This document is downloaded from DR-NTU (https://dr.ntu.edu.sg)
Nanyang Technological University, Singapore.

Complex Queries in a Shared Multi User Relational Cloud Database

Sidorov, Vasily; Ng, Wee Keong

2014

Sidorov, V., & Ng, W. K. (2014). Complex Queries in a Shared Multi User Relational Cloud Database. 2013 IEEE Sixth International Conference on Cloud Computing (CLOUD), 903-909.

https://hdl.handle.net/10356/83219

https://doi.org/10.1109/CLOUD.2013.49

© 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The published version is available at: [https://dx.doi.org/10.1109/CLOUD.2013.49].

Downloaded on 25 Aug 2022 03:44:47 SGT

Complex Queries in a Shared Multi User Relational Cloud Database

Vasily Sidorov
School of Computer Engineering
Nanyang Technological University
Singapore
Email: `vasily001@e.ntu.edu.sg`

Wee Keong Ng
School of Computer Engineering
Nanyang Technological University
Singapore
Email: `awkng@ntu.edu.sg`

***Abstract*—While DaaS is becoming more and more popular enterprises start considering it as an option to reduce IT maintenance costs. But data privacy and security issues keep most of them from moving to the cloud. Existing schemes of encrypting the database usually either provide a very basic search only (e.g., `SELECT * FROM t WHERE t.a = 5`) and nothing more or give users more flexibility at a cost of data security. In this paper we try to introduce a highly secure and flexible database encryption scheme allowing multiple users to perform more complex queries, including cross-table joins and still achieve acceptable performance by utilizing database's built-in indexing capabilities. Several dedicated proxy servers in between the user and the database server help to achieve this and make it all transparent for the user. Developed solution shows acceptable performance level in most of the testing cases.**

***Keywords*-cloud database security; querying encrypted data; encrypted joins;**

I. Introduction and Related Work

Such concepts as *DaaS* or *Cloud Storage* are becoming more and more popular among corporate and individual customers. Since the corporate customers tend to have higher requirements to the security and reliability of such services we will consider only them without any loss of generality. Data storage outsourcing allows organizations to reduce drastically IT maintenance costs (which is very important to new small businesses, especially in IT) and to improve flexibility and scalability of business.

On the other hand, the risks of data theft are increasing. Moreover, not only third party ill-wishers who could try to intercept data transfer from outsourced storage to client and back should be considered. In general, we should also assume the data storage server itself untrusted, as it has full access to the data.

Encryption could guarantee data privacy along with causing a severe performance bottleneck: without applying additional efforts, database becomes no longer able to answer data queries without sending back the whole encrypted piece of data and further client-side decryption and query execution. In fact, this is not the only issue appearing when moving to an encrypted database. Most common issues are inability to perform analytical processing on data anymore, key management, difficulties with compromised encryption keys and multi-user scenarios: differentiating access to the data, enrollment of new users, revoking access.

Many papers have proposed different solutions to the problem of querying encrypted data. Some of them concern general issues of searching over the encrypted data [1]–[3]. Others are more aimed at providing solutions for querying data in a relational database [4]–[7].

C. Dong *et al.* proposed to use an RSA-based *proxy re-encryption* to give differentiated access to the data [8]. Proxy re-encryption makes it possible to keep the data encrypted with a master key and give users only a share of the master key (each user has unique share), and the complimentary share is kept by a DBMS. Shares of the master key on their own has no ability to encrypt or decrypt data, but once they are combined we can fully operate the data. Typical workflow with proxy re-encryptions does not reveal the master key neither to user nor to DBMS. This approach also gives an opportunity to easily revoke/grant access from/to users. Procedure becomes as simple as create two shares (one for a new user and one for the proxy) of a master key to grant access or to instruct proxy to destroy a share of master key corresponding to certain user to revoke his access.

Later in 2006 Z. Yang *et al.* proposed a specific way to store redundant data for searching purposes along with the ciphertexts of the data values themselves [9]. The sole idea of storing additional data for searching purposes was first proposed by H. Hacıgümüş *et al.* [10], which is one of the most important papers in this field. Evolution of proxy re-encryption introduced by C. Dong [8] in combination with ideas by Z. Yang [9] was later used by N. T. Hung, D. H. Giang *et al.* in one of the most recent works on the topic [6].

Approach proposed by N. T. Hung *et al.* maintains several interesting and useful properties for the encrypted database. First, it supports easy enrollment and revocation of users and dealing with compromised keys without having to re-encrypt data in the database. Second, the system consists of

several independent parts (user, proxies, key manager and database server) no one of which (except for key manager) knows in full the secret needed to decrypt data — it is shared between all of them and since they are independent they could easily be hosted at different physical locations (e.g., at different cloud server providers) and unless all parties maliciously collude data is safe. The key manager is only needed when a new user is enrolled and could be kept offline most of the time. Third, proposed solution is flexible and allows database administrator to fine-tune security–performance trade-off for every user separately. In addition, in case keys were somehow compromised, solution gives an opportunity to easily re-encrypt all data right in the database *without decrypting* it. Lastly, proposed scheme conceals which cells in the database contain ciphertexts for equal plaintext values, i.e., two ciphertexts in the database for same original value are different, which blocks a range of attacks like statistical attack.

However, the system also has several limitations. The proposed design only allowed users to search for constant data in the database, not to match different attributes of a table or perform table equi-joins[1].

Our contribution made in this work is a database encryption scheme, which maintains the security properties listed above but at the same time has a much wider capabilities: support of database's built-in indexing mechanism; ability to perform basic data search (`... WHERE t.col = 3`) as well as more complex queries (`... WHERE t.col1 = t.col2`) and even joins; ways to do fine-grained adjustments between security and performance individually for each column.

The paper is organized as follows: Section II fully describes all parts of the system and their interactions; Section III provides information on how to construct and run different types of queries in this system and how to interpret query results; Section IV provides a security analysis of the proposed system design; Section V contains a comprehensive comparison of the proposed system in various setups with a conventional plaintext database; Section VI describes how system executes operations other than querying; Section VII summarizes the contribution made in this paper.

II. System Model Description

We assume that database consists of 2 tables A and B with attributes A_i and B_i respectively. Each cell of each table keeps a secure representation of plaintext value $T^*_{x,y}$[2], where x is a row number, y is an attribute index.

The general idea is that along with encrypted value database stores a *fingerprint* (in a form of bit array) of a plaintext value. When we need to do matching with a constant value or with another attribute we simply match their fingerprints. If they are not equal then the underlying plaintexts are definitely not equal also. If they match then underlying plaintexts are *probably* equal. This means that upon retrieval the response contains some amount of false positives, which are to be filtered on a client side (see Sections V and IV for false positive rate estimations).

A. Parts of the System

Key manager: KM generates and allocates keys to users and proxies so they are able to follow the protocols needed for the system to operate. Most of the time KM could and should be kept offline. It is needed online only during the system setup and then every time a new user is enrolled in the system.

Data owner: Data owner knows which information is more sensitive and which is less and thus sets the data sensitivity level $m^*_i \leq m$ for every column in every table. This information is public.

Set of proxies: There's a set of z proxies. Each proxy keeps a specific share of secret keys s and q, individual for every user. So, for the ith user the jth proxy keeps $s^P_{i,j}$ and $q^P_{i,j}$.

Hashing server: This is a special kind of a proxy. It is present in the system only once and is purposed to compute data fingerprints both on stages of *data encryption* and *query construction*.

Cloud database server: A relational DBMS hosted by a third-party cloud service provider. Users share the database and are able to perform simultaneous reads and writes. Since the DBMS is hosted somewhere outside the organization all sensitive data should be kept encrypted in the DB server.

User: Here in this paper we do not separate users and their client applications, which they use to access the cloud database; word "user" is therefore used to describe both application and user himself.

B. Trust and Attack Model

In our scheme we assume the key manager is fully trusted.

The database server, proxies and users are modeled as semi-honest. They follow all the protocols but are passively curious — they try to learn as much information as possible about the data stored database as well as users' queries without initiating unauthorized actions. In addition, when users leave the system we do not expect them to forget or to keep their keys in secret.

C. System Construction

1) System Setup: Key manager sets up a cyclic group G of large prime order p, w is the generator of G. KM also chooses a random master storage key s and a query key q. Additionally, KM chooses length of *fingerprint* in bits m (usually 16, 32 or 64) and generates m different keys $k_1, \ldots, k_m$ for hashing purposes. KM also selects a cryptographic hashing function $f(x, k)$ with image $[1, m] \cap \mathbb{N}$.

[1]Equi-join — join based on pairs of attributes from different tables being equal.

[2]$T^*_{x,y}$ stands for $T^A_{x,y}$ or $T^B_{x,y}$

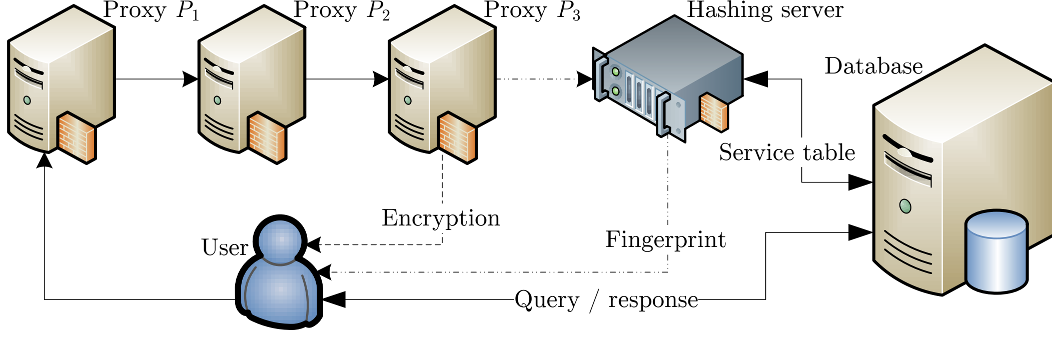

Figure 1. System architecture

KM publicizes (G, p, w, m). Hashing keys $k_1, \ldots, k_m$ and hashing function $f(x, k)$ are to be transferred securely to the hashing server. Keys s and q are kept secret.

2) Data Model Annotation: Before first user enrolls in the database, data owner should annotate data scheme. As mentioned earlier, we assume that database consists of two tables A and B, each has attributes A_i and B_i. All procedures described in this paper could easily be extended to any bigger amount of tables.

In addition to these tables DB also stores a public unencrypted service table S, containing a data sensitivity level m_i^* for every attribute, e.g., for attribute B_3 its data sensitivity level is m_3^B. Data sensitivity level refers to the amount of different hash functions used to generate fingerprints (which are constructed similar to Bloom filters; see Section II-C4 for details). The most sensitive information has data sensitivity level 1, least sensitive information has data sensitivity level m.

3) User Enrollment: When a new user U_i is enrolled in the system, KM generates shares s_i and q_i of secret keys s and q respectively for user and proxies. These shares are then securely transferred to respective parties. User U_i receives secret shares s_i^U and q_i^U. For every user enrolled in the system each proxy keeps corresponding shares of the keys; so for user U_i the jth proxy keeps $s_{i,j}^P$ and $q_{i,j}^P$, and following holds:

$$\begin{aligned} s &= s_i^U + s_{i,1}^P + \ldots + s_{i,z}^P = s_i^U + \sum_{j=1}^{z} s_{i,j}^P \\ q &= q_i^U + q_{i,1}^P + \ldots + q_{i,z}^P = q_i^U + \sum_{j=1}^{z} q_{i,j}^P \end{aligned}$$

4) Data Encryption: As we said earlier, along with encrypted value database stores a *fingerprint* of data. So, for every plaintext value $T_{x,y}^*$ database stores $\left\langle T_{x,y}^{\prime *}\langle 1\rangle, T_{x,y}^{\prime *}\langle 2\rangle \right\rangle$, where $T_{x,y}^{\prime *}\langle 1\rangle$ is the original value ciphertext and $T_{x,y}^{\prime *}\langle 2\rangle$ is the fingerprint.

To compute $T_{x,y}^{\prime *}\langle 1\rangle$ user U_i chooses a random number $r \in Z_p^*$ and computes $c_1 = T_{x,y}^* \times w^{rs_i^U}$ and $c_2 = w^r$. He sends c_2 to the proxy P_1. Upon receiving a value c_2 proxy operating in encryption mode computes $c_2' = c_2^{s_{i,j}^P}$ and sends it to the next proxy in chain, e.g., P_1 sends to P_2. For example, proxy P_1 computes $c_{2,j}' = c_2^{s_{i,j}^P} = w^{rs_{i,j}^P}$. Last proxy in the chain — P_z — sends the result $c_r = c_2^{\sum_{j=1}^{z} s_{i,j}^P} = w^{r\sum_{j=1}^{z} s_{i,j}^P}$ to the user. After receiving c_r from P_z user computes:

$$\begin{aligned} c_1' &= c_1 \times c_r = T_{x,y}^* \times w^{rs_i^U} \times w^{r\sum_{j=1}^{z} s_{i,j}^P} \\ &= T_{x,y}^* \times w^{r\left(s_i^U + \sum_{j=1}^{z} s_{i,j}^P\right)} = T_{x,y}^* \times w^{rs} \end{aligned}$$

After computing c_1', user sends (c_1', c_2) to the database server, which stores this pair as $T_{x,y}^{\prime *}\langle 1\rangle = (c_1', c_2) = \left(T_{x,y}^* \times w^{rs}, w^r\right)$.

The fingerprint $T_{x,y}^{*\prime}\langle 2\rangle$ of the value, which will be later used to perform value look-ups and joins, is constructed in a similar pipeline starting with user, going through all proxies (now operating in fingerprint construction mode) and ending with hashing server. First, user computes $b = T_{x,y}^* \times w^{q_i^U}$. He then passes b to proxy P_1, which computes $b_1 = b \times w^{q_{i,1}^P} = T_{x,y}^* \times w^{q_i^U + q_{i,1}^P}$. Proxy P_1 then transfers b_1 to proxy P_2 which in its turn computes $b_2 = b_1 \times w^{q_{i,2}^P}$ and so on until last proxy P_z finally computes $b_z = b_{z-1} \times w^{q_{i,z}^P} = T_{x,y}^* \times w^q$. This value b_z is then transferred to hashing server.

From this point, construction of the fingerprint is similar to creation of a Bloom filter (e.g., see [3, p. 5] or [2, p. 7]). First, hashing server refers to the service table S in the database and retrieves corresponding data sensitivity level m_y^*. Then it creates a bit array R of length m and initializes it with zeros. After that $f(b_z, k_1), \ldots, f(b_z, k_{m_y^*})$ are computed and the set of resulting values is used as indices in R where 0s are replaced with 1s, i.e., we perform `R[`$f(b, k_1)$`] := 1; ...; R[`$f(b, k_{m_y^*})$`] := 1;`. Some indices could coincide — we do not do anything about that. Eventually we obtain a bit array R of length m with 1s on certain places. This array is a *fingerprint* of the original value, which is then returned to the user, who sends it to the cloud database to be stored as $T_{x,y}^{\prime *}\langle 2\rangle$.

III. Query Construction

A. Constant Match

First subset of supported queries is (e.g., for a table A): `SELECT` *columns* `FROM` A `WHERE` $A_y = v$, where v is a constant value. To construct a query user U_i should obtain a fingerprint R_v for value v in a manner similar to described in *Data Encryption*. The sensitivity level should be retrieved for the queried attribute (A_y in this example). After that the original query should be rewritten in the following way: `SELECT` *columns* `FROM` A `WHERE` $T_y'^A \langle 2 \rangle = R_v$. Of course, for table B everything is the same.

The mechanism of signatures guarantees that if $T_{x,y}^* = v$ then $T_{x,y}'^* \langle 2 \rangle = R_v$. On the other hand, this does not work backwards, i.e., if $T_{x,y}'^* \langle 2 \rangle = R_v$ it does not mean that $T_{x,y}^* = v$. This helps to protect the information about cells with the same plaintext values but enlarges data transfer overhead since extra amount of tuples is transferred in a query result to a client, who then needs to filter false positives. Rate of false positives could be adjusted with data sensitivity level m_y^* — the lower it is, the bigger amount of false positives is, but information about coinciding data is more secure, and vice versa.

B. Columns Match

Another subset of supported queries is the one which involves matching values of two different columns on the same table A (or B): `SELECT` *columns* `FROM` A `WHERE` $A_{y_1} = A_{y_2}$. We are considering two possible scenarios: $m_{y_1}^A = m_{y_2}^A$ and $m_{y_1}^A \neq m_{y_2}^A$.

1) $m_{y_1}^A = m_{y_2}^A$*:* This scenario is very straightforward and similar to matching with constant value, it is enough to rewrite original query in a following way: `SELECT` *columns* `FROM` A `WHERE` $T_{y_1}'^A \langle 2 \rangle$ = $T_{y_2}'^A \langle 2 \rangle$. Additional benefit of the proposed scheme is that both in case of matching two columns with equal sensitivity levels and in case of matching with a constant we can utilize a built-in indexing capabilities of the database.

2) $m_{y_1}^A \neq m_{y_2}^A$*:* This scenario is slightly more complicated since fingerprints for two columns we are trying to match were built with different sets of hashing functions. To be specific, let us assume that $m_{y_1}^A > m_{y_2}^A$. To construct a fingerprint $T_{x,y_1}'^A \langle 2 \rangle$ for A_{y_1} a hashing function f was computed with keys $k_1, \ldots, k_{m_{y_1}^A}$; to construct a fingerprint $T_{x,y_2}'^A \langle 2 \rangle$ for A_{y_2} a hashing function f was computed with keys $k_1, \ldots, k_{m_{y_2}^A}$. Therefore, keys $k_1, \ldots, k_{m_{y_2}^A}$ were used in construction of both fingerprints, but $T_{x,y_1}'^A \langle 2 \rangle$ also used several more keys. This means that if there is a "0" in $T_{x,y_1}'^A \langle 2 \rangle$ and a "1" in $T_{x,y_2}'^A \langle 2 \rangle$ on the same position, then we can say definitely that $T_{x,y_1}^A \neq T_{x,y_2}^A$. Otherwise, they could match and server should return this tuple.

Thus the original query should be rewritten in a following way: `SELECT` *columns* `FROM` A `WHERE` $T_{x,y_1}'^A \langle 2 \rangle$ `&` $T_{x,y_2}'^A \langle 2 \rangle = T_{x,y_2}'^A \langle 2 \rangle$, where `&` denotes a bitwise `AND` operation.

The proposed scheme is rather flexible since it allows database manager to fine-tune the security–performance trade-off for every single column and still leave users the ability to perform matches between these columns. The only issue here is the inability to rely on a database indexing mechanisms in case $m_{y_1}^A \neq m_{y_2}^A$, since the `WHERE` predicate contains a bitwise operation.

C. Equi-Joins

Equi-join is a query of the type `SELECT` *columns* `FROM` A, B `WHERE` $A_{y_1} = B_{y_2}$. It is quite obvious, that the described in Section III-B mechanism of matching different columns does not rely on the columns being in same table and thus makes it possible to perform equi-joins without additional efforts.

D. Result Processing and Decryption

Assume the result to user U_i's query is a set of n records (tuples) $J = \{J_{i_1}, J_{i_2}, \ldots, J_{i_n}\}$, each of them consists of c ciphertexts $T_{x,y}'^* \langle 1 \rangle = \left(T_{x,y}^* \times w^{r_{x,y} s}, w^{r_{x,y}}\right)$, where $x \in \{i_1, \ldots, i_n\}$, $y \in \{j_1, \ldots, j_c\}$. Server sends this result to user U_i. User U_i then sends each ciphertext $T_{x,y}'^* \langle 1 \rangle$ to the chain of proxies. Each proxy P_j in the chain computes $t_j = w^{r_{x,y} s_{i,j}^P}$, multiplies encrypted value by t_j^{-1} and sends result to the next proxy in the chain. Last proxy in the chain — P_z — sends result back to user. The value received by user is in fact $T_{x,y}^* \times w^{r_{x,y} s_i^U}$. User then multiplies received value by $\left(w^{r_{x,y} s_i^U}\right)^{-1}$ and obtains cleartext value $T_{x,y}^*$.

After that when user has decrypted ciphertexts and knows cleartext values he needs to do the final filtering and remove from the query results those records which do not really satisfy the `WHERE` predicate in the query.

IV. Security Analysis

$T \langle 2 \rangle$ is used to search certain values among data in the database or to match columns with each other without revealing to anybody (should it even be a database administrator) which exactly cells of the database contain equal values. Database administrator could adjust data sensitivity level from 1 to m, where 1 corresponds to most sensitive data and m — to most insensitive data.

As it could be easily seen, during the construction of a fingerprint (which in fact appears to be a hash of a ciphertext of an original value encrypted with query key q) almost all information about original value is destroyed and adversary is incapable to reconstruct original value from it. Moreover, constructed fingerprint is generally a *good* hash function as defined by RFC4949 [11, p. 139]. This means that two values having same fingerprint tells adversary nothing about original values. Of course this depends on a selected hashing function $f(x, k)$ in every particular case and certain hashing

function $f(x,k)$ should be formally proved to deliver a good fingerprint before using.

As the very process of constructing $T\langle 2\rangle$ suggests, for a sensitivity level $m_i \in [1,m]$ the whole domain of a value is uniformly divided into $\sum_{j=1}^{m_i}\binom{m}{j}$ buckets[3]. This means that if a value domain contains d values (e.g., for a 32-bit integer $d = 2^{32} = 4\ 294\ 967\ 296$), amount of values in each bucket is

$$v = \frac{d}{\sum_{j=1}^{m_i}\binom{m}{j}}.$$

E.g., for a 32-bit value in a column with sensitivity 1 $v \approx 2.7\times 10^8$, and in a column with sensitivity 16 $v \approx 6.5\times 10^4$ (m is assumed to be 16).

V. Empirical Results

As long as current work leaves all the procedures of generating and decrypting of $T\langle 1\rangle$ unchanged as compared to [6] and mainly focuses on proposing a new approach to work with $T\langle 2\rangle$, we only were doing performance benchmarks of different scenarios of data retrieval and were not involving actual encryption or decryption of data.

Proposed model of working with $T\langle 2\rangle$ was implemented and various performance tests were conducted. Implementation was written in C# and consists of 3 parts: proxy server, hashing server and program emulating user activity, which communicated through network. MySQL v5.5.28 x64 for Windows was used as a database server for tests. Tests were run on following configuration: quad-core Intel Core i5 2.67 GHz, 4 GB RAM, Windows 7 Enterprise SP1 x64.

Fingerprint length m was set to 16 bit. Hashing function was constructed using HMAC based on MD5. It produces a 32 digit hexadecimal number, whose remainder from division by $m = 16$ is then used as an index of the bit to be set to 1 in the resulting fingerprint.

Database consisted of tables `t1` and `t2`. Table `t1` consisted of 4 columns `a_1 INT`, `a_2 SMALLINT`, `b_1 INT`, `b_2 SMALLINT`. Table `t2` consisted of 4 columns `c_1 INT`, `c_2 SMALLINT`, `d_1 INT`, `d_2 SMALLINT`. Index was built for columns `t1.a_1`, `t1.a_2`, `t2.c_1` and `t2.c_2`. Columns which have "_1" in their names contain original plaintext value, columns which have "_2" contain fingerprints and represent $T\langle 2\rangle$.

Tests were run in several stages, each with different sets of sensitivity values for each column. Each table was populated with 30 000 rows with uniformly random integer values from $[-500; 500] \cap \mathbb{N}$ and their fingerprints created with respect to corresponding sensitivity (i.e., using corresponding amount of different keys for hashing). Each test stage consisted of following tests:

[3] $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

Stage	Sensitivity values			
	`t1.a`	`t1.b`	`t2.c`	`t2.d`
1	1	1	1	1
2	2	2	2	2
3	4	4	4	4
4	8	8	8	8
5	16	16	16	16
6	1	1	16	16
7	2	2	16	16
8	4	4	16	16
9	8	8	16	16
10	15	15	16	16

Table I
System configuration sets used for testing

1) Constant matching on a column with index (`t1.a`):
`SELECT SQL_NO_CACHE`[4] `COUNT(*) FROM t1 WHERE t1.a =` *const*
2) Constant matching on a column without index (`t1.b`):
`SELECT SQL_NO_CACHE COUNT(*) FROM t1 WHERE t1.b =` *const*
3) Intra-table column matching (i.e., `t1.a = t1.b`):
`SELECT SQL_NO_CACHE COUNT(*) FROM t1 WHERE t1.a = t1.b`
4) Inter-table column matching (join) on two columns with index (`t1.a = t2.c`):
`SELECT SQL_NO_CACHE COUNT(*) FROM t1,t2 WHERE t1.a = t2.c`
5) Inter-table column matching (join) on two columns, only one of which has index (`t1.a = t2.d`):
`SELECT SQL_NO_CACHE COUNT(*) FROM t1,t2 WHERE t1.a = t2.d`
6) Inter-table column matching (join) on two columns, none of which has index (`t1.b = t2.d`):
`SELECT SQL_NO_CACHE COUNT(*) FROM t1,t2 WHERE t1.b = t2.d`

Each of them runs repeatedly, tests 1 and 2 — 150 times, tests 3–6 — 30 times for first 5 test stages and only tests 4–6 are run for the second 5 stages, each is repeated 20 times. Test stages were conducted with sensitivity values for each column shown in Table I.

Tests were conducted in order to estimate two performance metrics: rate of false positives (it was estimated as amount of *all* rows returned by a secure query divided by amount of rows which *should* have been returned, i.e., returned by plaintext version of the query — as it was mentioned earlier, client should filter out those rows that actually do not comply with `WHERE` predicate) and query execution time as compared to executing same queries on

[4] Query results caching disabled by specifying `SQL_NO_CACHE`. This is MySQL-specific.

Stage	1	2	3	4	5	1	2	3	4	5
Test	False positives ratio					Time ratio				
`t1.a = const`	65.573	8.667	1.253	1.007	1.027	3.094	1.232	1.014	0.979	0.979
`t1.b = const`	66.86	8.453	1.227	1.02	1.02	0.894	0.936	0.898	1.11	0.863
`t1.a = t1.b`	72	7	1	1	1	1	0.99	0.97	1	0.992
`t1.a = t2.c`	62	8	1	1	1	54.294	7.447	1.302	0.993	0.985
`t1.a = t2.d`	62	8	1	1	1	52.337	7.244	1.29	0.956	0.981
`t1.b = t2.d`	62	8	1	1	1	0.997	0.958	0.96	0.952	0.955

Stage	6	7	8	9	10	6	7	8	9	10
Test	False positives ratio					Time ratio				
`t1.a = t2.c`	639	419	183	42	5	74.276	69.81	66.614	64.314	63.81
`t1.a = t2.d`	640	420	183	42	5	86.534	76.005	74.073	68.319	66.962
`t1.b = t2.d`	639	420	183	42	5	1.565	1.499	1.345	1.242	1.216

Table II
PERFORMANCE EVALUATION RESULTS

plaintext values also obtained as a ratio of execution times of secure and plaintext queries. Results are calculated as average among several repetitions of the test and appeared to be as shown in Table II.

Data in the tables shows that as it was expected amount of false positives lowers with increase of the number of different hash functions used (which corresponds to lower security[5]) and eventually become negligible.

As for the query execution time, for test stages 1–5 we can see that in most cases query execution time is relatively same for plaintext version and for secure version. Exceptions are test `t1.a = const` on stage 1, where secure version is 3 times slower and joins `t1.a = t2.c` and `t1.a = t2.d` on stages 1 and 2. All three cases involve one indexed column `t1.a` (in plain query it's `t1.a_1` and in secure query it's `t1.a_2`), so bigger execution time is most likely caused by much bigger amount of returned rows (especially for joins, where there are up to 60 times more rows to return). In cases where indexed columns are not involved at all query execution times are virtually same on all stages.

For test stages 6–10 we can see that for tests involving indexed column query execution time is constantly much bigger. This is obviously caused by the fact that when sensitivity levels are different for two columns involved in matching in `WHERE` predicate, this predicate is transformed and uses bitwise operations which blocks database from using index in any way. For the tests with no indexed columns execution time is relatively same for secure and plain queries, small difference is caused by data overhead and by slightly more complex `WHERE` predicate in a secure version.

[5]See Section IV for more details

VI. OTHER CONSIDERATIONS

A. User Revocation

As long as user can not query data or correctly encrypt data for updates or inserts without a help from proxies, revocation of user U_i is done by a KM simply by instructing all proxies P_j to remove corresponding key shares $s_{i,j}^P$ and $q_{i,j}^P$. Even if user will gain access to the database he won't be able to generate a meaningful query.

B. Key renewal

Even though compromising key is much harder as it requires collusion of all parties, it still can not be neglected. Storage and query master keys s and q could be theoretically obtained directly from KM. Thus both these keys should be updated from time to time. Procedure for updating a storage key s did not change and could be performed in the database without need to decrypt data using proxy re-encryption scheme similar to the ones proposed in [8], [12]. It is described in details in [6, Section VI.C].

Modified scheme is not capable of changing a query key q in case it is compromised as effective as it can change storage key s, though. The only way to change query key q is to generate new fingerprints for all data which could be quite time-consuming for large databases. On the other hand, loss of the query key q would leak much less information to the adversary than loss of a storage key s. In fact, the query key itself would not reveal anything to the adversary. In case adversary also knows a hashing function $f(x, k)$ and hashing keys $k_1, \ldots, k_m$ (or has access to hashing server) he can calculate sets of plaintext values corresponding to a certain fingerprint and thus to all cells having this fingerprint (taking into account sensitivity level).

Apart from query key, hashing keys $k_1, \ldots, k_m$ should also be changed from time to time. The process of changing hashing keys does not differ much from process of query key changing and needs recalculating of all fingerprints.

Practical implementation could partly alleviate process of the query key renewal by changing query key for every column separately, one at a time (in a transaction), starting with those keeping the most sensitive data. During the procedure of changing a query key the system should support both new and old query keys for those columns which are already updated and those which are still waiting in a queue, respectively. Same could be done for hashing keys renewal. It would save a lot of time if query key renewal and hashing keys renewal are combined and performed as a single task.

VII. Conclusions

In this paper we presented an improvement to the scheme proposed in [6] making it possible to perform cross-column matching and cross-table equi-joins. Our modification still preserves nearly all of the good properties of the original approach and suggests new important features.

Presented solution allows us to use several proxies in order to make collusion attack much harder and also assign different amount of proxies to groups of users with different levels of trust in order to increase performance for more trusted users and reduce system load. Additionally each column in the database now has its own level of security, which gives us an ability to improve performance for insensitive data and still have high security level for sensitive data.

Enrollment of new users and, what is even more important, revocation of access from existing users is simple and straightforward and was not changed with the improvements suggested in this paper.

Ability to easily renew storage key was also preserved. We sacrificed an ability to renew query key with same ease though. It was done in order to gain capability to perform cross-column matching and equi-joins.

Conducted benchmarks showed acceptable performance in most scenarios with matching constants or columns of same sensitivity level as compared to plaintext database querying both time-wise and FP-wise. Matching of columns with different sensitivity levels is expectedly far behind plaintext matching since database is unable to utilize indices. If indices are not built the difference in query execution time between secure and plaintext versions is within the acceptable range. The rate of false positives is pretty high regardless of indices, so the database administrators should consider raising sensitivity level of less sensitive column to the same level with the more sensitive column if matching between them is expected to occur frequently.

A security analysis shows that proposed system provides required level of security and estimates a value essential for false-positive rate — an amount of values from data domain having same fingerprint.

References

[1] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Security and Privacy, 2000. S P 2000. Proceedings. 2000 IEEE Symposium on*, 2000, pp. 44 –55.

[2] H. Hacıgümüş, B. Hore, B. Iyer, and S. Mehrotra, "Search on Encrypted Data," *Secure Data Management in Decentralized Systems*, pp. 383–425, 2007.

[3] E. Goh, "Secure indexes," *Cryptography ePrint Archive, Report*, vol. 216, 2003.

[4] F. Bao, R. Deng, X. Ding, and Y. Yang, "Private Query on Encrypted Data in Multi-user Settings," in *Information Security Practice and Experience*, ser. Lecture Notes in Computer Science, L. Chen, Y. Mu, and W. Susilo, Eds., vol. 4991. Springer Berlin Heidelberg, 2008, pp. 71–85. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-79104-1_6

[5] Y. Yang, X. Ding, R. Deng, and F. Bao, "Multi-User Private Queries over Encrypted Databases," *Int. J. High Performance Computing and Networking*, vol. 1, no. 1/2/3, pp. 64–74, 2008.

[6] N. T. Hung, D. H. Giang, N. W. Keong, and H. Zhu, "Cloud-Enabled Data Sharing Model," *Intelligence and Security Informatics (ISI), 2012 IEEE International Conference on*, 2012.

[7] R. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Processing Queries On An Encrypted Database," *Communications of the ACM*, vol. 55, no. 9, pp. 103–111, 2012.

[8] C. Dong, G. Russello, and N. Dulay, "Shared and searchable encrypted data for untrusted servers," *Journal of Computer Security*, vol. 19, no. 3, pp. 367–397, 2011.

[9] Z. Yang, S. Zhong, and R. Wright, "Privacy-preserving queries on encrypted data," *Computer Security–ESORICS 2006*, pp. 479–495, 2006.

[10] H. Hacıgümüş, B. Iyer, C. Li, and S. Mehrotra, "Executing SQL over encrypted data in the database-service-provider model," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '02. New York, NY, USA: ACM, 2002, pp. 216–227. [Online]. Available: http://doi.acm.org/10.1145/564691.564717

[11] R. Shirey, "Internet Security Glossary, Version 2," RFC 4949 (Informational), Internet Engineering Task Force, Aug. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4949.txt

[12] G. Russello, C. Dong, N. Dulay, M. Chaudron, and M. van Steen, "Providing data confidentiality against malicious hosts in Shared Data Spaces," *Science of Computer Programming*, vol. 75, no. 6, pp. 426–439, 2010.