CSIS0801 - Final Year Project

SecureDB - A Secure Query Processing System with data interoperability

Supervisor: Professor Benjamin Kao

Group Member: Lin Haibin

Table of Contents

1. Introduction 3

2. Project Background and Literature Review 4

3. Project Methodology 6

    3.1 Theoretical Background 6

        3.1.1 Encryption Procedure 6

        3.1.2 Secure Operators 7

    3.2 System Architecture 10

        3.2.1 Overall Architecture 10

        3.2.2 System Components 11

    3.3 Query Rewriting 14

        3.3.1 Query Rewriting for Multiplication 14

        3.3.2 Query Rewriting for Addition 15

        3.3.3 Query Rewriting for Subtraction 15

        3.3.4 Query Rewriting for Comparison 16

    3.4 Web Interface 18

        3.4.1 Web Interface for Data Upload 18

        3.4.1 Web Interface for Query 19

4. Experiments and Results 21

    4.1 Experiment Environment 21

    4.2 Experiment Result 22

5. Conclusion and Future Works 25

    5.1 Secure Operator Extension 25

    5.2 Cryptographic Optimization 25

    5.3 Conclusion 26

6. References 27

7. Appendix 28

    7.1 Rewrite Rules for Other Secure Operators 28

    7.2 Experiment Data

1. Introduction

As Database-as-a-service(DBaaS) such as Amazon Web Service and Microsoft Azure gets more economically available, security issue in the public cloud becomes one of the major concerns of data owners(DO). Despite the advantages of computation elasticity and scalability provided by the cloud service provider(SP), data owner cannot afford the risk of having highly sensitive data compromised on the cloud, such as clients' credit card information, as the public cloud may be vulnerable to privileged database administrators and hackers who may gain accesses to disk-resident data, or learn from the query results requested by DO.

SDB[1] is a secure query processing system that supports *data interoperability* with secret sharing encryption scheme, where the output of an operator can be taken as input of another operator, which leads to a wide range of SQL queries executable by cloud database, without revealing sensitive information. Besides security, we also demonstrate that SDB is practically efficient.

In this project, we implemented prototype of SDB by query rewriting, on top of Apache Spark. Specifically, this version of SDB supports the following kind of data interoperable query operations:

- Encryption for integer type data
- Secure operators: addition, subtraction, multiplication, comparison
- Secure aggregation function: count
- Web interface for table creation, data uploading and secure querying

2. Project Background and Literature Review

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, DO has to adopt a *Decrypt-Before-Query(DBQ)* approach, that is, shipping all ciphertext to DO, decrypting them before executing queries.

On the other hand, *fully homomorphic encryption*(FHE) 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] 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.

SDB is a secure query processing system for relational database, using secret sharing encryption schema. With the data interoperable secure operators supported by secret sharing encryption, SDB simulates fully homomorphic encryption for sensitive data stored in database, which leads to a wide range of queries supported by cloud database server. A typical query "`SELECT name, salary + 1000 FROM employee WHERE salary * 12 > 100000 AND age < 25`"(with salary being sensitive), which requires addictive, multiplicative and order-preserving homomorphism, is supported by SDB. Such query will be executed at cloud server first, afterwards data owner receives the encrypted query result (decrypted values of `salary + 1000` and plaintext of `name`) and decrypts it to get the original result. This way, SecureDB turns out to be both secure and efficient.

3. Project Methodology

3.1 Theoretical background

3.1.1 Encryption Procedure

SDB encrypts sensitive data with a secret sharing method per column. The DO maintains the secret keys of columns while SP 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 DO first generate two secret numbers, g and n. n is the product of two big random prime numbers ρ_1, ρ_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 A to be encrypted. DO generates a pair of *column key*s <m, x> for column A randomly. Also, for each row of column A, DO generates a distinct *row-id* randomly. We require that 0 < r, m, x < 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

$$v_k = gen(r, \langle m, x \rangle) = mg^{(rx \bmod \phi(n))} \bmod n.$$

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

$$[\![v]\!] = \mathcal{D}(v_e, v_k) = v_e v_k \bmod n$$

Row-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, DO only needs to store the column keys, while SP maintains the bulk encrypted data.

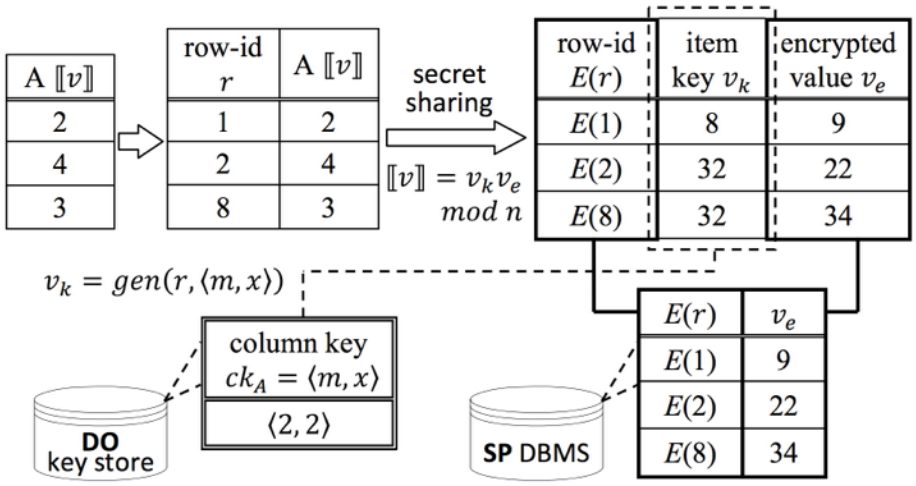

Figure 2: Encryption procedure ($g = 2$, $n = 35$).

3.1.1 Secure Operators

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

Operator	expression	description
×	A × B	vector dot product of two columns of the same table
+, −	A + B, A − B	vector addition/subtraction of two columns of the same table
=	A = B	equality comparison on two columns of the same table and output a binary column of '0' and '1'
>	A > B	ordering comparison on two columns of the same table and output a binary column of '0' and '1'
π	πS(R)	project table R on attributes specified in an attribute set S ⊗
Count	Count(R)	count the number of rows in a relation

Table 1: list of primitive secure operators

Notice that SDB supports data operability in different modes. For instance, the × operator, can be applied in three scenarios: operands are both encrypted(*EE Mode*); one operand is encrypted, the other is a constant(*EC Mode*); one operand is encrypted, the other is plain(*EP Mode*). Besides row-id, column R(with encrypted values of random number) and column S(with encrypted values of 1's) help achieve secure operation.

Data: Column A, B with column key $\langle m_A, x_A \rangle$ and $\langle m_B, x_B \rangle$
Result: $C = AB$ with C's column key $\langle m_C, x_C \rangle$
Client-protocol:
$x_C = x_A + x_B \bmod \phi(n)$;
$m_C = m_A m_B \bmod n$;
Server-protocol:
for *each row r* **do**
  Let a_e, b_e be the encrypted values on A, B;
  Set encrypted value of C $c_e = a_e b_e \bmod n$;
end

Algorithm 1: EE multiplication

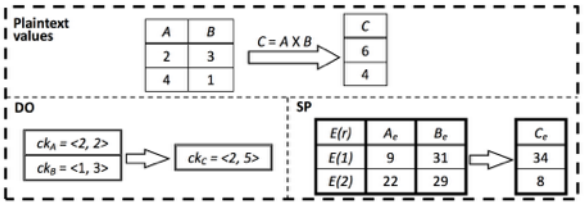

Figure 4: $C = A \times B$ $(g = 2, n = 35)$.

The protocol for secure multiplication in EE mode, and an illustrative figure are listed above. To calculate the secure multiplication between two sensitive columns, DO simply generates a new column key while SP performs expensive computation on bulk data. The encrypted result is then sent back to DO and decrypted by the new column key. The proof for correctness and security can be found in [1].

Key update is a helper operator which takes an column A and a target column key <m_C, x_C> and output a column C that shares the same plaintext of A with target column key, without revealing sensitive information. Other protocols of secure operators implemented in SDB are also listed below.

Data: Column A with column key $\langle m_A, x_A \rangle$ and a constant p
Result: $C = pA$ with C's column key $\langle m_C, x_C \rangle$

Client-protocol:
$x_C = x_A$;
$m_C = pm_A \bmod n$;
Indicate that C's encrypted column is A's encrypted column;

Server-protocol:
Nil

Algorithm 2: EC multiplication

Data: Column A, B with column key $\langle m_A, x_A \rangle$ and $\langle m_B, x_B \rangle$
Result: $C = A + B$ with C's column key $ck_C = \langle m_C, x_C \rangle$

Client-protocol:
Generate random m_C, x_C
$A' = \kappa(A, ck_C)$; `// DO executes client-protocol`
$B' = \kappa(B, ck_C)$; `// of key update.`
Set C's column key as $\langle m_C, x_C \rangle$;

Server-protocol:
$A' = \kappa(A, ck_C)$; `// SP executes server-protocol`
$B' = \kappa(B, ck_C)$; `// of key update.`
for *each row* r **do**
  Let a'_e, b'_e be the encrypted values on A', B';
  Set encrypted value of C $c_e = a'_e + b'_e \bmod n$;
end

Algorithm 4: EE addition/subtraction

Data: (i) Column A with column key $\langle m_A, x_A \rangle$; (ii) target column key $\langle m_C, x_C \rangle$
Result: $C = A$ with C's column key $\langle m_C, x_C \rangle$

Client-protocol:
Let $\langle m_S, x_S \rangle$ be the column key of S;
$p = x_S^{-1}(x_C - x_A) \bmod \phi(n)$;
$q = m_A m_S^p m_c^{-1} \bmod n$;
Send p, q to SP;
Set C's column key as $\langle m_C, x_C \rangle$;

Server-protocol:
Obtain p, q from DO;
for *each row* r **do**
  Let a_e, s_e be the encrypted values on A, S;
  Set encrypted value of C $c_e = qa_e s_e^p \bmod n$;
end

Algorithm 3: Key udpate

Data: Column A, B with column key $\langle m_A, x_A \rangle$ and $\langle m_B, x_B \rangle$
Result: A column of comparison results C: $= 0$ if $A = B$; $= 1$ if $A > B$; return $= -1$ if $A < B$

Client-protocol:
$Z = R(A - B)$; `// EE addition, EE multiplication`
$Z' = \kappa(Z, \langle 1, 0 \rangle)$; `// DO executes client-protocol`

Server-protocol:
$Z = R(A - B)$; `// Corresponding server protocol`
$Z' = \kappa(Z, \langle 1, 0 \rangle)$; `// Corresponding server protocol`
for *each row* r **do**
  Let z'_e be the values on Z';
  switch z'_e **do**
    case $= 0$
      $c_e = 0$; `//` c_e `is the result of this row`
    end
    case > 0
      $c_e = 1$;
    end
    case < 0
      $c_e = -1$;
    end
  endsw
end

Algorithm 5: Comparison

3.2 System Architecture

3.2.1 Overall Architecture

SDB 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, SDB's server protocol is pushed to Spark execution engine and executed as UDFs.

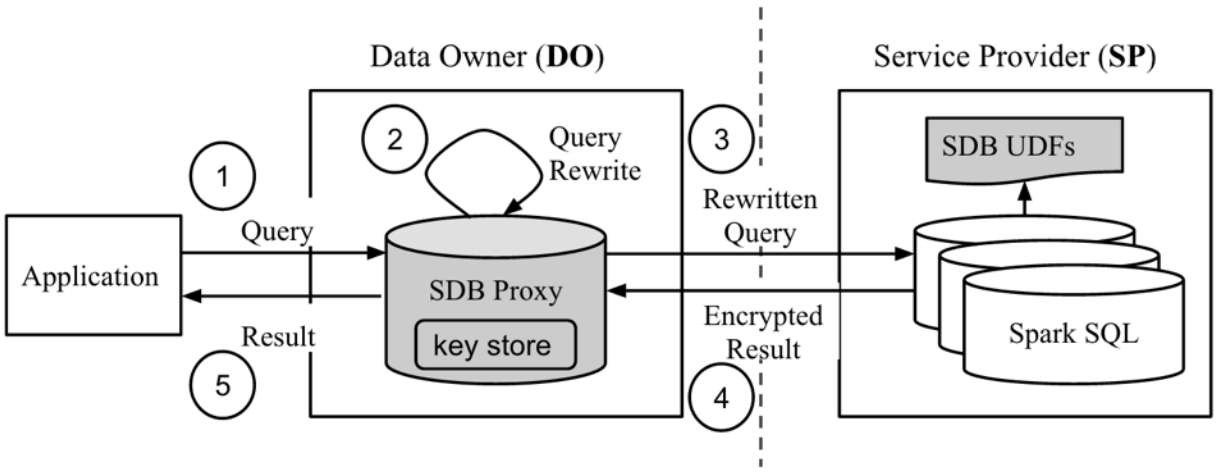

Figure 5: Architecture of SDB

The SDB 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 SDB proxy.
2. SDB proxy parses, analyses and rewrites the query with SDB UDFs.

3. SDB proxy submits the rewritten query to Spark SQL for server protocol execution.
4. Spark SQL sends back the encrypted query result to SDB proxy.
5. SDB proxy decrypts the query result and sends it back to application.

The advantages of SDB's architecture include

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.
2. Engineering wise
   - SDB leverages the SparkSQL's optimizer to optimize the server side queries.
   - SDB 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.

3.2.2 System Components

SDB proxy is composed of 4 major components:

1. SDB *Query Parser*
2. SDB *Semantic Analyser*
3. SDB *Query Rewriter*
4. SDB *Query Executor*

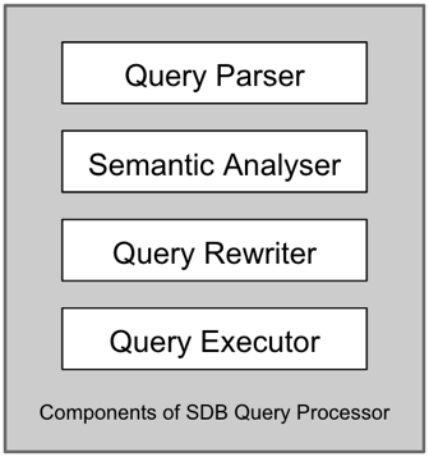

Figure 7: Components of SDB Proxy

SDB Query Parser parses a query string and transforms it into an *abstract syntax tree(AST)*. It is built based on HiveQL Parser, with additional tokens for column sensitivity declaration. For example, to declare a table T with sensitive column A and non-sensitive column B, the user has to submit query "`CREATE TABLE T (A int ENC, B int)`", which is transformed into the AST in figure 8.

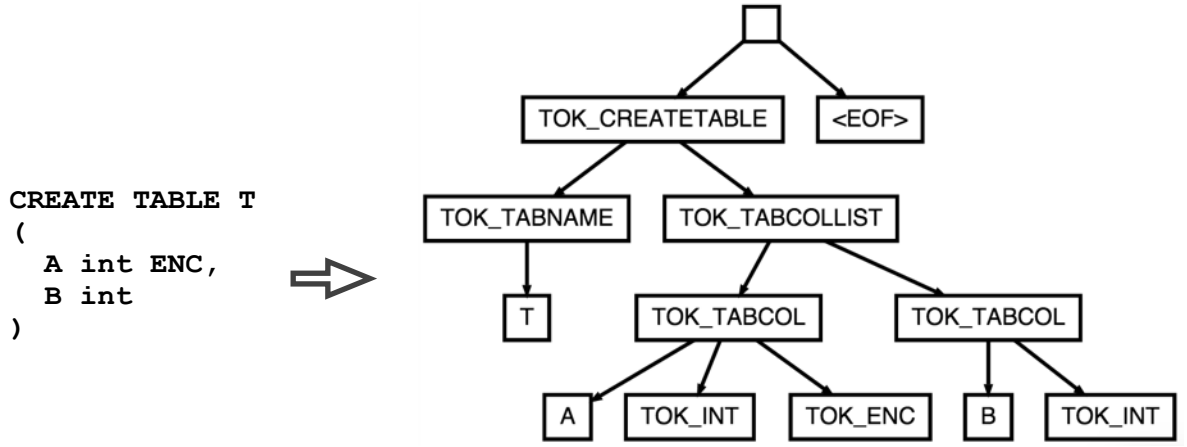

Figure 8: AST of "CREATE TABLE T (A int ENC, B int)

SDB Semantic Analyzer transforms an AST into a *Logical Plan Tree(LPT)* as the preparation for query rewrite. The semantic analyser traverses the AST to construct a LPT, which is easier to manipulate with, and access the key store to

1. verify whether the reference of queried tables and columns are valid
2. annotate a column with its sensitivity and its column key, if any

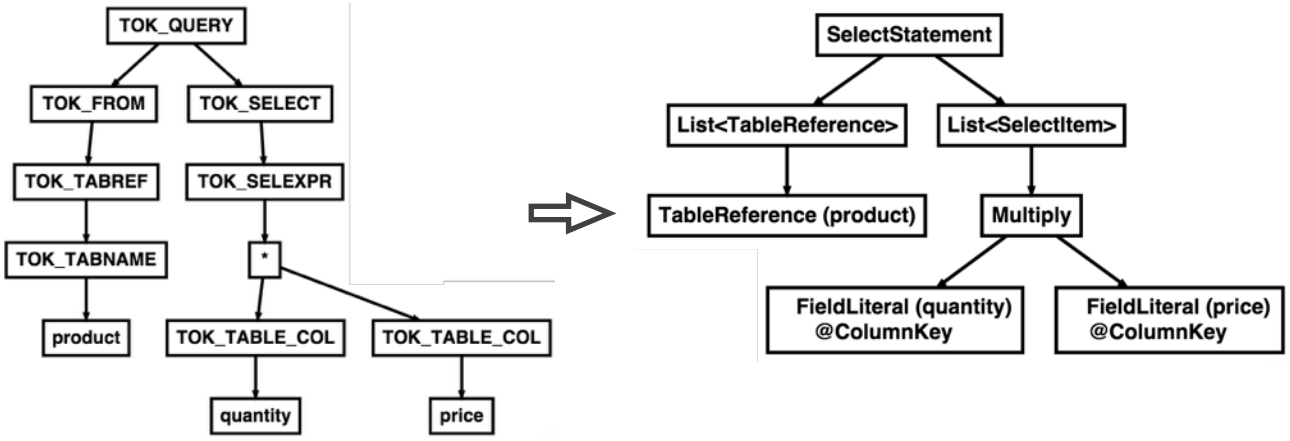

Figure 9: Transformation from AST to LPT

SDB Query Rewriter rewrites the LPT with secure operator, and constructs a *Physical Plan Tree(PPT)*. While traversing LPT, every time the rewriter identifies a normal arithmetic operator involving sensitive columns, it replaces the normal operator with the corresponding secure operator, based on SDB rewrite rule(discussed in detail in next section). Finally the rewrite construct a physical plan tree for SDB Executor to execute.

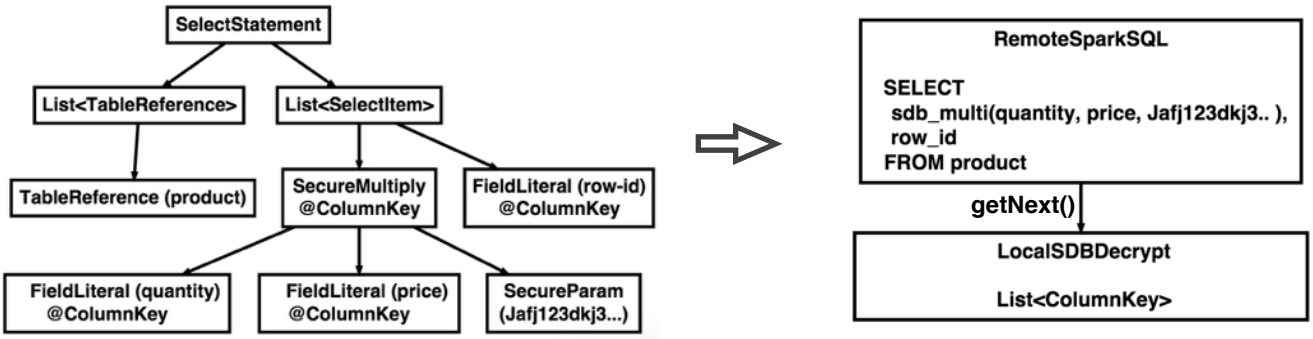

Figure 9: Transformation from LPT to PPT

SDB Executor executes the physical query plan and returns the result set. Finally, the executor executes the physical plan tree which normally splits the execution into two phases. In the first phase, it sends server-side query and waits for query result. In the second phase, it receives server-side query and decrypts query result. The decrypted result is then sent back to client application via connector.

3.3 Query Rewriting

We denote $ck_A = \langle m_A, x_A \rangle$ as the column key for column A, denote $\kappa_{AC} = \langle p, q \rangle$ as the client-side result of key update column A with target column key $\langle m_C, x_C \rangle$. Besides, we use ck_Z to denote the special column key <1, 0>.

Function	Explanation
rand(ρ_1, ρ_2)	Client-side function which generates a random number co-prime with n, $\Phi(n)$
key_update_client(ck_A, ck_C, ck_S, ρ_1, ρ_2)	Client-side key update which computes κ_{AC}
decrypt(C_e, ck_C, n, g, row_id)	Client-side decryption which computes ⟦c⟧
sdb_mul(Ae, Be, n)	Server-side multiplication (A × B) UDF
sdb_add(Ae, Be, Se, κ_{AC}, κ_{BC}, n)	Server-side addition (A + B as C) UDF
sdb_compare(Ae, n)	Server-side comparison (A > 0) UDF
sdb_key_update(Ae, Se, κ_{AC}, n)	Server-side key update UDF

Table 2: List of functions for query rewriting

3.3.1 Query Rewriting for Multiplication

According to the secret sharing scheme, query rewriting for secure multiplication is straightforward.

Input: SELECT A × B as C FROM T

$ck_c = \langle m_A \times m_B, x_A + x_B \rangle$

Output: SELECT row_id, sdb_mul(A_e, B_e, n) AS C_e FROM T

Input: SELECT A × u as C FROM T

$ck_C = \langle m_A \times u, x_A \rangle$

Output: SELECT row_id, A_e AS C_e FROM T

Rule 1: Rewrite Rule for Multiplication (EE & EC mode)

3.3.2 Query Rewriting for Addition

For secure addition in EE Mode, addition is achieved by applying key update to both A and B so that they share the same column key.

Input: SELECT A + B as C FROM T

ck_C = < rand(ρ_1, ρ_2), rand(ρ_1, ρ_2) >
κ_{AC} = key_update_client(ck_A, ck_C, ck_S, ρ_1, ρ_2)
κ_{BC} = key_update_client(ck_A, ck_B, ck_S, ρ_1, ρ_2)

Output: SELECT row_id, sdb_add(A_e, B_e, S_e, κ_{AC}, κ_{BC}, n) AS C_e FROM T

Rule 2: Rewrite Rule for Addition (EE mode)

For secure addition in EC mode, the expression `A+u` can be considered as `A+(S×u)`, as the plaintext of S is 1. Thus an EC addition operation can be done as an EE addition after EC Multiplication.

Input: SELECT A + u as C FROM T

ck_{Su} = <u × m_S, x_S>
ck_C = < rand(ρ_1, ρ_2), rand(ρ_1, ρ_2) >
κ_{AC} = key_update_client(ck_A, ck_C, ck_S, ρ_1, ρ_2)
κ_{SuC} = key_update_client(ck_{Su}, ck_C, ck_S, ρ_1, ρ_2)

Output: SELECT row_id, sdb_add(A_e, S_e, S_e, κ_{AC}, κ_{SuC}, n) as C_e FROM T

Rule 3: Rewrite Rule for Addition (EC mode)

3.3.3 Query Rewriting for Subtraction

For secure subtraction in EE Mode, the expression `A−B` can be considered as `A+(B×(-1))`, similar to addition in EC mode. Based on the modular equation -1 mod n == (n − 1) mod n, we compute the column key of -B as follows.

Input: SELECT A − B as C FROM T

ck_{Bi} = < (n−1) × m_S, x_B > *// inverse the value of B*
ck_C = < rand(ρ_1, ρ_2), rand(ρ_1, ρ_2) >
κ_{AC} = key_update_client(ck_A, ck_C, ck_S, ρ_1, ρ_2)
κ_{BiC} = key_update_client(ck_{Bi}, ck_C, ck_S, ρ_1, ρ_2)

Output: SELECT row_id, sdb_add(A_e, B_e, S_e, κ_{AC}, κ_{BiC}, n) AS C_e FROM T

Rule 4: Rewrite Rule for Subtraction (EE mode)

For secure subtraction in EC mode, the expression `A−u` can be considered as `A−S×u`, thus we simply compute based on the rule of EE subtraction.

Input: SELECT A − u as C FROM T

$ck_{Su} = <u \times m_S, x_S>$ // $S \times u$
$ck_{Sui} = < (n-1) \times m_{Su}, x_{Su} >$ // $-(S \times u)$
$ck_C = < rand(\rho_1, \rho_2), rand(\rho_1, \rho_2) >$
$\kappa_{AC} = key\_update\_client(ck_A, ck_C, ck_S, \rho_1, \rho_2)$
$\kappa_{SuiC} = key\_update\_client(ck_{Sui}, ck_C, ck_S, \rho_1, \rho_2)$

Output: SELECT row_id, $sdb\_add(A_e, S_e, S_e, \kappa_{AC}, \kappa_{SuiC}, n)$ AS C_e FROM T

Rule 5: Rewrite Rule for Subtraction (EC mode)

3.3.4 Query Rewriting for Comparison

For secure comparison in EC mode, the expression `A>B` can be considered as `(A−B)>0`, to ensure no sensitive information is leaked, we randomize the result by `R×(A−B)>0` at server side so that the result of `A−B` is not revealed.

Input: SELECT A FROM T WHERE A > B

$ck_{Bi} = < (n-1) \times m_S, x_B >$ // *inverse the value of B*
$ck_C = < rand(\rho_1, \rho_2), rand(\rho_1, \rho_2) >$
$\kappa_{AC} = key\_update\_client(ck_A, ck_C, ck_S, \rho_1, \rho_2)$
$\kappa_{BiC} = key\_update\_client(ck_{Bi}, ck_C, ck_S, \rho_1, \rho_2)$ //$A - B$
$ck_{RC} = < m_R \times m_C, x_B + x_C >$ // $R \times (A - B)$
$\kappa_{RCZ} = key\_update\_client(ck_{RC}, ck_Z, ck_S, \rho_1, \rho_2)$

Output: SELECT A, row_id FROM T
WHERE $sdb\_compare(sdb\_key\_update(sdb\_mul(R_e, sdb\_add(Ae, Be, Se, \kappa_{AC}, \kappa_{BiC}), n), S_e, \kappa_{RCZ}, n), n) > 0$

Rule 6: Rewrite Rule for Comparison (EE mode)

For secure comparison in EC mode, the expression `A>u` can be considered as `(A−S×u)>0`. Notice that we give the rewriting rule for >, to apply it to = or <, simply swap > with the target equality operator.

Input: SELECT A FROM T WHERE A > u

$ck_{Su} = <u \times m_S, x_S>$ // $S \times u$
$ck_{Sui} = < (n-1) \times m_{Su}, x_{Su} >$ // $-(S \times u)$
$ck_C = < rand(\rho_1, \rho_2), rand(\rho_1, \rho_2) >$
$\kappa_{AC} = key\_update\_client(ck_A, ck_C, ck_S, \rho_1, \rho_2)$
$\kappa_{SuiC} = key\_update\_client(ck_{Sui}, ck_C, ck_S, \rho_1, \rho_2)$ // $A - S \times u$
$ck_{RC} = < m_R \times m_C, x_B + x_C >$
$\kappa_{RCZ} = key\_update\_client(ck_{RC}, ck_Z, ck_S, \rho_1, \rho_2)$ // $R \times (A - B)$

Output: SELECT A, row_id FROM T
WHERE $sdb\_compare(sdb\_key\_update(sdb\_mul(R_e, sdb\_add(A_e, S_e, S_e, \kappa_{AC}, \kappa_{SuiC})), S_e, \kappa_{RCZ}, n), n) > 0$

Rule 7: Rewrite Rule for Comparison (EC mode)

Figure 10 shows the rewritten physical plan tree of EE Comparison query.

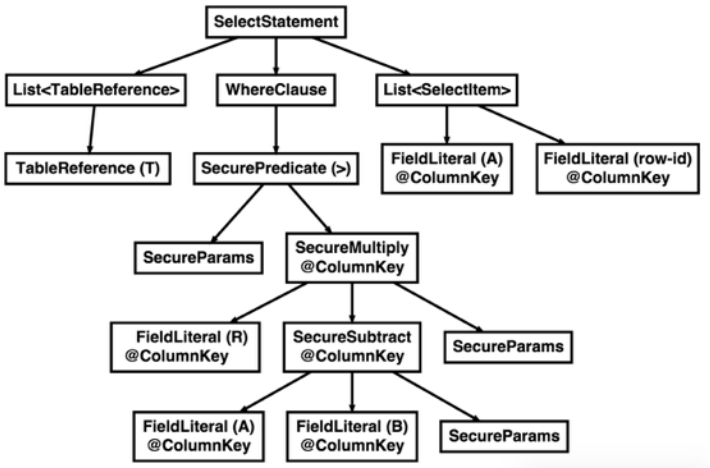

Figure 10: Rewritten LPT of SELECT A FROM T WHERE A > B

3.4 Web Interface

3.4.1 Web Interface for Uploading Data

We introduce the steps to upload a table of sensitive data as follows.

1. In the *SQL Editor* page, type in the CREATE TABLE query
2. Append keyword "`ENC`" to every sensitive field
3. Click "Execute SQL Query" button

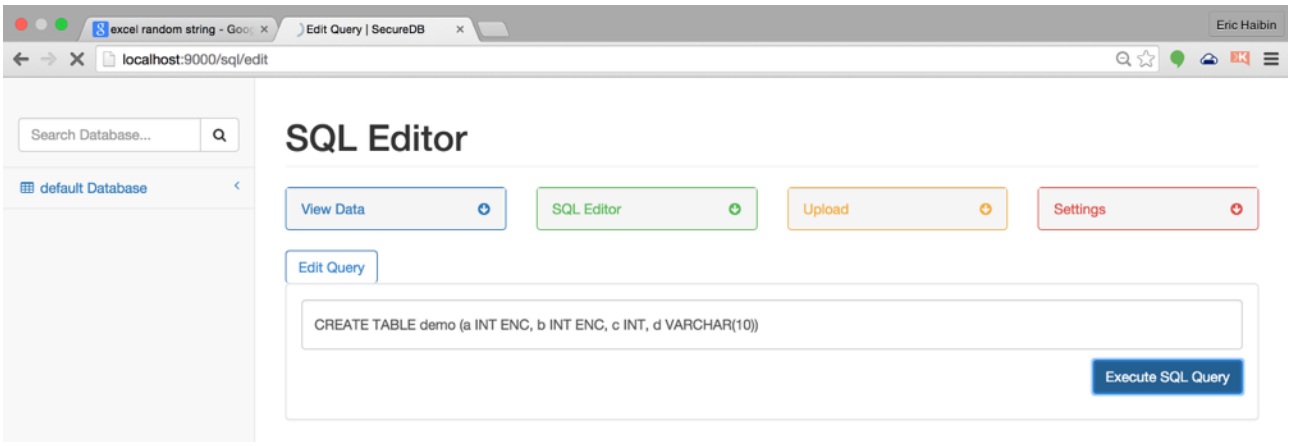

Figure 11: Web interface to create a table with two sensitive fields (A, B)

4. Receive confirmation message "Create Table Successful" after success

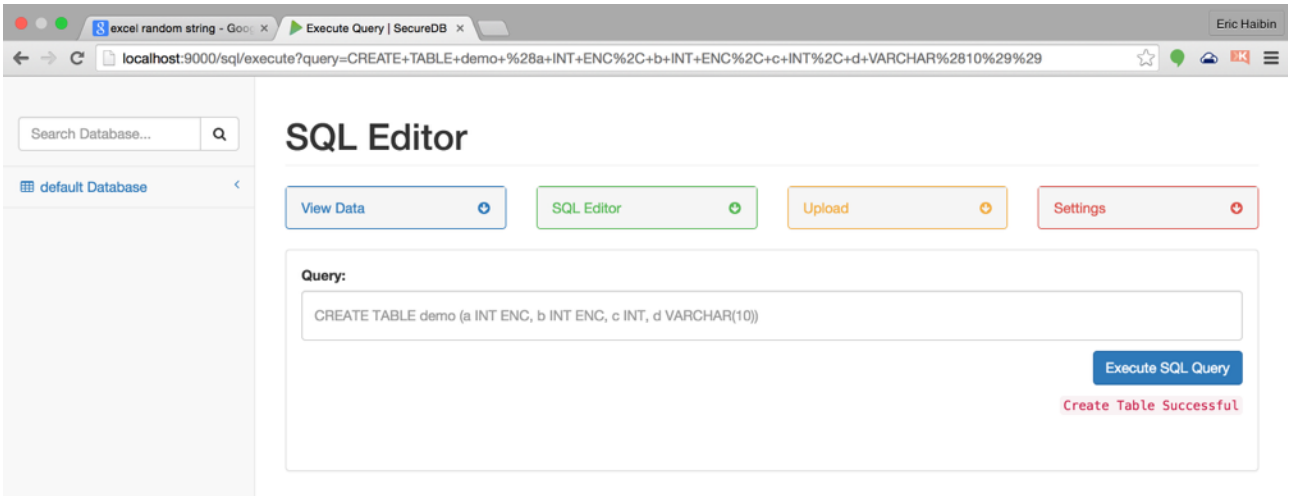

Figure 12: Confirmation message of table creation success

5. In the *Upload Data* page, select table name and data source file to upload.
6. Click "Upload Data" button to start the upload process.

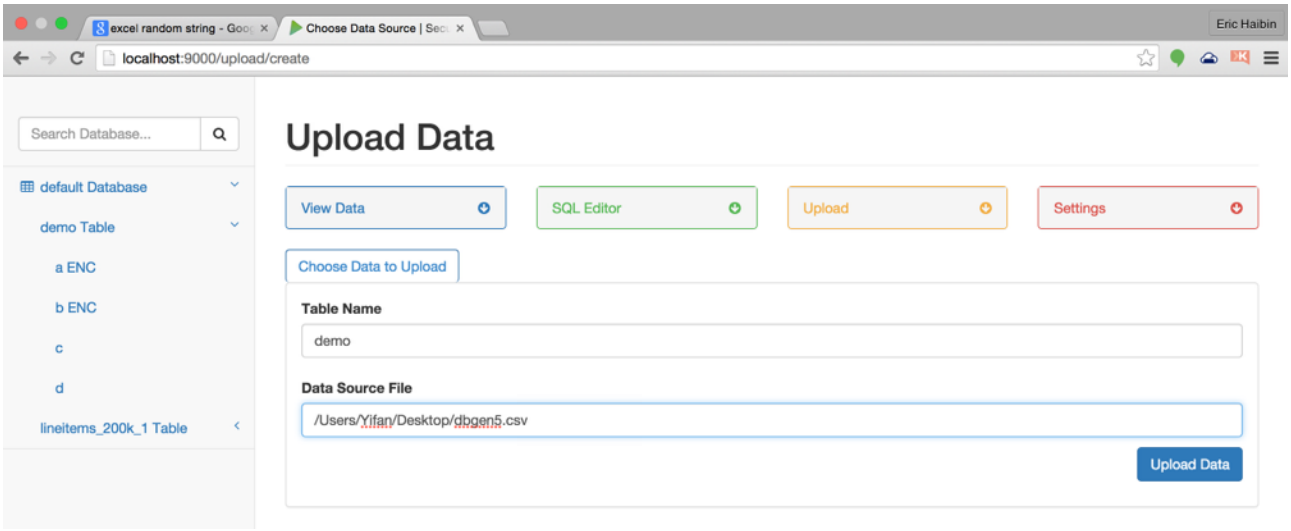

Figure 13: Web interface to upload data to table

3.4.2 Web Interface for Query

We introduce the interface for query as follows.

1. Type in the query in the SQL Editor page
2. Click “Execute SQL Query” button
3. Query result will be displayed as in Figure 14

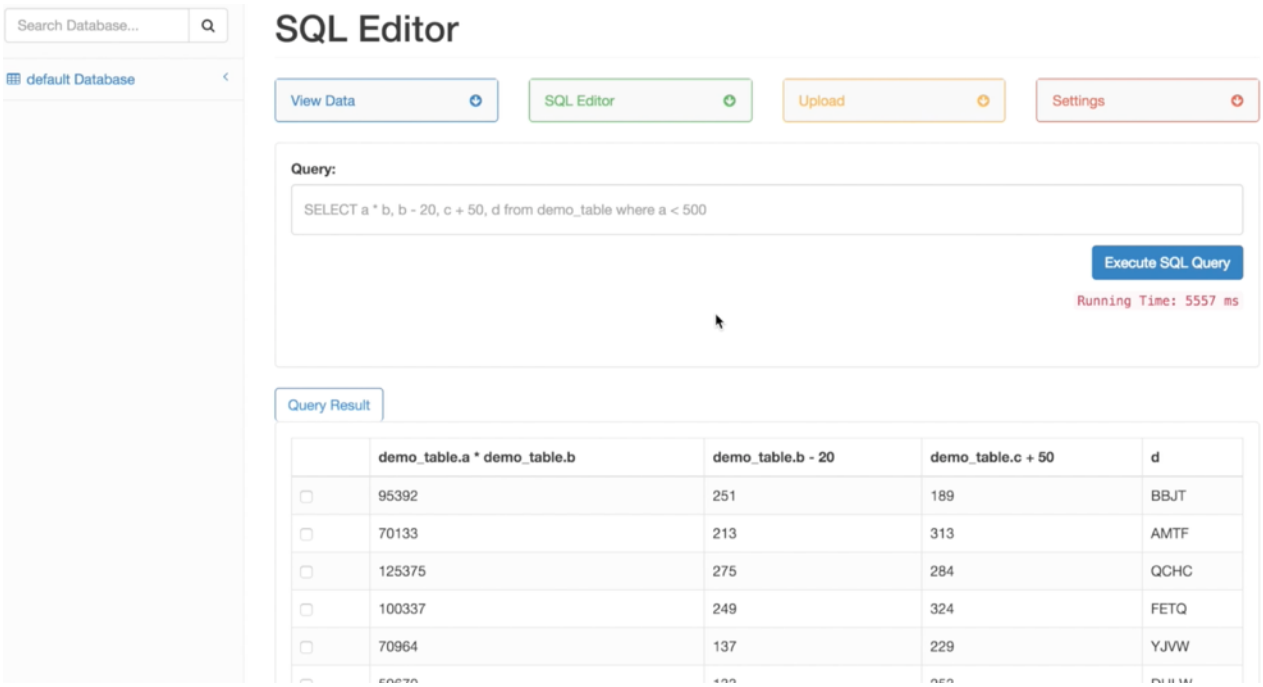

Figure 14: Web interface to execute query

4. At “Execution Analysis” section, the query executed at cluster server is displayed

5. Client and server side query execution cost are visualized in charts

Figure 15: Query Performance Analysis Section

Cloud service provider’s view of encrypted sensitive data is illustrated in Figure 16.

```
➜ ~ head mem_dump
50b7ut4gz48pryu21zjjq291b4b32mpk5ag3bg53lcytgvlny6utbcile9uddr2gswqzsiq2rpydugctzpuyjlup4v55y3qtkyf6f9n2b5776o79jfaf56w4dcfws5op
cl2wlb8ry5zuf2t1awlyjp3nw3swuu5h3n6mzc0g4swttxai61w9kpyil34zkrgb513gfz  c9brqgauvebfelrv1c90yup29blae2k8pav16nyynr62z47yr0qy6t9q
6avx0koy6bad3c5rop5p90jb13py7rwyaec69qkr9wgpmt251gqz0baecwehrhoi6rnylytiphubgdqu410hbj57e25jin296ry9r0emwr7ois2f0y9jpfpkt17wmte8
rz6pki45he4bh2  7s81pqulaju3dfw8z0svmnmtvzv9gadxnge3l3vbyh145y97ckm87dqer1juc1gj7bstx8v149f04zbm8qgs7mcgrra6ki3xpe7dnhlkt3iqdv1c
kusqybikjxcnp4d15vantu81ea9fr9f91bsx4u4az269oeto4b83uymfwj96515fzdxo71y8t1y171lhg6pwos
hrt2ljkec167i1w00yikqtdvbkhgsio6zjddzl4foo4t2nrlyfbmsqjrhd6e39jjyfq8rb98n4ulr1p6iuex63102rst6lw7ekg8t0u49uvwpbsvp92le7851llpbayb
wk71e740abhkgnkxypca8qlf0ziefn03fenrgxl0vtndy37awukpi4j1yb9umw69heqiks  gmhp4cv0k9k5tmhnm4021rbzkp15dtp53fc6sak1h27xi92x3u0ky9yw
5zvg9n4iwz2fk17amchh0sooc0so0vehf53remqac8o98kfxenp67bqqooeieh0n2opicebus5kg0ttd1n4j3hxb5ce15ffn5pidzepkavysvlh2uow1135te2qwk052
c6wgdca47gn3o  os3zb03zjcz5ba1fk9pjnbmhc676zdm4ao9h9pzb42t5r3ed3n8fpleol9g0jfobodth27kc8o6bq346y0ia33ee4c7ydo9af7nakea05o1gvz7c
v04e0w6f8u9boxir2j815rt7gh6l8sulh7gxrdun5elmbplc7x2lpgnub95pfo6wzgh99iy8u6tnyjf7jb48n6
izrp7p35cirf7gremw6vooex8cjxz9llj0f13lbjq133bbzyrd3w39uc9e4ogsbhbon22po4te6qc77yj88ja9d4vr0t32qsfdx21r4mvo3qy1fl9l43fc3zkq0rppuf
hwzgj13eaihyfzn2uu33aak8krgbhnh6r7lh09rdqe3ls348us6v5vb5quz4kt7w83t7cx  sicexwb14y3alcvpo6tmb1r1com70s9y3y4yyhu7huwax9i48dexrxv8
pkzld1oh03lozdic3i9cz6mzu7ybgeehlkjp9izdho2hlr77roqdcjpbzz05gvdcdpriqsb73vqu8w4l828w2wtukhvi74mlrgvkuz0jie3hctkfooyjcf6nxcarlf4v
1uq9xmdygm2whq  bs84e65o2o7dr99jq9xql0luabls7nlcojirxtql55vuz8n1dsmjua0u86bwnmmsclmz9r36ewi8zr9wygjm0syr4l8nfa1d3mhqj83nk2dznczo
2mranbun62yi7vvup0kfwr8spv0xa96mn7rdmb8g1t58wo94fff91prs26xbylt4j8el6wpcd76u2j37ym7qdv
5szwp3mp2pwkc6nkxsgpwj2ejalgwqezdhtx822r2nl1zuw7m2c8qgta0ytufdkfl34crxn9a00xk2cguk1lhu6u1tn954kox53iuuc0p9pmdtncqgjaekvi3us2v552
w9tsz6q2n7u1ak6byorkfwhll5njopycz9fdti4qr0qk65twmgwh6t9ol5kayceblk8bo0y  sj9jnyqanzkcrru3p17mdic0ovnappfdjca4vj0o5fw9xag9axiht82q
xmnzv7kw2jzk06c66x64lbp3c0qc8mivgenvh5ht3f9i9wc8aivbrqn6zrqxbd7kvvnvtx40m1gibfvjyogcjz3pwz2p1k9mwclke0u9jtdgeh380miuvcgoy8f222lv
78777zbzbnuey  2wavpwiy22aa81y3v4zrejba9ite9djwpnbetcb1mjt7u002vwduf08y0b7hiwgfnhhpkr1w0m3m8rep4wfn9c9gvnuxy521xzq9q0z4ejtom2r3
5czl54afbwen6nsf2k6evmwy7mbtltl04tc209nx3382kaf73g9rpo29yieau5wjn8gcrppql26ckclpiwfgu5
fxe4i37zmp8epy9clrvgna72ru9xunwry13oyuyih1hih3lxo1yo5g71auhwby86ln1iuqs47r80xujsbwq9nu1r7n45xnozijcyk0sc03ors6vaucxmcn6s9d71sh0y
cvlzi8r5ds0048631psmisk3xkr7vuonaiktsw7vcf427wue4wr5x2820cbrb13k4mppql  q702ewvmedatmq3orz49wjlmq3eb1o8pcoklridhcex5z3jqm3cen8gy
v1tfrarduhc9miptlv26yr57maq96zpogqik6rx4d2gwfk7a8hkhxnwpgken5az4a2g9wum1qomet3mmztfirvg7a8ijw485f6o4yu581mrvhzq3zs4wx1cn5u8xa77k
1aikd9xn92122w  jalezkql9jrg7v5w4nk4g4erxejubkt34xf8a64l3vpp6n3c17d2c7ironb7hdo4ih9vhcupfqitde4u496iehhdwg75f81mgjfq8nua2hv3zmm8
rnsqmqx3owq7s5c9huw3j9fjlz8buxeog4xf59qcpo88hyzrpytog1l7pip41jqrjlq3flsth3sj8t0lwcvrlz

➜ ~
➜ ~
```

Figure 16: Cloud Service Provider’s View of Data

4. Experiments and Results

4.1 Experiment Environment

The prototype of SDB is deployed on a cluster of 10 machines. Every server machine is equipped with eight Intel(R) Core(TM) i7-3770 CPUs, four 4GB 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 replication of files and 64 MB data blocks. Spark executors are configured to have memory of 256 MB. The SDB proxy resides on the client machine with a single Intel 2.40 GHz Core i5 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 SDB 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 EE Mode*] `SELECT A, B from T WHERE A < q`
- [*Range Query EC 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 SDB proxy to parse, analyse, rewrite queries and decrypt query results.

4.2 Experiment Result

4.2.1 SDB vs. DBQ

To answer *Q1*, we compare the performance of SDB 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 17, the query processing time of DBQ is dominated by client cost, while SDB is dominated by server cost. We conclude that the client cost of SDB is significantly less than that of DBQ.

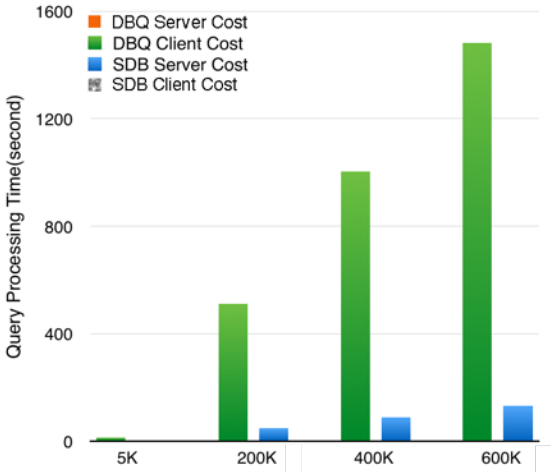

Figure 17: DBQ vs. SDB

4.2.2 SDB Cost Breakdown

We further break down the components of SDB cost as follows. Figure 18 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 19 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.

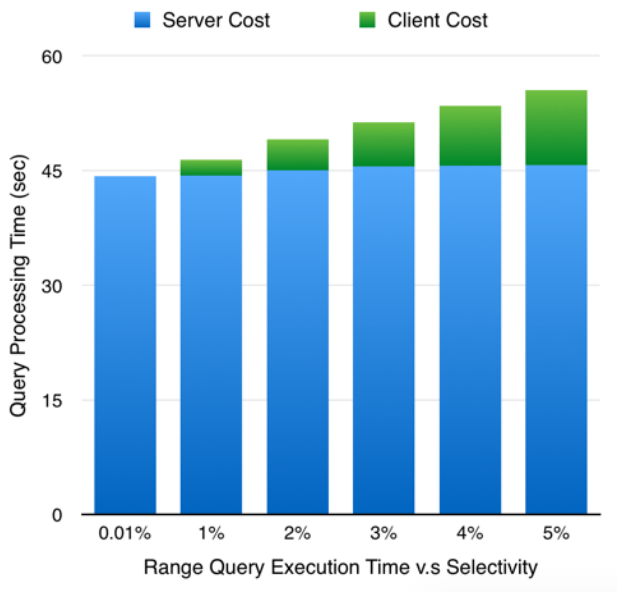

Figure 18: Client/Server Cost v.s. Selectivity

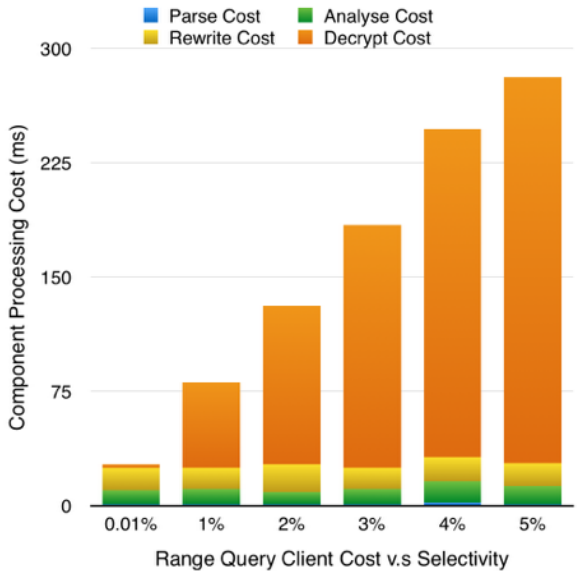

Figure 19: Client Costs v.s. Selectivity

4.2.3 SDB vs. SparkSQL

To answer *Q2*, we evaluate the performance of SDB 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 SDB, and use `TSDB` to denote the execution time. The radio `TSDB/TSparkSQL` captures the degree to which SDB's encryption and secure protocol slows down the entire query processing time.

We observe that SDB introduces a 170 times slow-down 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.

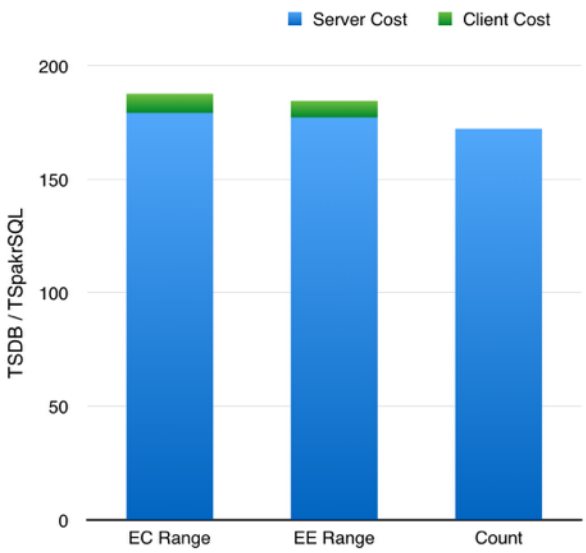

Figure 20: T_{SDB} / T_{Spark}

Finally, Figure 18 shows the client/server cost for the three queries with different data size.

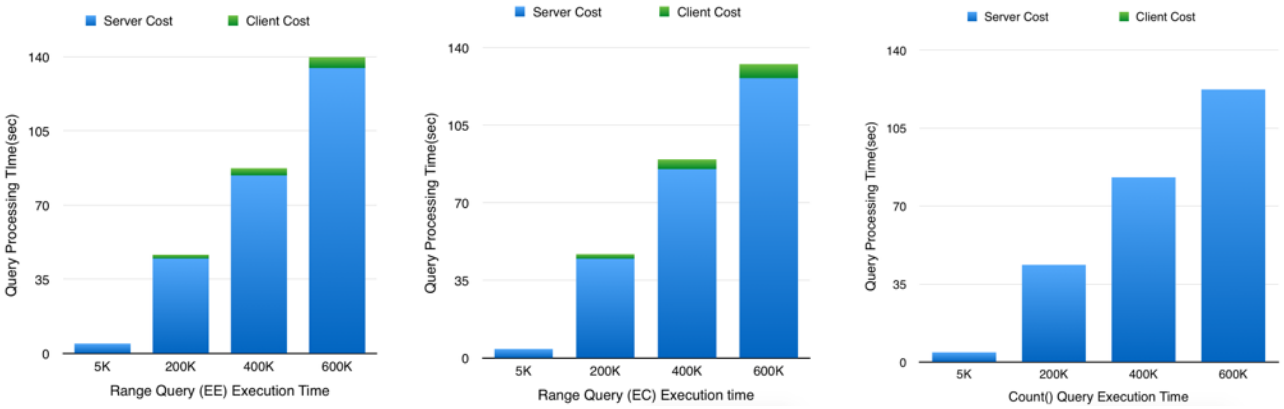

Figure 21 Client/Server Cost vs. Data Size

5. Conclusion and Future Works

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

5.1 Secure Operator Extension

SDB is designed to support a wide range of query with several secure operators. In Appendix 7.1, we included the query rewriting rules for the following secure operators which are not fully implemented yet:

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

Since the query executor of SDB proxy is implemented with the *iterator* pattern, SDB 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, SDB proxy shall identify the parts of query supported at cluster server and push them to server as much as possible. Then SDB proxy performs the rest of the queries.

5.2 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 $mb^r \bmod n$ with $b = g^x$. 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 b^1, b^2, b^4, b^8, b^{16}, … To compute b^{13}, we only need to retrieve b^1, b^2, b^8 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.

5.3 Conclusion

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

6. 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, and Nigel P. Smart. "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* (pp. 563-574). ACM.

[4] Bajaj, Sumeet, and Radu Sion. "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, et al. "Cryptdb: protecting confidentiality with encrypted query processing." *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011.

[6] Tu, S., Kaashoek, M. F., Madden, S., & Zeldovich, N. (2013, March). *Processing analytical queries over encrypted data. In Proceedings of the VLDB Endowment* (Vol. 6, No. 5, pp. 289-300). VLDB Endowment.

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

[8] Scott (2014, February 14). Faster RSA in Java with GMP. Retrieved 19 April, 2015, from https://corner.squareup.com/2014/02/faster-rsa-jnagmp.html

[9] Hore, B., Mehrotra, S., & Tsudik, G. (2004, August). A privacy-preserving index for range queries. *In Proceedings of the Thirtieth international conference on Very large data bases-Volume 30* (pp. 720-731). VLDB Endowment.

7. Appendix

7.1 Rewrite Rules for Other Secure Operators

Function	Explanation
SIES(row_id, Ck_{ROW_ID}, ρ_1, ρ_2)	Client-side function which decrypts row_id with SIES
sdb_int_add(row_id1, row_id2, n)	Server-side row-id addition UDF
sdb_cartprod(A_e, S_{e2}, p_A, n)	Server-side cartesian product (A ⊗ B as C) UDF

Secure operation in EP mode is similar to EC mode that plaintext is treated as a column with column key <1, 0>.

SELECT A x P as C FROM T

$ck_{PN} = < rand(\rho_1, \rho_2), rand(\rho_1, \rho_2) >$
$\kappa_{ZPN} = key\_update\_client(ck_Z, ck_{PN}, ck_S, \rho_1, \rho_2)$
$ck_C = <m_A \times m_{PN}, x_A + x_{PN} >$

SELECT row_id, sdb_mul(A_e, sdb_key_update(P, S_e, κ_{ZPN}, n), n) AS C_e FROM T

Rule 8: Rewrite Rule for Multiplication (EP Mode)

SELECT (A+P) as C FROM T

$ck_{PN} = < rand(\rho_1, \rho_2), rand(\rho_1, \rho_2) >$
$\kappa_{ZPN} = key\_update\_client(ck_Z, ck_{PN}, ck_S, \rho_1, \rho_2)$
$ck_{PN} = < rand(\rho_1, \rho_2), rand(\rho_1, \rho_2) >$
$\kappa_{AC} = key\_update\_client(ck_A, ck_C, ck_S, \rho_1, \rho_2)$
$\kappa_{PNC} = key\_update\_client(ck_{PN}, ck_C, ck_S, \rho_1, \rho_2)$

SELECT row_id, sdb_add(A_e, sdb_key_update(P, S_e, κ_{ZPN}, n) ,S_e, κ_{AC}, κ_{PNC}, n) AS C_e FROM T

Rule 9: Rewrite Rule for Addition (EP Mode)

SELECT (A-P) as C FROM T

$ck_{PN} = < rand(\rho_1, \rho_2), rand(\rho_1, \rho_2) >$
$\kappa_{ZPN} = key\_update\_client(ck_Z, ck_{PN}, ck_S, \rho_1, \rho_2)$
$ck_{PNi} = < (n-1) \times m_{PN}, x_{PN} >$ *// inverse the value of P*
$ck_C = < rand(\rho_1, \rho_2), rand(\rho_1, \rho_2) >$
$\kappa_{AC} = key\_update\_client(ck_A, ck_C, ck_S, \rho_1, \rho_2)$
$\kappa_{PNiC} = key\_update\_client(ck_{PNi}, ck_C, ck_S, \rho_1, \rho_2)$

SELECT row_id, sdb_add(A_e, sdb_key_update(P, S_e, κ_{ZPN}, n) S_e, κ_{AC}, κ_{PNiC}, n) AS C_e FROM T

Rule 10: Rewrite Rule for Subtraction (EP Mode)

SELECT * FROM T WHERE A > P

$ck_{PN} = < rand(\rho 1, \rho 2), rand(\rho 1, \rho 2) >$
$\kappa_{ZPN} = key\_update\_client(ck_Z, ck_{PN}, ck_S, \rho 1, \rho 2)$
$ck_{PNi} = < (n-1) \times m_{PN}, x_{PN} >$
$ck_C = < rand(\rho 1, \rho 2), rand(\rho 1, \rho 2) >$
$\kappa_{AC} = key\_update\_client(ck_A, ck_C, ck_S, \rho 1, \rho 2)$
$\kappa_{PNiC} = key\_update\_client(ck_{PNi}, ck_C, ck_S, \rho 1, \rho 2)$
$ck_{RC} = < m_R \times m_C, x_R + x_C >$
$\kappa_{RCZ} = key\_update\_client(ck_{RC}, ck_Z, ck_S, \rho 1, \rho 2)$

SELECT * FROM T
WHERE $sdb\_key\_update(sdb\_mul(R_e, sdb\_add(A_e, sdb\_key\_update(P, S_\cdot, ck_{PN}, n), S_e, ck_{AC}, ck_{PNiC})), S_e, \kappa_{RCZ}, n) =/> 0$

Rule 11: Rewrite Rule for Comparison (EP Mode)

For secure Cartesian product between two tables, we make use of the additive homomorphic property of row_id's encryption (SIES) so that T1 JOIN T2 can be decrypted with the new row_id.

SELECT A, B FROM T1 JOIN T2

$p_A = (x_{s2})^{-1} \times x_A \bmod \Phi(n)$
$ck_{A'} = < m_A \times (m_{S2})^{p_A}, x_A >$
$p_A = (x_{s1})^{-1} \times x_B \bmod \Phi(n)$
$ck_{B'} = < m_B \times (m_{S2})^{p_B}, x_B >$

SELECT $sdb\_int\_add(T1.row\_id, T2.row\_id, n)$ as row_id, $sdb\_cartprod(A_e, S_{e2}, p_A, n)$, $sdb\_cartprod(B_e, S_{e1}, p_B, n)$ FROM T1 JOIN T2

Rule 12: Rewrite Rule for Cartesian Product

To perform secure aggregation function SUM(), we perform a key update operation on target column A with a special column key < rand, 0 >. Particularly, we insert with a different row_id in case the result of SUM() is is referred in other nested queries. Finally, The decryption of SUM ciphertext is handled differently by equation $[\![C]\!] = m_z \times C_e$.

SELECT sum(A) as C FROM T

$Ck_{SUM} = < rand(\rho 1, \rho 2), 0 >$
$\kappa_{ASUM} = key\_update\_client(ck_A, ck_{SUM}, ck_S, \rho 1, \rho 2)$
$ck_{PNi} = < (n-1) \times m_{PN}, x_{PN} >$
$row\_id = rand(\rho 1, \rho 2)$
$enc\_row\_id = SIES(row\_id, Ck_{ROW\_ID}, \rho 1, \rho 2)$

SELECT enc_row_id, $sdb\_sum(sdb\_key\_update(A_e, S_e, \kappa_{ASUM}, n))$ FROM T

Rule 13: Rewrite Rule for SUM()

With computed values of SUM() and COUNT(), AVG() can be easily computed by dividing SUM with COUNT. $[\![C]\!] = m_z \times C_e$ / count.

SELECT avg(A) as C FROM T

$Ck_{SUM} = < rand(\rho_1, \rho_2), 0 >$
$\kappa_{ASUM} = key\_update\_client(ck_A, ck_{SUM}, ck_S, \rho_1, \rho_2)$
$ck_{PNi} = < (n-1) \times m_{PN}, x_{PN} >$
$row\_id = rand(\rho_1, \rho_2)$
$enc\_row\_id = SIES(row\_id, Ck_{ROW\_ID}, \rho_1, \rho_2)$

SELECT enc_row_id, sdb_sum(sdb_key_update(A_e, S_e, κ_{ASUM}, n)) as sum, count(A_e) as count FROM T

Rule 14: Rewrite Rule for AVG()

7.2 Experiment Data

	5K	200K	400K	600K
SDB Server	4.246	46.934	89.619	132.614
SDB Client	0.082	2.107	4.372	6.366
DBQ Server	0.230	8.93	17.860	26.8
DBQ Client	14.33	510.92	1003.931	1483.658

Figure 17: SELECT A, B from T WHERE A < p, 1% selectivity (seconds)

	0.01%	1%	2%	3%	4%	5%
Server Cost	44.239	44.327	45.035	45.502	45.627	45.714
Client Cost	0.071	2.107	4.048	5.791	7.789	9.835

Figure 18: SELECT A, B from T WHERE A < p, on 200K records (seconds)

	0.01%	1%	2%	3%	4%	5%
Parse Cost	1	1	1	1	2	1
Analyse Cost	9	10	8	10	14	12
Rewrite Cost	15	14	18	14	16	15
Decrypt Cost	2	56	104	159	215	253

Figure 19: SELECT A, B from T WHERE A < p, on 5K records (milliseconds)

	EC Range	EE Range	Count
Server Cost	44827	44867	43728
Client Cost	2107	1786	35
SparkSQL Cost	250	253	254

Figure 20: TSDB vs TSparkSQL on 200K records (milliseconds)

	5K	200K	400K	600K
Server Cost	4.633	44.867	84.053	134.601
Client Cost	0.14	1.786	3.414	5.358

Figure 20A: SELECT A, B from T WHERE A < B, 1% selectivity (seconds)

	5K	200K	400K	600K
Server Cost	4.164	44.827	85.247	126.248
Client Cost	0.082	2.107	4.372	6.366

Figure 20B: SELECT A, B from T WHERE A < p, 1% selectivity (seconds)

	5K	200K	400K	600K
Server Cost	4.45	43.728	82.991	122.435
Client Cost	0.017	0.035	0.021	0.021

Figure 20B: SELECT COUNT(A) from T WHERE A < p, 1% selectivity (seconds)