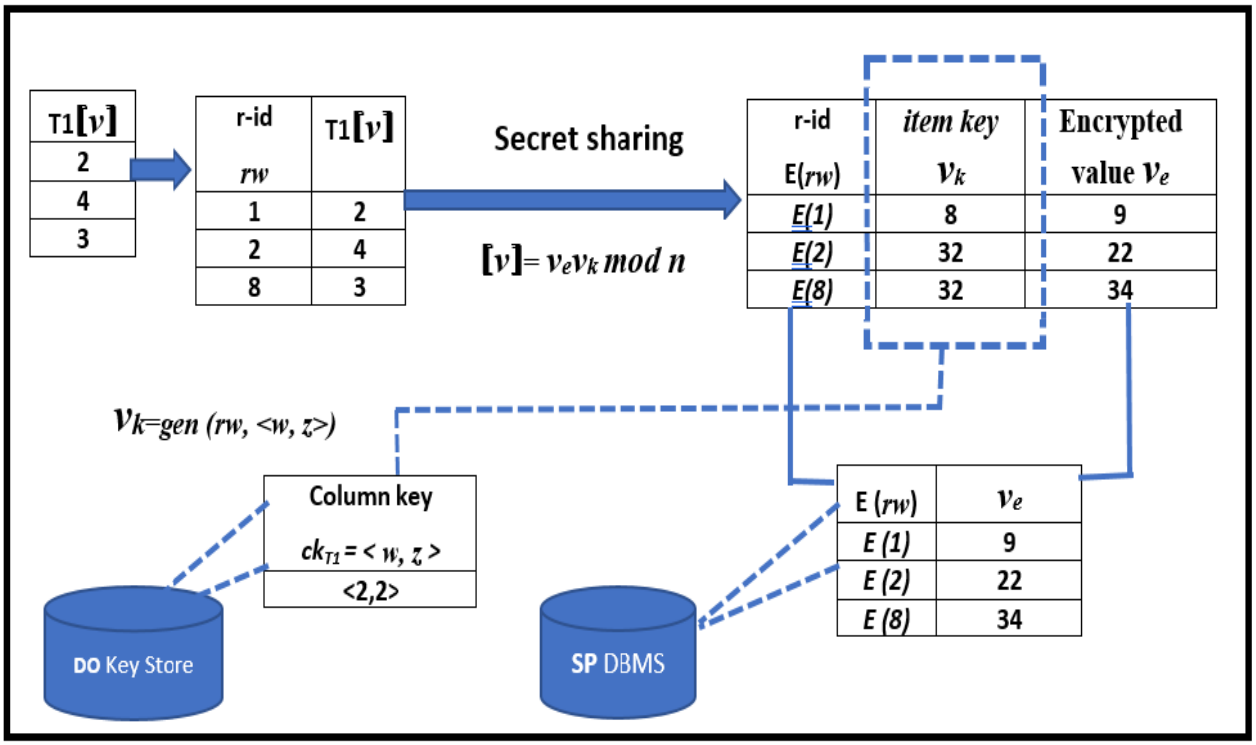


Figure 1. Encryption procedure ($g = 2$, $n = 35$)

S. No.	Operator	Expression	Description
1.	×	*C1 x C2*	Vector dot product of two columns of the same table
2.	+, -	*C1+C2, C1-C2*	Vector addition/subtraction of two col umns of the same table
3.	=	*C1=C2*	Equality comparison of the same table and output a binary column of '0' or '1'
4.	>	*C1>C2*	Ordering comparison on two columns of the same table and output a binary column of '0' or '1'
5.	⊓	⊓ *S(R)*	Project table R on attribute specified in an attribute set S EX-OR
6.	count	*Count(R)*	Count the number of rows in a relation

Table 1. List of Primitive Secure Operators

3.3 [Multiplication] (Algorithm 1 and Algorithm 2)

Apart from the message from Data-Owner telling Service-Provider which columns are the operands of the multiplication, there is no other message. So, multiplication reveals no information.

Inputted Values: Column key <w_{C1}; z_{C2}> and <w_{C2}; z_{C2}> for the columns $C1$ and $C2$ respectively

Computed Result: R's column key <w_R; z_R> for given $R = C1.C2$

Client-Side Operation:

$z_R = z_{C1} + z_{C2} \bmod \Phi(n)$;

$w_R = w_{C1.} w_{C2} \bmod n$;

Server-Side Operation:

So, **for** each row r` **do** the following

Assume that $c1_e$, $c2_e$ be the encrypted values on $C1$, $C2$;

Compute the encrypted values of R by the following, $r_e = c1_{e.} c2_e \ mod \ n$;

End

Algorithm 1. BE multiplication

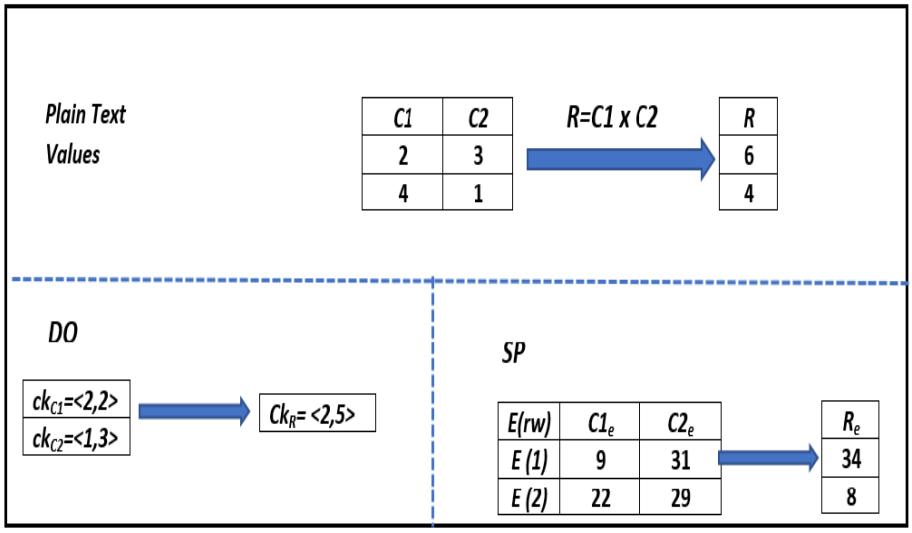


Figure 2. $R = C1 \times C2$ (g =2, n =35)

Inputted Values: Column C1 with column key <$\omega C1$; $zC1$> and a constant x

Computed Result: $R = xC1$ with R's column key <ωR; zR>

Client-Side Operation:

$zR = zC1$;

$\omega R = x\omega C1 \ mod \ n$;

Indicate that R's encrypted column is $C1$`s encrypted column;

Server-Side Operation:

None

Algorithm 2. OE multiplication

3.4 [Key Update] (Algorithm 3)

Key update is a helper operator which takes a column C1 and a target column key $<wC1; zC1>$ and output a column R that shares the same plaintext of $C1$ with target column key, without revealing sensitive information. The following theorem bounds the information revealed in the key update.

Inputted Values: (a) Column $C1$ with column key $<w_{C1}; z_{C1}>$;
(b)target column key $<w_R; z_R>$;
Computed Result: $R = C1$ with R's column key $<w_R; z_R>$;
Client-Side Operation:
Assume that $<w_K; z_K>$ be the column key of K;
$i = z_K^{-1} (z_R - z_{C1}) \ mod \ \Phi(n)$;
$j = w_{C1} z_K^{\ i} w_R^{-1} \ mod \ n$;
Send i, j to SP;
Set R's column key as $<w_R; z_R>$;
Server-Side Operation:
Obtain i, j from DO;
So, **for** each *row r`* **do** the following
Assume that $c1_e$, k_e be the encrypted values on $C1$, K;
Compute the encrypted values of R by the following, $r_e = jc1_e, k_e^{\ i} \ mod \ n$;
End

Algorithm 3. Key update

3.5 [ADDITION] (Algorithm 4)

Inputted Values: Column $C1$, $C2$ with column key $<w_{C1}; z_{C1}>$ and $<w_{C2}; z_{C2}>$
Computed Result: $R = C1 + C2$ with R's column key $rke_R = <w_R; z_R>$
Client-Side Operation:
Produce arbitrary $w_R; z_R$
$C1` = ke(C1; rke_R)$; // *DO* executes client-protocol
$C2` = ke(C2; rke_R)$; // of key update.
Set R's column key as $<w_R; z_R>$;
Server-Side Operation:
$C1` = ke(C1; rke_R)$; // SP executes client-protocol
$C2` = ke(C2; rke_R)$; // of key update.
So, **for** each row r` **do** the following
Assume that $c1`e$, $c2`e$ be the encrypted values on $C1`$, $C2`$;
Compute the encrypted values of R by the following, $re = c1`e.c2`e \ mod \ n$;
End

Algorithm 4. BE addition/subtraction

3.6 [Comparison] (Algorithm 5)

Inputted Values: *Column C1, C2 with column key* $<w_{C1}; z_{C1}>$ *and* $<w_{C2}; z_{C2}>$

Computed Result: *A column of comparison results R: = 0 if*

C1 = C2; = 1 if C1 > C2; return = -1 if C1 < C2

Client-Side Operation:

P = Q(C1- C2); // EE addition, EE multiplication

P' = k(P; <1; 0>); // DO executes client-protocol

Server-Side Operation:

P = Q(C1- C2); // Corresponding server protocol

P' = k(P; <1; 0>); // Corresponding server protocol

So, ***for*** *each row r`* ***do*** *the following*

Assume that $p`_e$ *be the values on P`;*

Switch $p`_e$ *do*

case = 0

$a_e = 0$; // a_e *is the result of this row*

end

case > 0

$a_e = 1$;

end

case < 0

$a_e = -1$;

end

End switch

End

Algorithm 5. Comparison

3.7 [Cartesian Product] (Algorithm 6)

The Cartesian product requires the Service-Provider to do the same computation for each column.

Inputted Values: *Relation* $R_1 1(r\text{-}id, C1_1, C1_2, ..., C1_l, V_1, O_1)$;

Relation $R_2(r\text{-}id, C2_1, C2_2, ..., C2_k, V_2, O_2)$

Computed Result: $R` = R1 \otimes R2$. *R` has a schema* $(r\text{-}id, C1_1`, ...C1_l, C2_1`, ... C2_k, V`, O`)$.

Client-Side Operation:

Assume that $<w_{V2}; z_{V2}>$ *be the column key of* V_2 *in* R_2;

// The column V`and O` comes from R_1

for each column C1 in $(C1_1, C1_2, ..., C1_l, V_1, O_1)$ *from* R_1

do

Assume that $<w_{C1}; z_{C1}>$ *are Column keys of column C1*

$q = z_{V2}^{-1} z_{C1} \bmod \Phi(n)$;

Send q to SP;

$Rke_{c1`} = <w_{C1}(w_{V2})^q, z_{C1}>$;

end

Assume that $<w_{V1}; z_{V1}>$ *be the column key of* V_1 *in* R_1;

// V2 and O2 are not needed as we get V` and O` from R_1 *already*

for each column C_2 *in* $(C2_1, C2_2, ..., C2_k)$ *from* R_2 *do*

Assume that $<w_{C2}; z_{C2}>$ *are Column keys of column C2*

$q = z_{V1}^{-1} z_{C2} \bmod \Phi(n)$;

Send q to SP;

$Rke_{c2`} = <w_{C2}(w_{V1})^q, z_{C2}>$;

end

Server-Side Operation:

// Compute the encrypted row-id of all tuples in R` firstly

for each row rw1 in R1 do

for each row rw2 in R2 do

calculate the encrypted r-id of the joined tuple as En(rw1) + En(rw2);

end

end

// Calculate encrypted values of columns originated from R1

for each column C1 in $(C1_1, C1_2, ..., C1_l, V_1, O_1)$ *from* R_1 *do*

Obtain q from DO

for each row rw1 in R1 do

for each row rw2 in R2 do

Assume that $c1_e$ *be the encrypted value on C1 in R1;*

Let $v2_e$ *be the encrypted value on V2 in R2;*

Compute encrypted value $c1`_e$ *for the row*

rw1 + rw2 as $c1_e (v2_e)^q \bmod n$;

end

end

end

// Calculate encrypted values of columns originated from R2

for each column C2 in $(C2_1, C2_2, ..., C2k)$ *from R2 do*

Obtain q from DO

for each row rw1 in R1 do

for each row rw2 in R2 do

Assume that $c2_e$ *be the encrypted value on C2 in R2;*

Let $v2_e$ *be the encrypted value on V2 in R2;*

Set the encrypted value $c2`_e$ *for the row*

rw1 + rw2 as $c2_e(v2_e)^q \bmod n$;

end

end

End

Algorithm 6. Procedure of transformation of one column in Cartesian product

3.8 [Aggregate] (Algorithm 7)

Count is the same as selection. So, we focus on SUM here. The major operation is the key update operation *(C1; <m_P; 0>)* where C1 is the column to be summed.

Inputted Values: *Column C1column key <w_{C1}; z_{C1}>*

Computed Result: *The encrypted sum σ_P of all values on C1.*

(Note that this encrypted sum is viewed as a relation R with one column P and One row only)

Client-Side Operation:

P = k(C1; <m_P; 0>); // DO executes client-protocol

// Note that P's column key is <m_P; 0>

Server-Side Operation:

P = k(C1; <m_P; 0>); // SP executes server-protocol

σ_P = 0;

for each row rw do

Let p_e be the encrypted value on P;

σ_{P+} = p_e mod n;

end

// σ_P is the encrypted sum w.r.t. r-id = 0.

Algorithm 7. SUM

3.9 [OEOP Operations] (Algorithm 8)

To handle OEOP operation with a plain column C1, a key update (C1; <w_{C1}; z_{C1}>)is executed where <w_{C1}; z_{C1}> is randomly determined by Data-Owner.

Inputted Values: *A column C1*

Computed Result: *R` = R1 ⊗ R2. R` has a schema (r-id, C1`). C1 in R` are the values from C1 in R1.*

Client-Side Operation:

$q = z_{V2}^{-1} z_{C1} \bmod \Phi(n)$;

Send q to SP;

$Rke_{c1`} = \langle w_{C1}(w_{V2})^q, z_{C1} \rangle$;

Server-Side Operation:

Obtain q from DO;

for each tuple (En(rw1), $c1_e$) from R1(r-id, C1) do

for each tuple (En(rw2), $v2_e$) from R2(r-id, V2) do

Insert the values (En(rw1) + En(rw2), $c1_e (v2_e)^q$) to R`(r-id, C1`);

end

End

Algorithm 8. OEOP transformation

4. Optimization

In our system, the query processing cost at user is insignificant as the user always operate on column key level and the cost is independent from database size. The encryption/decryption cost at user is relatively more expensive.

These actions and the majority of work at Service-Provider are computing modular exponentials, which are much more expensive than other operations, e.g., modular multiplication. So, it is important for our system to optimize modular exponential computations.

One way to reduce the cost is to use Garner's algorithm [25]. The basic idea is to use Chinese Remainder Theorem: computing $b = xe \bmod pq$ is equivalent to computing $b = [(xe \bmod q - xe \bmod p)*(p\text{-}1 \bmod q) \bmod q]*p + (xe \bmod p)$. This reduces the computation of a 1024-bit modular exponentiation to two 512-bit modular exponentiations with some cheaper additions and multiplications.

Besides, modular exponentials are commonly computed by exponential squaring. For example, to compute x7 mod n, we may compute $x2 = xx \bmod n$, $x4 = x2x2$ mod n and finally compute $x7 = xx2x4$ mod n. So, we need 4 multiplications only instead of 6 in trivial multiplications of x itself. If multiple exponential operations are having the same base, i.e., we are computing $be1$, $be2$, ... for the same b, the above preparation of exponents of squares can be saved. For example, if we need to further compute $x5$ mod n after computing x7 mod n, we just need 1 more multiplication $x5 \bmod n = x4x \bmod n$ instead of restarting from scratch.

In our scheme, the item key with column key $<w, z>$ and row key rw can be computed as mgrw.z mod n. It can be rephrased as mbrw mod n with $b = gz$. Thus, to compute the item keys of the same column but different rows, the item key generation is in fact a list of modular exponentials with the same base. This can significantly reduce the number of multiplications required to compute modular exponential for large exponents.

Furthermore, a careful choice of parameters can also help to reduce the cost. Note that the cost of modular exponential depends heavily on the size of exponent. If the exponent is small, modular exponentials can be computed efficiently.

(In practice, the public exponent e in RSA (where the encryption function is xe mod n for plain data x) is usually set to a small value, e.g., 65537.) As we discussed above, the encryption/decryption process is a list of modular exponentials with different exponents as row-id. So, if row-ids are small, the encryption/decryption cost can be further reduced.

5. Indexing

Indexing is an important feature in database system. The objective of index is to speed up query processing at Service-Provider. Indexing is essential as any query will incur at least $O(N)$ cost without using index where N is the number of tuples in the database. The processing cost is then not acceptable in large database, which may contain millions of tuples. It is then important to develop indexing techniques for our secure database system. On the other hand, this index has to be secure.

Note that indexing is by nature a security compromise. If Service-Provider skips processing certain tuples and the queries are known, Service-Provider can obtain some information about the original data. For example, if the query is to retrieve tuples with SALARY < 1000, Service-Provider can know that tuples pruned by the index are with SALARY 1000. This is the same scenario as comparison operator. So, the user has to balance between indexing effectiveness and security compromises.

[21] is a 'secure' (with user-controlled security strength) indexing scheme developed for querying on encrypted data. The general idea is to transform the original plain data into uncertain data. For example, in [21], data values 1, 5, 10 may be grouped into the same partition with range 0 to10.The partition is assigned an ID, say 1. Thus, instead of indexing on exact values 1, 5, 10, these values are indexed on the partition ID in the index. Service-Provider cannot observe the exact values of the data but the index can help to prune certain tuples on the index. The user can choose the granularity of uncertainty which controls the security compromises and the effectiveness of the index. If the uncertainty range is larger, it provides a better security protection but the index is less effective. The index here works independently with our encryption method. Before we use our operators to process the query, the database is first filtered by the index. This gives us a smaller number of tuples to be processed. Then, we will use our operators to find out the exact answer.

In our system, we set row-id rw as a random 32-bit number. It reduces the encryption/decryption cost while Service-Provider can't easily guess the correct value of rw as there are 232 possible values of rw.

6. System Architecture

VRS-DB is implemented as a software-layer on top of Apache Spark. Apache Spark is a fast and general engine for large-scale data processing, with in-memory primitives that enhance fast cluster computing. Apache Hadoop serves as the cluster manager and distributed storage system for very large data sets to support Spark's cluster computation. Apache Hive is the distributed data warehouse to store data.

Spark SQL is a component on top of Spark Core that provides support for structured data and SQL support. Spark SQL supports User-Defined Functions, which provides a mechanism for extending the functionality of the cluster database server by adding cryptographic function that can be evaluated in Spark SQL. This way, VRS-DB's server protocol is pushed to Spark execution engine and executed as UDFs.

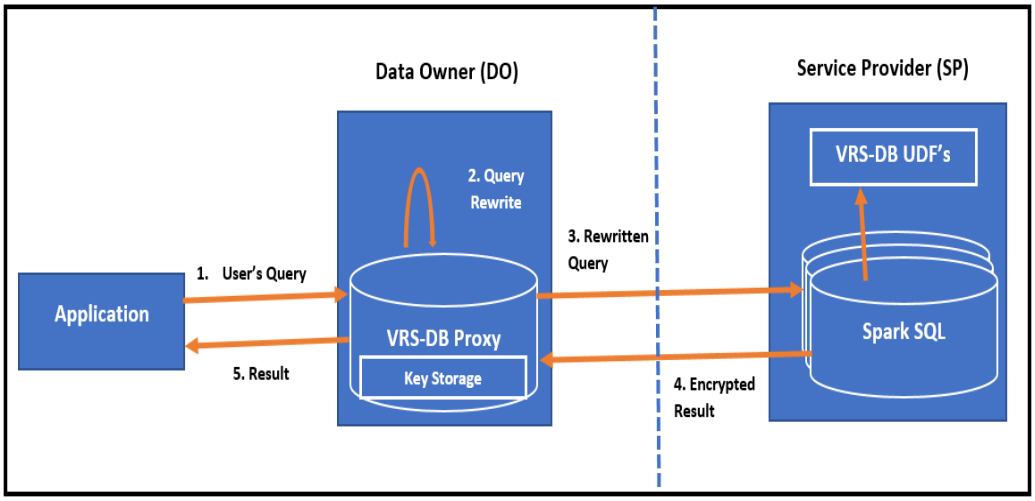


Figure 3. Architecture of VRS-DB

The VRS-DB Proxy resides on client side and stores the secret keys in the key store. A typical life cycle of a query consists of 5 stages:

1. Application submits a query to VRS-DB proxy.

2. VRS-DB proxy parses, analyses and rewrites the query with VRS-DB UDFs.

3. VRS-DB proxy submits the rewritten query to Spark SQL for server protocol execution.

4. Spark SQL sends back the encrypted query result to VRS-DB proxy.

5. VRS-DB proxy decrypts the query result and sends it back to application.

The advantages of VRS-DB's architecture include

6.1 Performance wise

• Secure operators are implemented as UDF which are executed in the same address space of the SparkSQL, which leads to less memory copy, less network transfer and no IPC.

6.2 Engineering wise

• VRS-DB leverages the SparkSQL's optimizer to optimize the server side queries.

• VRS-DB leverages all the normal operators provided by SparkSQL.

• Machine failures of the working nodes, disk-based processing and parallelism are well taken care of by Spark.

7. Experiments and Results

7.1 Experiment Environment

The prototype of VRS-DB is deployed on a cluster of 10 machines. Every server machine is equipped with eight Intel(R) Core(TM)

i7-3770 CPUs, four 8 GB Kingston DDR3 1333MHz RAM, running on Ubuntu Release 12.04. We use Apache Hadoop 2.4.1, Spark 1.1.0 and Hive 0.12.0, running on Java 1.6.0. The Hadoop File System is configured to have 3 replications of files and 64 MB data blocks. Spark executors are configured to have memory of 512 MB. The VRS-DB proxy resides on the client machine with a single Intel 2.40 GHz Core i7 CPU.

In our experiment, we answer the question of Q1: "How important it is for the secure database operators to be data interoperable in a cloud database environment?" Second, we address the questions of Q2: "How much performance overhead is introduced by VRS-DB to protect sensitivity? Is it practical in query processing?"

We prepare a synthetic table T with two sensitive columns A and B. The values of each column are generated with a uniform distribution. We execute 3 kinds of queries:

• [Range Query BE Mode] SELECT A, B from T WHERE A < q

• [Range Query OE Mode] SELECT A, B from T WHERE A < B

• [Count Query] SELECT count(A) from T WHERE A < q

The parameter q controls the selectivity of a range query. A smaller q leads to a smaller result size. We split the total query time into two components.

(1) Server Cost: The time taken for the cluster server to execute rewritten queries.

(2) Client Cost: The time taken for VRS-DB proxy to parse, analyse, rewrite queries and decrypt query results.

7.2 Experiment Result

7.2.1 VRS-DB vs. DBQ

To answer Q1, we compare the performance of VRS-DB with Decrypt-Before-Query Model(DBQ). DBQ is configured with the same cluster machines at server-side and runs Spark standalone mode at client side. Since no computation is done on sensitive data at server side, sensitive columns are encrypted using RSA encryption. For a sample query "SELECT A, B from T WHERE A < q", DBQ's server cost is simply the time taken to ship all the data back to client. DBQ client then decrypts all sensitive data to execute the range query.

As shown in Figure 4, the query processing time of DBQ is dominated by client cost, while VRS-DB is dominated by server cost. We conclude that the client cost of VRS-DB is significantly less than that of DBQ.

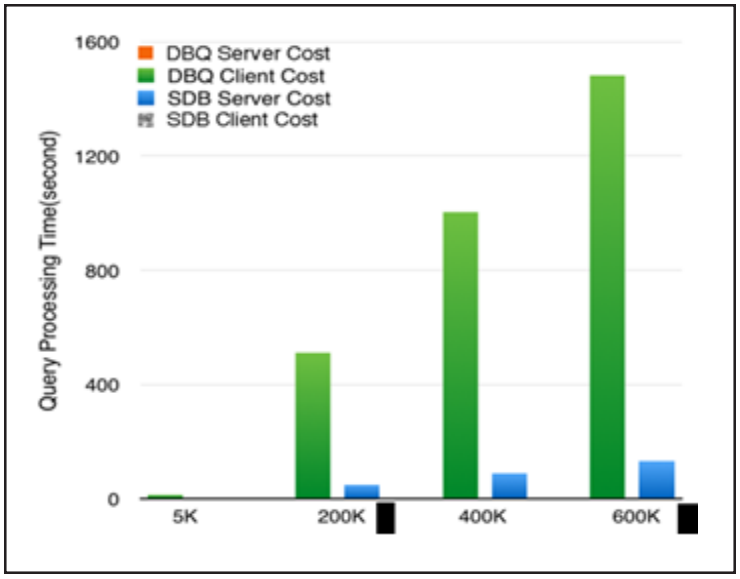


Figure 4. DBQ vs. VRS-DB

7.2.2 VRS-DB Cost Breakdown

We further break down the components of VRS-DB cost as follows. Figure 5 shows the client/server cost of a range EC query(on 200K dataset). With increasing selectivity, server cost stays stable, while client cost increases linearly. Figure 6 shows the further detail of client cost of a range EC query(on 5K dataset). This shows that decryption cost increases as the number of results to decrypt increases. Decryption cost is the major contributor to client cost. We observe that the decryption cost is significant, which may become the bottleneck of query execution when result size is large.

Therefore, improving the performance of decryption is one of the key optimizations to be done in the future.

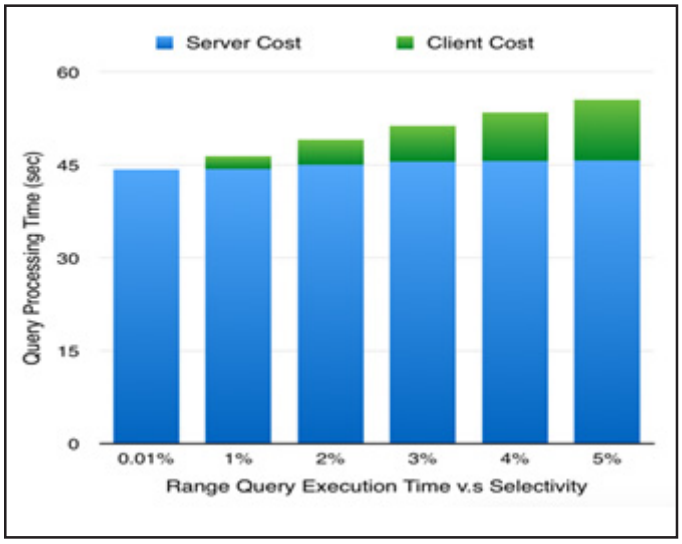


Figure 5. Client/Server Cost Vs Selectivity

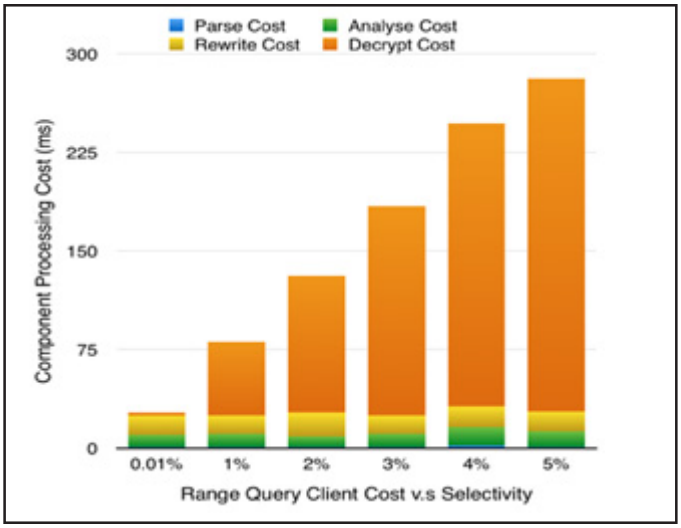


Figure 6. Client Costs Vs Selectivity

7.2.3 VRS-DB vs. SparkSQL

To answer Q2, we evaluate the performance of VRS-DB as follows. We first execute the three queries on SparkSQL directly with all columns stored as plaintext values, bypassing all secure operators.

The time to execute under this scenario is denoted as TSparkSQL. Then we execute the same queries on encrypted data on VRS-DB, and use TVRS-DB to denote the execution time. The radio TVRS-DB/TSparkSQL captures the degree to which VRS-DB's

encryption and secure protocol slows down the entire query processing time.

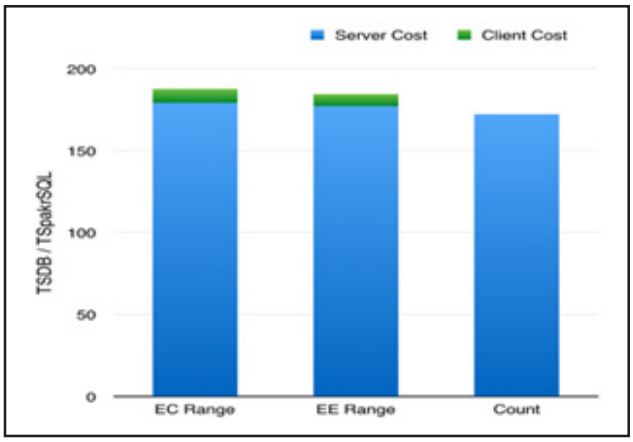
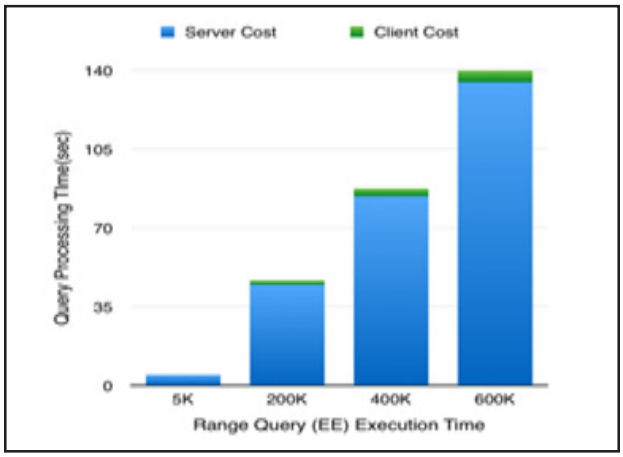
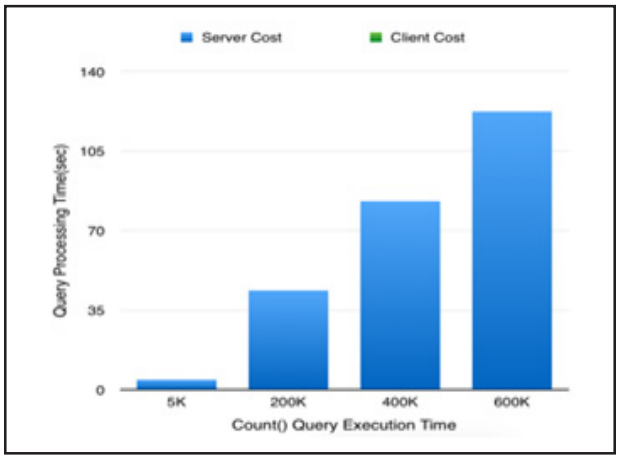


Figure 7. TVRS-DB / TSpark

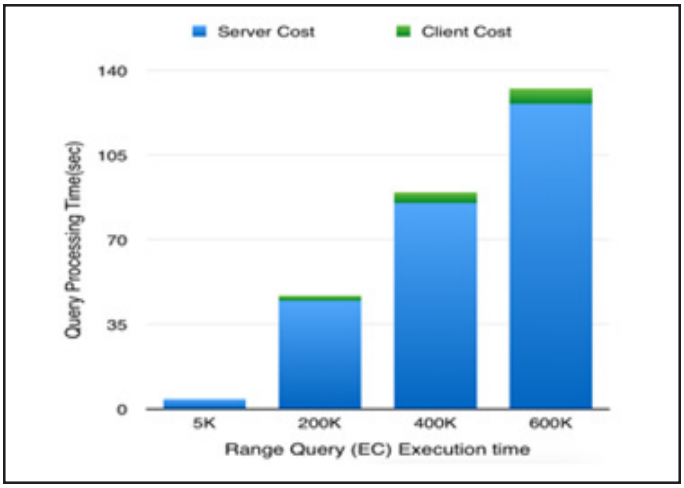
Finally, Figure 6 shows the client/server cost for the three queries with different data size.



(a)



(b)



(c)

Figure 8. Client/Server Cost vs. Data Size

We observe that VRS-DB introduces a 170 times slowdown of the query processing time. This is mainly the result of expensive modular exponent computation of big integers. In particular, the secure comparison involves exponent computation at least three times. This forms another aspect of future optimization we will focus on.

8. Conclusion and Future Works

The future work of VRS-DB is mainly divided into two parts: secure operator extension and cryptographic optimization.

8.1 Secure Operator Extension

VRS-DB is designed to support a wide range of query with several secure operators. In this, we included the query rewriting rules for the following secure operators which are not fully implemented yet:

- Secure multiplication, addition, subtraction, comparison in OEP mode
- Secure cartesian product
- Secure aggregation function: sum(), avg()

Since the query executor of VRS-DB proxy is implemented with the iterator pattern, VRS-DB proxy can also be extended to adopt the split client/server execution approach proposed by MONOMI[6]. This way, in the case of un-supported queries, VRS-DB proxy shall identify the parts of query supported at cluster server and push them to server as much as possible. Then VRS-DB proxy performs the rest of the queries.

8.2 Cryptographic Optimization

As we mentioned in the experiment, the decryption cost dominates other costs at client side. Therefore, it's crucial to improve the performance of decryption. During the computation of the item key of the same column but different rows, it can be computed as with . Thus it can be viewed as exponential computation with the same base, which can be optimized by Exponentiation by squaring [7]. To optimize such computation, we first pre-compute a table of b1, b2, b4, b8, b16, … To compute b13, we only need to retrieve b1, b2, b8 from table and multiply them together.

At server side, currently User Defined Functions at server side are all written in Java, whose modular exponential computation is slow. Instead, we can achieve 4x better performance [8] by calling native math methods in GMP library, which is written in C++.

Besides, there are studies on secure indexing techniques [9], which groups the value of sensitive columns into coarse ranges. Such index filtering can be applied before we process a query using our secure operators.

9. Conclusion

In conclusion, we gave a comprehensive description of VRS-DB's query rewriting and analysis of performance. By deploying such secret sharing scheme that supports multiple secure operators with data interoperability, VRS-DB is a query processing system that is both secure and practically efficient.

References

[1] Wong, W. K., Kao, B., Cheung, D. W. L., Li, R., Yiu, S. M. (2014, June). Secure query processing with data interoperability in a cloud database environment. In Proceedings of the 2014 ACM SIGMOD international conference on Management of data (pp. 1395-1406). ACM.

[2] Gentry., Craig., Shai Halevi., Nigel P. Smart. (2012). Homomorphic evaluation of the AES circuit. Advances in Cryptology–CRYPTO 2012. *Springer Berlin Heidelberg*, 2012. 850-867.

[3] Agrawal, R., Kiernan, J., Srikant, R., & Xu, Y. (2004, June). Order preserving encryption for numeric data. *In*: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (563-574). ACM.

[4] Bajaj, Sumeet, and Radu Sion. (2014). TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality. *Knowledge and Data Engineering, IEEE Transactions on* 26.3 (2014):752-765.

[5] Popa, Raluca Ada. (2011). Cryptdb: Protecting confidentiality with encrypted query processing. *In*: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. ACM.

[6] Tu, S., Kaashoek, M. F., Madden, S., Zeldovich, N. (2013). Processing analytical queries over encrypted data. *In*: Proceedings of the VLDB Endowment 6 (5) 289-300). VLDB Endowment. (March).

[7] Menezes, A. J., Van Oorschot, P. C., Vanstone, S. A. (1996). Handbook of applied cryptography. CRC press.

[8] Scott. (2014). Faster RSA in Java with GMP. Retrieved 19 April, 2015, from https://corner.squareup.com/2014/02/faster-rsa-jnagmp.html. (February).

[9] Hore, B., Mehrotra, S., Tsudik, G. (2004). A privacy-preserving index for range queries. In Proceedings of the Thirtieth International Conference on Very large Data bases- 30 (720-731). VLDB Endowment. (August).

[10] Rauthan, J. S., Kunwar Singh Vaisla. (2017). Privacy and Security of User's Sensitive Data: A Viable Analysis. *In*: Proceedings of the Second International Conference on Research in Intelligent and Computing in Engineering, *ACSIS*, 10, 67–71, DOI: http://dx.doi.org/10.15439/2017R45. (April).

[11] Rauthan J. S., Vaisla K. S. (2017). Scrambled database with encrypted query processing: CryptDB a computational analysis. *In*: Proceedings of the 1st IEEE International Conference on Intelligent Systems and Information Management (ICISIM) 199-21. (December).

[12] Felipe, M. R., Aung, K. M. M., Ye, X., Yonggang, W. (2015). Stealthycrm:A secure cloud crm system application that supports fully homomorphic database encryption, *In*: International Conference on Cloud ComputingResearch and Innovation (ICCCRI), October 2015, 97–105.

[13] Mallaiah, K., Ramachandram, S., Gandhi, R. K. (2015). Multi user searchable encryption schemes using trusted proxy for cloud based relational databases, In International Conference on Green Computing and Internet of Things (ICGCIoT), October 2015, 1554–1559.

[14] Liu, Y., Dong, M., Ota, K., Liu, A. (2016). Activetrust: Secure and trustable routing in wireless sensor networks, *IEEE Transactions on Information Forensics and Security*, 11 (9) 2013–2027.

[15] Li, H., Liu, D., Jia, K., Lin, X. (2015). Achieving authorized and ranked multi-keyword search over encrypted cloud data, *In*: Proceedings of ICC. IEEE, 2015, 7450 – 7455.

[16] Li, H., Dai, Y., Tian, L., Yang, H. (2009). Identity-based authentication for cloud computing, *In*: *Cloud computing*. *Springer*, 157–166.

[17] Li, H., Yang, Y., Luan, T. H., Liang, X., Zhou, L., Shen, X. S. (2016). Enabling fine-grained multi-keyword search supporting classified sub-dictionaries over encrypted cloud data, *IEEE Transactions on Dependable and Secure Computing*, 13 (3) 312–325.

[18] Li, H., Liu, D., Dai, Y., Luan, T. H. (2015). Engineering searchable encryption of mobile cloud networks: When qoe meets qop, *IEEE Wireless Communications*, 22 (4) 74–80.

[19] Tian, X., Huang, B., Wu, M. (2014). A transparent middleware for encrypting data in mongodb, *In*: IEEE Workshop on Electronics, Computer and Applications, May, 906–909.

[20] Wen, S., Xue, Y., Xu, J., Yang, H., Li, X., Song, W., Si, G. (2016). Toward exploiting access control vulnerabilities within mongodb backend web applications, *In*: IEEE Annual Computer Software and Applications Conference (COMPSAC) 1, 143–153. (June).

[21] Hou, B., Qian, K., Li, L., Shi, Y., Tao, L., Liu, J. (2016). Mongodb nosql injection analysis and detection, *in IEEE International Conference on Cyber Security and Cloud Computing* (*CSCloud*) 75–78. (June).

[22] Ramesh, D., Sinha, A., Singh, S. (2016). Data modelling for discrete time series data using cassandra and mongodb, *In: International Conference on Recent Advances in Information Technology (RAIT)* 598–601. (March).