PERFORMANCE-CONSCIOUS KEY MANAGEMENT IN ENCRYPTED DATABASES

Hakan Hacigümüs and Sharad Mehrotra

Abstract In this paper, we investigate the key management issues in encrypted database environments. We study the issues in the context database-as-a-service (DAS) model that allows organizations to outsource their data management infrastructures to a database service provider. In the DAS model, a service provider employs data encryption techniques to ensure the privacy of hosted data. The security of encryption techniques relies on the confidentiality of the encryption keys. The dynamic nature of the encrypted database in the DAS model adds complexity and rises specific requirements on the key management techniques. Key updates are particularly critical because of their potential impact on overall system performance and resources usage. In this paper, we propose specialized techniques and data structures to efficiently implement the key updates along with the other key management functions to improve the systems' concurrency performance in the DAS model.

1. INTRODUCTION

The commodity pricing of processors, storage, network bandwidth, and basic software is continuously reducing the relative contribution of these elements to the total lifecycle cost of computing solutions. Operating and integration costs are increasing, in comparison. The research community has responded by working on approaches to automated system administration as in [2]. Increasingly, large companies are consolidating data operations into extremely efficiently administered data centers, sometimes even outsourcing them [4].

The *Database-as-a-Service* (DAS) model [8] is one manifestation of this trend. In the DAS model, the client's database is stored at the service provider. The provider is responsible for provisioning adequate CPU, storage, and networking resources required to run database operations, in addition to the system administration tasks such as backup, recovery, reorganization etc.

A fundamental challenge posed by the DAS model is that of database privacy and security [8]. In the DAS model, the user data resides on the premises of the database service provider. Most companies and individuals view their data as an asset. The theft of intellectual property already costs organizations great amount of money every year [3]. The increasing importance of security in databases is discussed in [6, 12, 11–1, 8, 7, 5, 9, 10]. Therefore, first, the owner of the data needs to be assured that the data is protected against malicious attacks from the outside of the service provider. In addition to this, recent studies indicate that 40% of those attacks are perpetrated by the insiders [3]. Hence, the second and more challenging problem is the privacy of the data when even the service provider itself is not trusted by the owner of the data. Data encryption is proposed as a solution to ensure the privacy of the users' data. The first problem is examined in [8] and the second one is studied in [7], which explores how SQL queries can be executed over encrypted data.

The security of any encryption technique relies on the confidentiality of the encryption keys. Hence, key management plays an essential role in a system, which employs encryption techniques. In this paper, we mainly focus on the key management issues in the context of the database-as-a-service model, where the clients' databases are stored at the service provider site in the encrypted form. We argue that the key management in the hosted databases requires special consideration especially due to the dynamic nature of the database systems.

The update transactions are an essential part of the database systems and applications. Each update transaction requires at least one invocation of the encryption function to encrypt the data in the system.[1] It is known that encryption is a CPU intensive process [8]. Therefore the update transactions may hold locks on the certain set of database records for an extended period of time causing a decline in the system performance. Besides the database update transactions, re-keying is another process, which requires the invocation of the encryption function in the system. As we discuss in Section 3 re-keying is recommended and sometimes required for the systems that employ encryption. Re-keying large amounts of data entails significant encryption costs and interferes with the other transactions thereby causing a performance degradation in the system. In this study, we address these issues by proposing a specialized key management architecture in Section 3. We also introduce a system architecture taxonomy in Section 2.3, which is coupled with the key management architecture to enable the performance-conscious encryption key management in dynamic database environments.

[1]The actual number of invocations depends on various factors such as the data unit subject to the encryption, i.e., the granularity of the encryption, specifics of the transaction, e.g., an insert only transaction, a transaction on a number of data objects, etc.

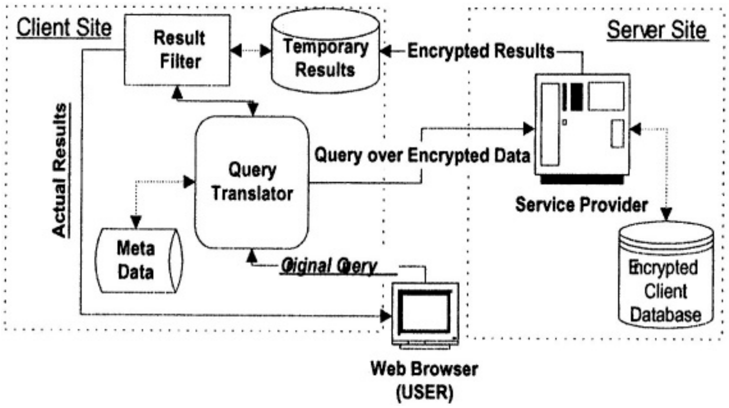

Figure 1. Database-as-a-Service architecture

Table 1. Relation *emp*

eid	ename	salary	addr	did
23	Tom	70K	Maple	40
860	Mary	60K	Main	80
320	John	23K	River	35
200	Sarah	55K	River	10

2. SYSTEM ARCHITECTURES

2.1 Overall DAS Architecture

The system we use in this study is based on the architecture proposed and described in [7]. The basic architecture and the control flow of the system are shown in Figure 1. It is comprised of three fundamental entities. A *user* poses the query to the client. A *server* is hosted by the service provider that stores the encrypted database. The encrypted database is augmented with additional information (which we call the index) that allows the certain amount of query processing to occur at the server without jeopardizing the data privacy. A *client* stores the data at the server. Client[2] also maintains the *metadata* for translating the user queries to the appropriate representation on the server, and performs post-processing on server query results. From the privacy perspective, the most important feature is, the client's data is always stored in the encrypted form at the server site. The server never sees the unencrypted form of the data, and executes the queries directly over the encrypted data without decrypting it.

[2]Often the client and the user might be the same entity.

Table 2. Encrypted representation emp^S of emp

RID	KID	*etuple*	eid^{id}	$ename^{id}$	$salary^{id}$
1	45	=*?Ew@R*((¡¡=+,-...	2	19	81
2	78	b*((¡¡(*?Ew@=l,r...	4	31	59
3	65	w@=W*((¡¡(*?E:,j...	7	59	22
4	52	ffTi* @=U(¡?G+,a...	8	49	59

2.2 Storing Encrypted Data in the Database

We briefly summarize how the client's data stored at the server in an encrypted fashion in the DAS model.[3]

For each relation $R(A_1, A_2, \ldots, A_n)$, we store, on the server, an encrypted relation: $R^S(RID, KID,\ etuple, P_1^{id}, P_2^{id}, \ldots, P_i^{id})$, where $1 \le i \le n$. Here, an *etuple* stores an encrypted string that corresponds to a tuple in a relation *R*. Each attribute P_i^{id} stores the partition index for the corresponding attribute A_i that will be used for query processing at the server.

For example, consider the relation *emp* given in Table 1 that stores information about employees. The *emp* table is mapped to a corresponding table, shown in Table 2, at the server: $emp^S(RID, KID, etuple,\ eid^{id}, ename^{id}, salary^{id})$.

The RID represents record identifier, which is a unique number created by the client for each tuple. Here, the RIDs are not the same as unique identifiers, which are used as references to the records and assigned by the database manager, as it is done in most of the commercial database products. Instead, these RIDs also uniquely identify the records, however, they are created and assigned by the client to facilitate the schemes we present in the study. The KID represents the key identifier, which is also created and assigned by the client. The KID indicates which key is used to encrypt the *etuple* of the corresponding tuple. We elaborate the use of KIDs in Section 3.3. The column *etuple* contains the string corresponding to the encrypted tuples in *emp*. For instance, the first tuple is encrypted to "=*?Ew@R*((¡¡=+,-... " that is equal to $\mathcal{E}_k(1, 23, Tom, 70K, Maple, 40)$, where $\mathcal{E}$ is a deterministic encryption algorithm with key k. Any deterministic encryption technique such as AES, DES etc., can be used to encrypt the tuples. The column eid^{id} corresponds to the index on the employee ids.[4]

[3]We will not repeat all of the details of the storage model here, since it is thoroughly discussed in [7]. Rather, we only provide the necessary notations to explain the constructs we develop in this work.
[4]The details of creation of those index values can be found in [7].

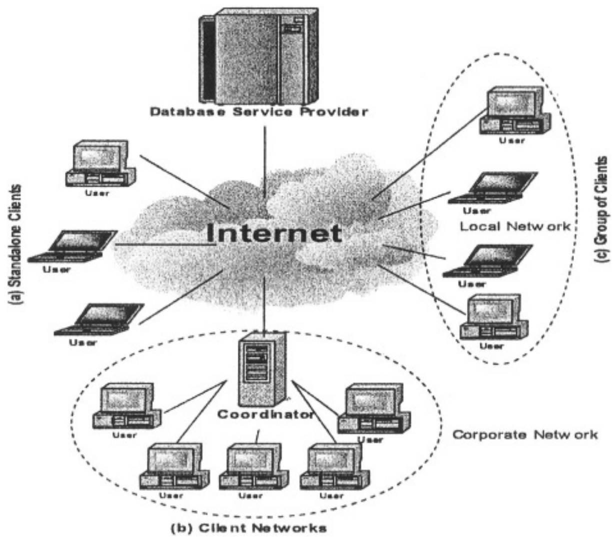

Figure 2. Architectural model alternatives for database service

2.3 Classification of the System Architectures

In this section, we propose different instantiations for the overall system architecture presented above. Our classification of the system architecture alternatives is *client-oriented.* In other words, we identify the architecture model based on how the clients interact with the service provider. We classify the system architecture models under three categories; *standalone clients, group of clients,* and *client networks.* Each model has implications on the characteristics of the system including the control flow, index management, key management, and query processing. We first present the details of each architecture below.

Standalone clients: In the standalone clients model, shown in Figure 2(a), each client is a single node connecting to the service provider individually. The client does not directly share the data with the other clients. Possible example for the clients of this architecture is personal users accessing to the services, such as e-mail, rent-a-spreadsheet etc., via a web browser or a lightweight application interfaces.

Client networks: In this architecture, shown in Figure 2(b), the client of the service is a network rather than the individual nodes. A characteristic example for this architecture is larger corporations, which maintain their own network infrastructure as corporate networks and outsource some or all of their IT operations. In this model, the nodes inside the network utilize a connection point (or multiple points) to communicate with the service provider. We call this distinguished node as *coordinator node.* The coordinator node is responsible for set of operational tasks, such as maintaining metadata information required to execute queries directly over encrypted data (as described in Section 2.1), executing transactional semantics in the multi-tier client/server architecture, and the key management tasks as we describe in Section 3.

Group of clients: In this case, as shown in Figure 2(c), multiple clients access to the same service individually. Those clients are somehow related to each other. The relationship can be organizational, i.e., the group of clients belonging to an organization, or data sharing or both. A typical example for this model is small companies, which have multiple but limited number of users. They do not want to (or need to) maintain an integrated network infrastructure containing the coordinator nodes as in client networks case. Nonetheless, they need to enable collaboration among the user nodes in the organization as the users (or employees) of them would be sharing the data in terms of querying and updating and are related by business means. Therefore the user nodes are connected to each other to share local information, such as the metadata. Inherently this information is managed in a distributed fashion. We will not further discuss the distributed data management techniques in this context since it would cause us to diverge from the main content of the paper.

3. KEY MANAGEMENT

Key management is a group of policies and procedures that regulate the maintenance of encryption keys within the system. Key management techniques have been extensively studied in the applied cryptography literature [13]. In this study, we will discuss the most relevant aspects of the key management techniques to database-as-a-service model by considering their implications on the system implementation issues. We consider the following components of the key management architecture: *key generation, key installation, key distribution,* and *key update.*

We will discuss each of these functionalities in the context of the DAS model and indicate where the each of the tasks are identified in the respective sections. However, before that we will discuss another important notion, key assignment granularity, which affects the discussion of the techniques and the constructs.

3.1 Key Assignment Granularity

A key can be used to encrypt different database objects in the database, such as a table, or a row. We call this as the assignment granularity of the key. The selection of granularity would have its own pros and cons, depending on the system setup, limitations on computing and storage of the client etc., and the security requirements. We classify the key assignment granularity into three categories; database-level, table-level, and vertical-partitions-level.

• *Database-level* granularity indicates that only one key is used for the whole database. Any data unit, which is processed in the database, is encrypted with the same key, which is created for the whole database.

• *Table-level* granularity indicates that there is one key that is used for a given group of tables. Note that, in a table group, there might be only one table. If the group consists of multiple tables, then each table will have an individual entry in the key registry (discussed in Section 3.3) differing only in the key correspondence values. For example, key k_1 can be created for table *emp* and key k_2 can be used for the tables *mgr* and *proj.*

• In *vertical-partitions-level* case, a group of database rows are encrypted with the same key. In the most extreme case, a different key is used for each row. Alternatively, the rows can be grouped. A typical example would be using the domain value intervals that are used to create the partition ids in the encrypted version of the table (if the equi-width histograms are used for partitioning). All rows in a value interval can be encrypted with the same key. For example, the key k_1 can be used to encrypt the rows of *emp* table, whose *mgr.salary* values fall in $[30K, 50K)$ and the key k_2 can be used for the rows, whose *mgr.salary* values fall in $[50K, 70K)$.

Note that, the key assignment granularity is different from the granularity of data that is subject to the encryption. For example, we may choose table level-key assignment granularity for the *emp* table. Thus, one key is used to encrypt any data, which would be inserted into the table. However, since we use row-level encryption as data granularity in the model, each tuple is encrypted individually with the key assigned for *emp* table to create *etuples.*

3.2 Key Generation

Key generation involves the creation of the encryption keys that meet the specifications of the underlying encryption techniques. These specifications define the properties, such as size, randomness, that the keys should have. The medium in which keys are created is a particular interest for the DAS model since the decision has both security and performance implications. We propose the classification of key generation schemes in two categories; the *pre-computation* based scheme and the *re-computation* based scheme.

Pre-computation: The encryption keys are computed and stored (the storage is discussed in Section 3.3) in a ready-to-use format at any time. In this approach, the keys are stored in a directly usable form. Hence we do not need to re-compute the keys when they are used. This saves the computation required for the key generation, which is an advantage for this approach. However, if the number of keys in the system increases, the size of the key registry (discussed in Section 3.3) increases as well. This will lead to an increased storage

KID List	Correspondence	Mode	Material
45,62,109	table:mgr	pre-compute	*key_for_emp_table*
92,4	proj.RID:[1,20]	pre-compute	*key_for_1st_RIDs*
112,52,97	proj.RID:[21,50]	pre-compute	*key_for_2nd_RIDs*
77,32	emp.eid.PID:1,5,6	re-compute	*seed_for_1st_PIDs*
23,13,11	emp.eid.PID:3,7,9	re-compute	*seed_for_2nd_PIDs*

Figure 3. Key Registry

requirement, which is a disadvantage. The number of required keys is related to the key assignment granularity selection, which is discussed in Section 3.1.

Re-computation: In this case, the keys are re-computed whenever they are needed for encryption/decryption. The required information to re-compute the keys is obtained from the data items we discuss in Section 3.3. Specifically, the *key material* column of the key registry provides the seed for the key generation algorithm, such as MD5, SHA, etc., used in the system. The key generation algorithm is executed with the key material information to re-compute the needed keys. If the number of keys is small, the approach reduces the size of the key registry. On the other hand, if the number is larger, then the overhead due to re-computation can be significant.

In the DAS model there are two places where the key generation may take place. The first option is the client itself and the second option is a third party trusted server, which provides the key generation (and possibly additional key management functions) as a service. Note that, we do not consider the server as an option since the server is considered as an untrusted party in the model.

Generating the keys at the client site provides flexibility, less complexity in terms of system management, and eliminates the requirement for trust mechanisms, which regulate the collaboration between the client and the third party key server. The flip side is that the key generation process becomes the client's responsibility, which may incur computational overhead on the client's system resources that may be limited.

3.3 Key Installation and Key Registry

Once the keys are generated, they need to be operational and accessible for the authorized users. The key installation defines how and where the key are stored during the regular use. We propose a specialized data structure, *key registry,* that is responsible for storing the key material information.

The *key registry* is the data structure that stores all the required information to manage the keys. It has a tabular structure, shown in Figure 3, which consists of four columns corresponding to *Key ID (KID) List, Key Correspondence, Key Mode, Key Material,* and an indefinite number of rows, each corresponding to a different key that is used in the system. We will discuss where and how the key registry is stored in Section 3.4.

• *Key ID (KID) List* provides a list of numbers that are used to identify the corresponding key. Note that a key does not need to have a unique identifier. These numbers are just used to make the associations between the records read from the encrypted database tables and the key registry entries. When an encrypted tuples is read from the database (or a tuple is to be inserted into the database) the system should know which key had been used to encrypt the record. KID column in the encrypted storage model (Section 2.2) provides that information. Maintaining multiple identification numbers for the keys also increases the security afforded by the system. An adversary cannot directly recognize the *etuples,* which are encrypted with the same key.

• *Key Correspondence* indicates the database object to which the key is assigned. The database object is one of the granularity choices, as defined in Section 3.1, in the system. In the correspondence column of the key registry, we use a special notation to indicate the correspondence to the database objects. (Note that, here we describe the conceptual implementation of the key registry. An actual implementation of this framework could be done in different ways to achieve a better performance.) The set of correspondence identifiers are: *database, table, RID, PID,* where *database* indicates whole database, *table* indicates single or multiple tables, *RID* indicates a set of record identifiers, and *PID* indicates a set of partition ids. *RID* and *PID* identifiers are qualified by the necessary qualifiers, such as a table name and a column name, that they belong to. The interval of values is represented in the brackets, e.g., [20,50] indicating the continuous interval of values between 20 and 50. The list of values are given separated by comma, e.g., 20, 22, 45.

• *Key Mode* specifies whether the key generation method is pre-computation or re-computation, as they are described in Section 3.2.

• *Key Material* contains either the actual key, if the key mode is pre-computation, or the necessary initialization parameter(s) required for the re-computation of the key, if the key mode is re-computation.

For example, consider the key registry entries given in Figure 3. The first row indicates that only one key is assigned to *mgr* table (see the second column), thus any data item inserted into the table is encrypted with that key. The third column, Mode, shows that pre-computation mode is used, therefore the key is already generated and stored in the Material column. The second row shows that any record of *proj* table whose RID is between 1 and 20 is encrypted by using the key given in the Material column. The forth row examplifies the use of PIDs. Any record of *emp* table whose partition id for *eid* attribute is in the list of {1,5,6} is encrypted by using the key given in the Material column. Note that, for this entry, the Mode column indicates re-compute. Thus, each time the key is re-generated by using the seed value given in the Material column.

3.4 Key Distribution

After a key is generated, a corresponding entry is created in the key registry. Upon request, the keys should be provided to the authorized users. This process is called *key distribution.* Similar to the case for the key generation function there are different alternatives where the key distribution can be handled, the client site, a trusted third party service provider, and the server site.

For the standalone clients model, the client either stores the key registry on its machine or utilizes a trusted third party server for this purpose. Yet another possibility is to store the key registry at the server site unlike key generation function. The key registry can be encrypted by using a *master key* and stored at the server securely. When the client needs to use key material, it can be downloaded from the server and be decrypted with the master key. These alternatives are also valid for the client networks and the group of clients models. For the former, coordinator node can act as a medium for storage and communication with the other users.

If the server or a third party server is chosen for the key distribution, the user authentication is an issue to address. This can be solved by using public key infrastructure (PKI). After the key generation, the key registry can be locked with the public key. This way anyone can request the encrypted key registry but only the authorized users can decrypt using their private key.

3.5 Key Updates

From the security perspective the lifetime of an encryption key should be limited and the key should be removed from the active usage under certain circumstances. Re-keying is recommended and sometimes required. Periodic re-keying is considered as a good practice, especially, for data stored over an extended period of time to prevent a potential key compromise. If a key compromise is suspected or expected, an emergency re-keying has to be performed. In those cases, all affected keys should be changed.

The key update has significant implications on the DAS model in which large amount of data is divided into the parts and encrypted with different keys. Therefore we particularly emphasize the need for the efficient mechanisms to handle the key updates. Above, we presented how the encryption keys can be applied at different granularity levels. Choosing a finer level granularity would increase the security, at the increased cost of key management since larger number of data items would be encrypted with different keys.

From a database system point of view, the interference between the key update procedure and regular database queries, being executed by the users, should be minimized. This relates to the concurrency performance of the system. Generally, the key update procedure consists of five main steps:

1) Generation and installation of a new key

2) Fetching the *etuples* that are subject to key change
3) Decryption of the *etuples*
4) Re-encryption of the *etuples* with the new key
5) Replacement of the *etuples,* re-encrypted with the new key

In this procedure, the records, which are subject to the key change are re-encrypted with the new keys. Therefore, duration of this process, the records should be locked for any update transaction. Otherwise an update transaction may update a record with a new content while the re-encryption process is in progress. When the re-encryption is completed the old content, which is encrypted with the new key is inserted back into the database. This would overwrite the updated value of the record causing inconsistency in the database. Note that, usually, the client has limited computational and storage power and encryption and decryption are particularly computationally very expensive operations [8]. Therefore, this may lead to a longer duration of key update procedures. If the key update blocks out significant amount of user transactions then throughput of the system may considerably deteriorate.

It may appear that by choosing a finer granularity for processing, for example a row-at-a-time, the key update procedure could be speeded up with lesser interference with the other transactions. This would, however, cause an increased network traffic and message initiation cost since the number of transmission requests increase with the finer granularity. Yet, there is another deeper problem that we have to consider from the security point of view. If the client re-encrypts and inserts back a single *etuple,* then the server would know that the new and the old *etuples* correspond to the exact same data but just encrypted with different keys. This nullifies the key change. Alternatively the client can perform the key updates over group of *etuples,* but this solution takes us back to the concurrency problem described above.

Secondly, we need to be judicious about the system resource usage due to key updates. This includes, network bandwidth and I/O overhead.

To address these issues, we devised techniques to handle the key updates in a performance-conscious manner. In these techniques, our goal is to minimize the interference between the key update procedures and the other user transactions, and to minimize the system resources usage. We describe our techniques in the client networks architectural class where the requirements we stated above are most pronounced. The applications of the techniques extend to the other classes, namely; standalone clients and the group of clients, if it is needed.

3.5.1 Key update protocol. The overall key update protocol is shown in Figure 4. We bring the *etuples* (along with RIDs and KIDs) that are subject to key change to the coordinator node in groups in a certain size, α. α is a system parameter, which is determined by considering the performance and

```
Input: $\alpha$: number of etuples in a group

1 Bring $\alpha$ etuples to the coordinator
2 Decrypt etuples
3 Encrypt etuples with the new key
4 Generate $\alpha$ new RIDs
5 Shuffle etuples with the RIDs
6 Insert etuples back to the database
```

Figure 4. The key update protocol

security requirements. The number of *etuples* that can be brought to the coordinator node is limited by the processing and storage resources in the node. We elaborate the security aspect of it below.

The coordinator node first decrypts the *etuples* and re-encrypts them with a new key. (A new entry for the key in the key registry is created before the initiation of this process.) Afterward, the coordinator node shuffles the tuples by replacing the RIDs with the new randomly generated ones. Finally, those re-encrypted and shuffled tuples are inserted back into the database at the server by replacing the old ones. (For the sake of simplicity, here we assume that there are no other pointers that are maintained separately at the server to the records, such as indices over partition ids.) Note that, as we shuffle the RIDs, the server cannot know one-to-one correspondence between the old *etuples* and newly encrypted ones. The server can only try to match them. This point is the motivation behind using the group of data units in certain size. As the size of the group increases it becomes unfeasibly expensive to try all the possibilities for the server.

3.5.2 Concurrent updates. Having discussed the security side of key update scheme, now we turn our attention to the system performance side of it. We consider user transactions in two groups, **read** transactions and **update** transactions. The read requests do not change the data in any means. The update requests, on the other hand, may insert/delete data or change some part or all of the existing data stored at the server. (Note that, the implementation of transactional semantics in a client/server environment is an orthogonal issue to our discussion here.)

Read transactions can be executed concurrently with the key update procedure. When a group of *etuples* are brought in to the coordinator node, the original copies of those are still available at the server for querying. Note that, from the query processing (over encrypted data) perspective the only critical attributes in the storage model are partition indexes. All supported query conditions are handled by making use of the partition indexes [7].

```
Input: TTL: time-to-live

1  read the etuple
2  decrypt the etuple
3  update the data
4  check the update list of the coordinator by RIDs
5  if the updated tuple is not in the list then
6      read key information from the key registry
7      encrypt the tuple
8      insert the etuple into the database
9  else
10     if processing time < TTL then
11       send the tuple to the coordinator
12       encrypt the tuple at the coordinator
13       include it in the coordinator update list
14     else
15       read key information from the key registry
16       encrypt the tuple
17       drop the tuple from the coordinator update list
18       insert the etuple into the database
```

Figure 5. Handling the update transactions with the key update

Based on the encrypted data storage model, predicates in the user query are evaluated as described in [7]. This process includes the translation of the query into a form that retrieves the (super)set of *etuples* by evaluating the predicates directly over encrypted data. When the qualified *etuples* (along with RIDs and KIDs) are fetched, the client looks up the key registry and finds out the valid key(s) for each *etuple* and decrypts them. Note that, even the coordinator node runs a key update over the *etuples* that are returned as the answer of the query, the content of the data is the same. The only information the user needs to correctly decrypt is the valid keys and this information is provided by the key registry.

Update transactions need a special attention as, unlike the read transactions, they change the content of the data. Algorithmic steps to efficiently handle the update transactions is given in Figure 5. Upon request, the user receives and decrypts the *etuples.* After this, the user performs the changes over the data. Next, a tuple, which has been updated, has to be encrypted to produce corresponding *etuple.* (Here, we assume that the data update procedure is performed a tuple-at-a-time.)

The user checks if the tuple is in the list of tuples being processed by the coordinator node (Line 4). This can be done by using the RIDs. To make the look-up even more efficient, we can store the tuples in a sorted list or in a tree based data structure based on their RIDs at the coordinator node. (As we stated earlier, RIDs are assigned by the client and they are not used as references to records by the server.) If the tuple is not in the coordinator node's list then the user retrieves the encryption key information from the key registry, encrypts the tuple and inserts it into the database.

If the tuple is in the coordinator node's list then the user has two options; first (Line 10), the user can transfer the updated tuple to the coordinator node for encryption. The coordinator node first replaces the copy of the tuple with the updated tuple, encrypts it with the new key, and inserts it into the database along with the other tuples by following the procedure described earlier. Here the RID of the tuple is not shuffled since it will replace the old version, containing the old content, in the database. As a second option (Line 14), the user can encrypt the updated tuple with the new key by itself. Note that, the new key information is placed in the key registry before the coordinator node starts its processing. Then the user sends a notification to the coordinator node and the coordinator node drops the corresponding tuple from its list. Following this, (Line 18), the client inserts the updated tuple into the database. Since the coordinator node drops the tuple from the update list, it is not included in the tuples that are re-encrypted and inserted back by the coordinator node thereby preventing the overwriting and inconsistency.

The decision between those two alternatives should be made dynamically by considering the performance requirements of the system and the current status of the processes. Another system parameter we maintain for the coordinator node is *time-to-live* (TTL). The TTL defines the maximum time frame for a completely processed group of re-encrypted tuples before they are inserted back into the database. If the TTL will have been passed when the update have been finalized then the user chooses the second alternative (Line 14).

This procedure allows us to defer the updates to the database and piggyback the re-encrypted tuples with the ones in coordinator update list to be inserted into the database thereby increasing the system's concurrency performance and security. For some situations, deferring the updates is preferred. The coordinator node can wait, instead of inserting back the re-encrypted tuples, for an updated tuple, which is included in the coordinator update list. This can be done as long as the TTL is valid. By doing this the coordinator node can piggyback the tuples with the tuples that are already in its list and inserts them together into the database. This both improves the systems resources usage and the security afforded by the system.

4. CONCLUSIONS

We have studied the efficient encryption key management problem in the encrypted database environments, specifically in database-as-a-service (DAS) setups. We have presented specialized techniques and data structures to improve the efficiency of the key management functions, which also deliver higher degree of concurrency in the system. We particularly observed the importance of the key update procedures and proposed efficient key update alternatives, which allow the system to update the keys in a concurrent fashion.

The system specific parameters, namely; the group size α and TTL parameters, have implications on the security and the performance afforded by the system. Hence, the quantification of their impact is an important issue for the future work. Moreover, there are system design issues that require further research. For example, fault-tolerance issues for the key update procedures should be studied in detail.

References

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *Proc. of VLDB,* 2002.

[2] S. Chaudhuri, E. Christensen, G. Graefe, V. R. Narasayya, and M. J. Zwilling. Self-tuning technology in microsoft sql server. *Data Engineering Bulletin,* 22(2):20–26, 1999.

[3] Computer Security Institute. CSI/FBI Computer Crime and Security Survey. *http://www.gocsi.com,* 2002.

[4] ComputerWorld. J.P. Morgan signs outsourcing deal with IBM. Dec. 30, 2002.

[5] E. Damiani, S. D. C. di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted Relational DBMSs. In *Proc. of 10th ACM Conf. On Computer and Communications Security,* 2003.

[6] B. Fernandez, R. C. Summers, and C. Wood. *Database Security and Integrity.* Addison-Wesley, 1981.

[7] **H. Hacıgümüş,** B. Iyer, C. Li, and S. Mehrotra. Executing SQL over Encrypted Data in Database Service Provider Model. In *Proc. of ACM SIGMOD,* 2002.

[8] **H. Hacıgümüş,** B. Iyer, and S. Mehrotra. Providing Database as a Service. In *Proc. of ICDE,* 2002.

[9] **H. Hacıgümüş,** B. Iyer, and S. Mehrotra. Ensuring the Integrity of Encrypted Databases in Database as a Service Model. In *Proc. of 17th IFIP WG 11.3 Conference on Data and Applications Security,* 2003.

[10] **H. Hacıgümüş,** B. Iyer, and S. Mehrotra. Efficient Execution of Aggregation Queries over Encrypted Relational Databases. In *Proc. of International Conference on Database Systems for Advanced Applications (DASFAA),* 2004.

[11] J. He and M. Wang. Cryptography and relational database management systems. In *Proc. of IDEAS '01,* 2001.

[12] T. Lunt and E. B. Fernandez. Database Security. *ACM SIGMOD Record,* 19(4), 1990.

[13] D. R. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.