Leveraging NoSQL for Scalable and Dynamic Data Encryption in Multi-Tenant SaaS

Ansar Rafique, Dimitri Van Landuyt, Vincent Reniers, Wouter Joosen
imec-DistriNet, KU Leuven
3001 Leuven, Belgium
E-mail: firstname.lastname@cs.kuleuven.be

***Abstract*—In the context of multi-tenant SaaS applications, data confidentiality support is increasingly being offered from within the application layer instead of the database layer or the storage layer to accommodate continuously changing requirements of multiple tenants. Application-level data management middleware platforms are becoming increasingly compelling for dealing with the complexity of a multi-cloud or a federated cloud storage architecture as well as multi-tenant SaaS applications.**

However, these platforms typically support traditional data mapping strategies that are created under the assumption of a fixed and rigorous database schema. Thus, mapping data objects while supporting varying data confidentiality requirements, therefore, leads to fragmentation of data over distributed storage nodes. This introduces significant performance overhead at the level of individual database transactions (e.g., CRUD transactions) and negatively affects the overall scalability.

To address these challenges, we present a dedicated data mapping strategy that leverages the data schema flexibility of columnar NoSQL databases to accomplish dynamic and fine-grained data encryption in a more efficient and scalable manner. We validate these solutions in the context of an industrial multi-tenant SaaS application and conduct a comprehensive performance evaluation. The results confirm that the proposed data mapping strategy indeed yields scalability and performance improvements.

***Keywords*-Cloud data storage, NoSQL, Data encryption, Untrusted clouds, Secure data management, Multi-tenant SaaS**

I. INTRODUCTION

The limitations of traditional relational databases in a cloud environment has lead to the development of a set of cloud-friendly databases commonly known as NoSQL databases [7], [26]. NoSQL databases have become the backbone of cloud-based applications as they ease handling of large volumes of data, while also ensuring optimal performance, high availability, and elastic scalability [4], [19].

Cloud data storage –as a key enabler of cloud computing– offers numerous benefits in terms of infinite capacity on demand, no upfront cost, elastic scalability, etc [9]. For these reasons, many organizations actively outsource their storage to external third-party cloud storage providers [27]. However, organizations are also often reluctant to store their sensitive data to external third-party cloud providers, due to limited trust and privacy issues [9], [12], [17]. This forms a main obstacle to wider adoption of cloud storage and the respective cloud storage providers [18], [25]. In addition, relying only on database-level access control does not suffice [15] as it still relies on trust in the storage provider who can bypass such mechanisms. Therefore, cryptographically encrypting confidential data from within the *application layer* is the most feasible and widely adopted cloud storage data security tactic [20].

In the context of a multi-tenant Software-as-a-Service (SaaS) application, complex confidentiality and privacy requirements, which differ considerably between tenants must be supported in an efficient and scalable manner. For example, different customer organizations (tenants) may impose different confidentiality requirements at any level of granularity (i.e. from entity towards individual attributes). However, addressing these requirements in the *storage layer* is impossible as it does not support encryption at the most fine-grained level [9]. Similarly, encryption at the *database layer* operates on a fixed schema, makes it extremely difficult to handle continuously changing encryption requirements of individual tenants at run-time, which seek flexibility in the schema and therefore is undesirable. Encryption at increasing levels of granularity, which can also be altered at run-time is feasible in the *application layer*, but is not transparent to the application [9]. This requires significant modification in the application and hence introduces substantial application complexity.

Moreover, the existing traditional data mapping strategies hinder performance significantly by the use of encryption and thus do not allow efficient database transactions. For example, applying encryption for a particular tenant at a more fine-grained level (i.e. individual attributes) leads to with encrypted data being organized in multiple database rows, which is a data mapping strategy that suffers from *scalability* and *performance* problems. Furthermore, the overall *performance* of CRUD database transactions is affected when encrypted attributes of an entity end up being stored as multiple rows in an encrypted table. Consequently, in a distributed setup —in which rows of a table are distributed across multiple storage nodes based on the partition key— data logically belonging together (e.g. encrypted attributes of the same entity) easily become scattered on different nodes, which limits *scalability* in general and further introduces a *performance* disadvantage when having to reassemble the entity.

This paper presents an alternative data mapping strategy that addresses these problems by leveraging the data schema flexibility of columnar NoSQL databases such as Apache Cassandra [2], thereby supporting dynamic data encryption at different levels of granularity and avoiding data fragmentation in a distributed storage setup. We envision our data mapping

2324-9013/17 $31.00 © 2017 IEEE
DOI 10.1109/Trustcom/BigDataSE/ICESS.2017.327

strategy to be supported within the *middleware layer* in order to (i) make it transparent and reusable solution to be used within different applications, (ii) address different confidentiality requirements of individual tenants at run-time by offering data encryption support at various levels of granularity, and (iii) provide a support to secure both data-in-motion (while data moving between the application and the database) and data-at-rest (data is stored in the permanent storage such as database), while also ensuring good performance and better scalability. Our prototype implements and integrates this strategy in the context of an existing data access middleware platform, Impetus Kundera [1], and functionally validates this in the context of a realistic multi-tenant SaaS application. By systematically benchmarking, we clearly show the performance and scalability benefits of the proposed solution.

The remainder of this paper is organized as follows: Section II introduces a motivating case and describes the state-of-practice to realize the confidentiality requirements of the motivating case. Section III describes our solution, which leverages columnar NoSQL databases to accomplish scalable, dynamic, and fine-grained data encryption. Section IV presents our evaluation, while Section V connects and contrasts our work with other related research. Finally, Section 7 concludes the work and indicates future research directions.

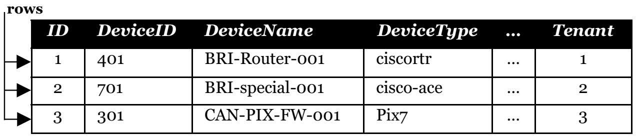
II. MOTIVATION

This paper is motivated by requirements that we encountered in a number of industrial SaaS application cases [3], [6]. For illustration purposes, we focus on one such application case which we introduce in Section II-A. However, we do intend our solution to be applicable to a wide range of applications. Then, Section II-B outlines tenant-specific confidentiality requirements of the motivating case. Finally, Section II-C illustrates the application of data encryption in the current state-of-practice, and Section II-D discusses the limitations of this approach, as such motivating this work.

A. Multi-Tenant Log Management Case

The running example in this paper is a multi-tenant Log Management-as-a-Service (LMaaS) application. This SaaS application provides log management facilities to its customer organizations (i.e. tenants). The tenants of this SaaS application are organizations from different application domains, for instance, banks, hospitals, supermarkets, etc. The application focuses on storing large amounts of heterogeneous data: *raw log entries*, *archived logs*, *log meta-data*, *historical logs*, *incident reports*, and *time series data* and is successful in doing so by combining external cloud storage resources and different database technologies in a so-called *federated storage architecture*, and by applying multi-tenancy, i.e. sharing these storage resources and technologies maximally among tenants.

In the case of the multi-tenant log management application, log aggregation components are installed at the tenants' side, which generate streams of log events. Figure 1 illustrates three such log events, each sent by a different tenant organization, which are stored in a single `Log` table.



ID	DeviceID	DeviceName	DeviceType	...	Tenant
1	401	BRI-Router-001	ciscortr	...	1
2	701	BRI-special-001	cisco-ace	...	2
3	301	CAN-PIX-FW-001	Pix7	...	3

Fig. 1: `Log` table for storing events information.

The table holds a chunk of log data, identified by an *ID* attribute, which uniquely identifies each row in the `Log` table. The (*DeviceID*, *DeviceName*, *DeviceType*, and ...) attributes hold information about the devices that generate the log events. The *Tenant* attribute refers to the tenant for which the log event is generated.

B. Tenant-specific Data Confidentiality Requirements

In the context of a multi-tenant SaaS application, not all tenants share the same confidentiality requirements and different data confidentiality requirements may affect different levels of granularity. As an example from the multi-tenant log management application, we contrast three tenant organizations, a financial agency (i.e. a bank), a medical institution (i.e. a hospital), and a small and medium-sized enterprise (SME) representing a supermarket, that each imposes contrasting confidentiality requirements: stricter regulations on data confidentiality apply for the financial and medical institutions, as opposed to the SME (see Figure 2).

To illustrate this, as the tenant with id 1 is a financial agency, even the meta-data about the device is considered highly sensitive. Similarly, stricter confidential requirements apply to the tenant with id 3, which represents a medical institution (DeviceID, DeviceName, and DeviceType attributes) as compared to the tenant with id 2, which represents an SME (DeviceID and DeviceName attributes) as shown in Figure 2.

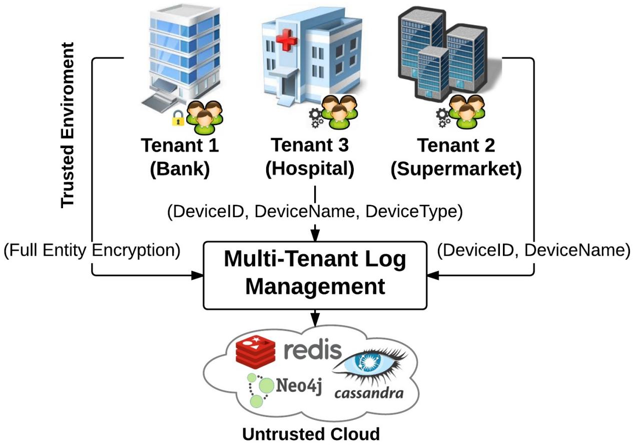


Fig. 2: Tenants of the log management application enforce confidentiality requirements at differing levels of granularity.

A multi-tenant SaaS application such as the log management application has to deal with heterogeneous data of varying degrees of confidentiality from different tenants. Therefore, such applications must have the technical ability to vary their data storage policies accordingly and dynamically (for example, when tenants update their service-level-agreement (SLA) stipulations).

C. State-of-Practice Data Mapping

The current research is mainly focused on solving the encryption problem at the *database layer*. Therefore, existing solutions in both the current state-of-art [10], [28] and the state-of-practice [11], [14] are based on a fixed database schema and provide limited support for the data encryption[1]. However, addressing the contrasting and continuously changing confidentiality requirements of each tenant requires flexibility in the data model to support considerably different levels of encryption (e.g., encrypt full entity, specific attributes).

The state-of-practice solution to perform encryption at different levels of granularity is to separate the sensitive data from the non-sensitive data in different database tables. The data is commonly separated to support encryption at differing levels of granularity, which cannot be expressed in the fixed database schema model. This involves grouping all sensitive attributes of a row, which need to be encrypted and storing them together. This results in a reduced number of encryption operations, however, it comes with a considerable amount of computational overhead. For example, performing the decryption operation on one out of several encrypted attributes, computational overhead would be incurred due to decryption of all other encrypted attributes. Therefore, to perform encryption at differing levels of granularity without incurring too much computational overhead, each attribute of an entity, which needs to be encrypted must be stored separately.

In the context of database mapping, a new table is commonly created to separate the sensitive data from the non-sensitive data. As shown in Figure 3, `Encrypted_Log` table, which holds ciphertext is created to store the sensitive data of the `Log` table, which contains plaintext (see Figure 1).

partition key

ID	*Table*	*EncName*	*Value*	*TID*	*Tenant*
Log|1|1	Log		Byte[]	1	1
Log|2|2|DeviceID	Log	DeviceID	Byte[]	2	2
Log|2|2|DeviceName	Log	DeviceName	Byte[]	2	2
Log|3|3|DeviceID	Log	DeviceID	Byte[]	3	3
Log|3|3|DeviceName	Log	DeviceName	Byte[]	3	3
Log|3|3|DeviceType	Log	DeviceType	Byte[]	3	3

partitions/rows

Fig. 3: `Encrypted_Log` table for storing encrypted events information (ciphertext).

The *ID* attribute is a single partition key, which uniquely identifies each row within the `Encrypted_Log` table. The partition key is made up by combining several attributes (*Table*, *ID*, *Tenant*, and *EncName*) of the `Log` table. This combination of attributes to create a partition key helps finding each encrypted attribute of the `Log` table for a particular tenant by a simple key lookup without a full table scan. In addition, it neglects the need to create costly (in terms of performance) secondary indexes on non-primary key attributes for search reasons. The *Table* attribute contains the name of the table for which the data is encrypted and thus allow us to store confidential data from multiple tables in a single encrypted table. The (*EncName* and *Value*) attributes store the name and the value of encrypted attributes of the `Log` table respectively. The *TID* attribute holds the primary key of the encrypted row of the `Log` table, whereas the *Tenant* attribute contains the information about the tenant, for which the data is encrypted.

Let us resume the multi-tenant log management application, in which the tenant with id 1 (represents a financial agency) enforces all the device information including the meta-data about the device to be highly sensitive, which is combined together, encrypted, and stored in a single row of the `Encrypted_Log` table (shown in white in Figure 3). The *EncName* attribute is specified as an empty value, which represents that the full entity of the `Log` table is encrypted. In the case of fine-grained data encryption, the tenant with id 2 has specified the (*DeviceID* and *DeviceName*) attributes as confidential, which are stored in two different rows in the `Encrypted_Log` table (shown in gray in Figure 3). The first row contains encrypted information about the *DeviceID*, whereas the second row holds the encrypted information about the *DeviceName*. Similarly, for the tenant with id 3, (*DeviceID*, *DeviceName*, and *DeviceType*) attributes contain sensitive information and therefore are stored in three different rows (shown in dark gray in Figure 3). After completing the encryption process, encrypted data of the `Log` table is stored in 6 partitions in the `Encrypted_Log` table (see Figure 3) and the `Log` table is organized schematically in Figure 4.

rows

ID	*DeviceID*	*DeviceName*	*DeviceType*	...	*Tenant*
2	**********	**********	cisco-ace	...	2
3	**********	**********	**********	...	3

Fig. 4: `Log` table after applying the encryption process.

D. Limitations in Current State-of-Practice

Data encryption impacts performance significantly. However, in the state-of-practice, data separation (i.e. separating sensitive data from the non-sensitive data) without considering an efficient data mapping strategy and the underlying data storage model, results in an inefficient implementation.

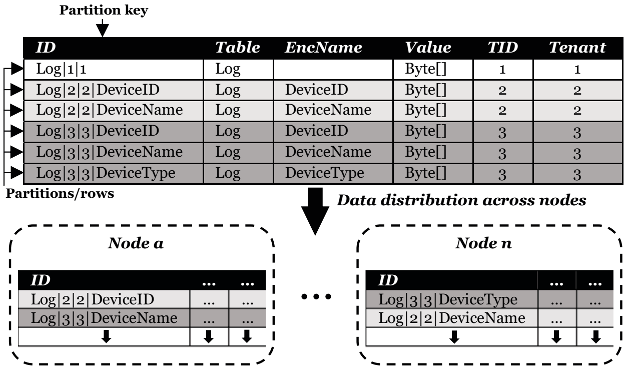
Let us consider the confidentiality requirements of the tenant with id 3, which represents a medical institution and enforces encryption on (*DeviceID*, *DeviceName*, and *DeviceType*) attributes. After applying the encryption process as discussed in Section II-C, each of these encrypted attributes is stored as a separate row in the `Encrypted_Log` table (shown in dark gray in Figure 3). This creates one-to-many mapping between the `Log` table (contains plaintext, see Figure 1) and the `Encrypted_Log` table (contains ciphertext, see Figure 3). The one-to-many mapping implies that the attributes, which

---

[1] We further discuss these solutions in Section V.

need to be encrypted for a particular tenant in a single row of the `Log` table are stored in multiple rows in the `Encrypted_Log` table. For example, encrypting 3 attributes of the `Log` table for the tenant with id 3, 3 rows are added in the `Encrypted_Log` table, each representing a single encrypted attribute (shown in dark gray in Figure 3).

This significantly impacts the performance when reading, updating, and deleting a single row of the `Log` table, which involves several attributes to be encrypted. In that case, multiple rows of the `Encrypted_Log` table will be needed to consider to resemble a single row of the `Log` table. The performance impact is even worse in a distributed setup, where the data among the nodes is distributed based on the partition key. In such a setup, attributes of a single row of the `Log` table, which are encrypted and stored in multiple rows in the `Encrypted_Log` table (see Figure 3), distribute across multiple nodes in a cluster. In a distributed environment, each row of the `Encrypted_Log` table belongs to a specific node in a cluster. As illustrated in Figure 1, encrypted data of a single row of the `Log` table which belongs to the tenant with id 3 and stored in multiple rows in the `Encrypted_Log` table (shown in dark gray in Figure 3) is distributed across multiple nodes (i.e. Node a ... Node n) in a cluster (shown in dark gray in Figure 5).



ID	Table	EncName	Value	TID	Tenant
Log|1|1	Log		Byte[]	1	1
Log|2|2|DeviceID	Log	DeviceID	Byte[]	2	2
Log|2|2|DeviceName	Log	DeviceName	Byte[]	2	2
Log|3|3|DeviceID	Log	DeviceID	Byte[]	3	3
Log|3|3|DeviceName	Log	DeviceName	Byte[]	3	3
Log|3|3|DeviceType	Log	DeviceType	Byte[]	3	3

ID	...	...
Log|2|2|DeviceID	...	...
Log|3|3|DeviceName	...	...

ID	...	...
Log|3|3|DeviceType	...	...
Log|2|2|DeviceName	...	...

Fig. 5: Encrypted attributes of a single row of the `Log` table are partitioned across multiple nodes in a distributed setup.

Consequently, considering more attributes of a row for a particular tenant to be encrypted, additional rows will be inserted in the `Encrypted_Log` table. In a distributed setup, this leads newly inserted rows to be either stored on (i) existing nodes that contain the same row data, but rows will be added in a vertical fashion (see Figure 5), or (ii) different nodes in a cluster than the one that already contains a part of the row data. Therefore, read, update, and delete operations come with an additional performance degradation. For example, to perform such operations in a distributed setup, first multiple nodes in a cluster need to be coordinated (i.e. inter-node communication is required). Then, multiple rows within a node need to be considered to perform a single read, update, and delete operation.

III. LEVERAGING NOSQL FOR SCALABLE AND DYNAMIC ENCRYPTION

As discussed in the previous section, encryption impacts performance significantly due to the computational complexity and an inefficient implementation. The inefficiency is mostly related to the management and the distribution of encrypted data. This, in turn, causes an additional performance degradation than the impact introduced by the encryption itself. The key challenge is to perform encryption without incurring too much performance overhead. Therefore, designing a solution to support scalable, dynamic, and fine-grained data encryption requires, among other things, an understanding of the underlying database storage model.

In order to support scalable and dynamic data encryption for multi-tenant SaaS applications, we need an alternative data mapping strategy. As the confidentiality requirements of each tenant are variable, the flexibility of the Cassandra data model makes a clear and convenient choice to deal with the problem of data encryption. Therefore, our proposed solution leverages the flexibility of the Cassandra data model, in terms of its ability to support dynamic columns. In the underlying data storage model, we have also focused on separating the sensitive data from the non-sensitive data. The separation is clearly desirable when encryption at various levels of granularity at run-time is required, which demands flexibility in the data model and may also differ considerably among tenants. However, we have considered a mapping strategy that avoids data fragmentation across nodes, while also achieving high performance lookup and linear scalability.

We implemented and integrated this strategy in the *middleware layer* and thus it is generic, reusable, and transparent to the application. The middleware provides a set of annotations to be used in the application to specify different data confidentiality requirements at various levels of granularity. Any application which requires dynamic and fine-grained data encryption can be modelled on top of our middleware without having to re-write an extensive source code in the application.

In the underlying data storage model, the *middleware layer* changes the mapping strategy for the table which stores sensitive data. As an example, in the case of multi-tenant log management application, the middleware modeled the `Encrypted_Log` table, which stores ciphertext using a composite partition key and a simple clustering key. The composite partition key and a simple clustering key are made up of multiple attributes of an entity. For example, (*Table*, *ID*, and *Tenant*) attributes are used as a partition key, whereas the *EncName* attribute is used as a clustering key as shown in Figure 6. In a distributed setup, partition key is responsible for the data distribution across the nodes and the data within the partition is sorted based on the clustering key.

Let us consider again the multi-tenant log management application in which the tenants impose confidentiality requirements at increasing levels of granularity. After applying the encryption process, the `Encrypted_Log` table contains 3 partitions (see Figure 6) instead of 6 partitions in the state-

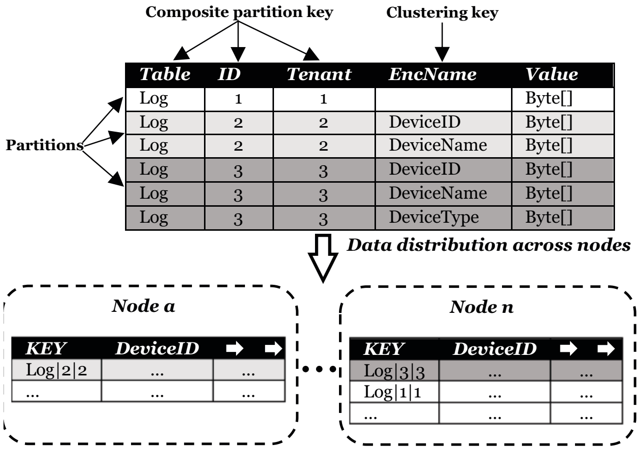


Table	ID	Tenant	EncName	Value
Log	1	1		Byte[]
Log	2	2	DeviceID	Byte[]
Log	2	2	DeviceName	Byte[]
Log	3	3	DeviceID	Byte[]
Log	3	3	DeviceName	Byte[]
Log	3	3	DeviceType	Byte[]

KEY	DeviceID	➡ ➡
Log|2|2	...	...
...	...	...

KEY	DeviceID	➡ ➡
Log|3|3	...	...
Log|1|1	...	...
...	...	...

Fig. 6: All encrypted attributes of a single row of the `Encrypted_Log` are stored exactly on one node in a distributed setup.

of-practice solution (see Figure 5). Each partition represents a single row of the `Log` table for which the data is encrypted. Therefore, instead of keeping one-to-many mapping between the `Log` table (see Figure 1) and the `Encrypted_Log` table (see Figure 3) in the state-of-practice solution, the proposed solution has a one-to-one mapping between the `Log` table and the `Encrypted_Log` table. This implies, all sensitive data of a single row of the `Log` table is mapped and stored exactly in a single row of the `Encrypted_Log` table. For example, consider confidentiality requirements of the tenant with id 3 which specifies (*DeviceID*, *DeviceName*, and *DeviceType*) attributes as sensitive. Instead of storing each attribute, which needs to be encrypted of a single row of the `Log` table in a separate row, all encrypted attributes of a single row for a particular tenant are stored in one row (i.e. one partition), but in different columns where each column represents one encrypted attribute. Considering more attributes of a single row of the tenant with id 3 to be encrypted, more columns are added to that row (in a horizontal fashion). This also makes sure that enforcing encryption at any level of granularity in a single row of the `Log` table will end up encrypted data being stored in a single row of the `Encrypted_Log` table.

In case of a distributed setup where the data is distributed based on the partition key, sensitive data within a single row in our proposed solution is not scattered across multiple nodes. As shown in Figure 6, each partition is stored exactly on one node. For example, all the encrypted data of the tenant with id 2 (shown in gray in Figure 6) is stored in a single row on one node (i.e. Node a). Similarly, all the encrypted attributes of the tenant with id 3 (shown in dark gray in Figure 6) are stored in a single row on another node (i.e. Node n).

Consequently, to perform read, update, and delete operations in a non-distributed setup, a single row of the `Encrypted_Log` table needs to be considered to perform an operation on the `Log` table. Similarly, in a distributed setup, only one node holds all the encrypted data of a single row of the `Log` table, which is stored in the `Encrypted_Log` table. Thus, inter-node communication is not required to perform an operation. Therefore, the performance is significantly better than that of the approach discussed in the previous section.

IV. EVALUATION

The previous section presented our proposed solution, which leverages the flexible data model of a columnar NoSQL database such as Apache Cassandra to realize scalable and dynamic data encryption. This section further evaluates the benefits offered by our proposed solution in terms of good performance and better scalability to deal with the problem of data encryption at various levels of granularity.

In Section IV-A, we first briefly describe our experimental setup and give the essential details of prototype implementations used for the evaluation. Then, we discuss the deployment setups and characterize different workloads in which these prototype implementations are evaluated. Finally, we present our experimental results in Section IV-B.

A. Experimental Setup

We implemented two prototypes[2] which support encryption at various levels of granularity: one is based on the state-of-practice solution described in Section II-C (**Prototype A**) and the other is based on our proposed solution discussed in Section III (**Prototype B**). Consequently, the Prototype A does not take into account an efficient data mapping strategy and the underlying data storage model, whereas the Prototype B takes these factors into consideration. Both prototypes were implemented on top of the Kundera platform [1] by extending it with a support to execute encryption-enabled CRUD transactions at differing levels of granularity.

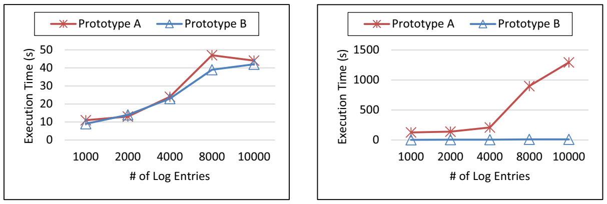
The prototypes are evaluated in two different deployment environment setups: a single node setup and a cluster setup. In a single node setup, both client and server processes run on the same physical machine in which the client process runs the implemented prototypes, whereas a single Apache Cassandra service is deployed as the server process. In a cluster setup, client and server processes operate on different physical machines. We create a cluster consisting of 3 nodes as the server process in which the Apache Cassandra service is deployed and run implemented prototypes on a separate physical node as the client process.

In our experimental setup, the client/single node is equipped with Intel(R) Core(TM) i5 CPU (@ 2.60GHz) Dual and 8 GB RAM, running Windows 8 and Apache Cassandra 2.0.6. In case of a cluster setup, the client node connects to distributed Cassandra consisting of 3 nodes, each consists of Intel(R) Core(TM) 6400 (@ 2.13GHz) and 4 GB RAM, running 64-bit Ubuntu 14.04 LTS and Cassandra 2.0.6.

The experiments are performed for the multi-tenant LMaaS application under the condition of encryption-enabled workloads. We start our measurements with $1,000$ `Log` entries and increase that number up to $10,000$ `Log` entries to see

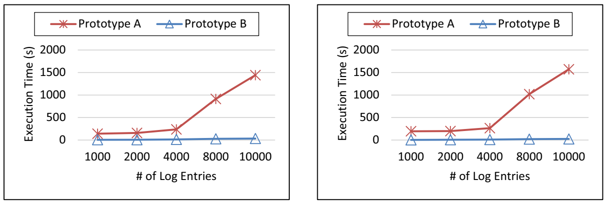
---

[2] Both prototypes are available at: https://people.cs.kuleuven.be/ansar.rafique/BDSE2017-Prototypes.zip.



(a) Execution time for the insert operation

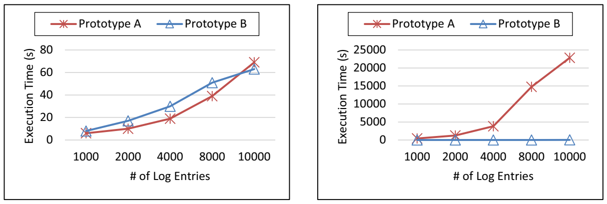
(b) Execution time for the read operation



(c) Execution time for the update operation.

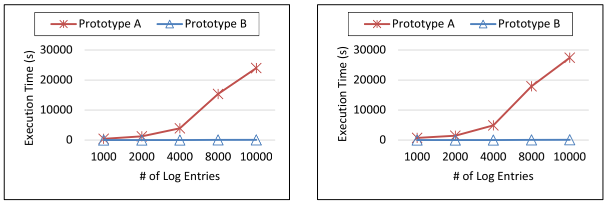
(d) Execution time for the delete operation.

Fig. 7: Total execution time to perform encryption-enabled CRUD transactions for both prototype implementations under different workloads on a **single node setup** with 3 encrypted attributes.



(a) Execution time for the insert operation.

(b) Execution time for the read operation.



(c) Execution time for the update operation.

(d) Execution time for the delete operation.

Fig. 8: Total execution time to perform encryption-enabled CRUD transactions for both prototype implementations under different workloads on a **cluster setup consisting of** 3 **nodes** with 3 encrypted attributes.

how both prototypes scale when the data size increases. We specifically measure (i) the performance in terms of execution time for both prototypes by executing encryption-enabled CRUD transactions under different workloads with the data size ranging between $1,000$ and $10,000$ `Log` entries, (ii) the performance impact on both prototype implementations under different workloads at various levels of granularity when the database service scales out (i.e. # of nodes), and (iii) the cost of executing CRUD transactions under the condition of encryption-enabled workloads at various levels of granularity (i.e. # of attributes). Beyond these experiments, we also run additional benchmarks for both prototypes where we scale out the database service and varied the number of attributes to see the impact on CRUD transactions. However, due to space limitations, we only focus on the above measurements.

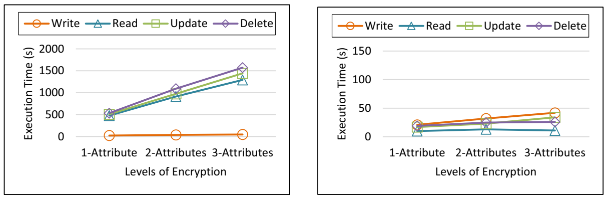
B. Experimental Results

In this section, we analyze the performance results to quantify the performance impact of both prototypes.

The results for encryption-enabled CRUD transactions at the fine-grained level (i.e. encrypting 3 attributes of an entity) with the data size ranging between $1,000$ and $10,000$ `Log` entries on a single node setup, are presented in Figure 7 and on a cluster setup when the database service scales out are presented in Figure 8.

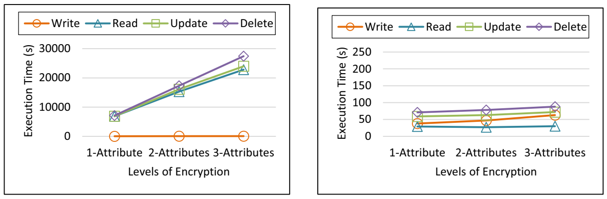
As shown in Figure 7(a), there are no variations, both prototypes take almost the same amount of time to perform an insert operation for varying data size. However, for read, update, and delete operations as shown in Figure 7(b), Figure 7(c), and Figure 7(d) respectively, the execution time of the Prototype A, which is based on the state-of-practice is higher than the Prototype B, which is built on our proposed solution. The primary reason is the underlying distribution of encrypted data. For the Prototype A, each encrypted attribute of an entity is stored in a separate row. Therefore, multiple rows needed to be read first and then decrypted to construct a single row of the `Log` table. In addition, the execution time of the Prototype A drastically increases with the increase in the data size. On the other hand, as we can see that the execution time of the Prototype B remains relatively the same. This is mainly because of the underlying data mapping strategy where all encrypted attributes of a single row (i.e. an entity) for a particular tenant are stored in one row. Therefore, a single row, which contains ciphertext (i.e. encrypted attributes) needed to be read and decrypted to construct a row of the `Log` table, which contains plaintext.

In case of a distributed setup, as depicted in Figure 8(a), the execution time of the prototype A for the insert operation is less than the execution time of the Prototype B for different data size with small variations in execution time. The reason that the Prototype B takes slightly more time than the Prototype A is mainly because the Prototype B needs to find the node, which is responsible for storing the data to be able to co-locate all encrypted attributes on the same node. However, the differences between both prototypes are clearly visible in Figure 8(b), Figure 8(c), and Figure 8(d) for read, update, and delete operations respectively. As we can see that Prototype A takes more time to execute read, update, and delete operations when the database service scales out. In order to read, update, and delete a single row (i.e. an entity) using the Prototype A, multiple nodes need to be coordinated first and then within each node, multiple rows need to be manipulated. In addition, the execution time of the Prototype A increases significantly when the data size increases. On the other hand, the data size



(a) Execution time for the **Prototype A** on a **single node**.

(b) Execution time for the **Prototype B** on a **single node**.



(c) Execution time for the **Prototype A** on a **cluster**.

(d) Execution time for the **Prototype B** on a **cluster**.

Fig. 9: Total execution time for $10,000$ `Log` entries with increasing # of attributes on a **single node setup** and a **cluster setup consisting of** 3 **nodes**.

has no impact on the Prototype B. In order to read, update, and delete a single entity, only a single node needs to be considered and within that node, a single row needs to be manipulated. Therefore, the Prototype B, which is built on our proposed solution is better performance-wise and is more scalable than the prototype A.

It is important to note that the execution time of the Prototype B for read, update, and delete operations on both deployment setups (i.e. a single node setup and a cluster setup) increases in a linear fashion with the number of `Log` entries (not very visible in Figures 7 and 8). However, the execution time of the Prototype B is negligible compared to the execution time of the Prototype A especially when the data size increases from $4,000$ `Log` entries as shown in Figure 7 and Figure 8.

The results of executing encryption-enabled CRUD transactions at various levels of granularity (i.e. # of attributes) on a single node setup are shown in Figures 9(a) and 9(b) for Prototypes A and B respectively and on a cluster setup are outlined in Figure 9(c) for the Prototype A and Figure 9(d) for the Prototype B.

As shown, the execution time of the Prototype A for the insert operation on a single node setup is the same for different levels of granularity, with only small variations visible when the number of attributes, which need to be encrypted increase. However, by increasing the number of attributes to be encrypted, the execution time increases rapidly for read, update, and delete operations for the Prototype A. For the Prototype B, as illustrated in Figure 9(b), the execution time for read, update, and delete operations remains more or less the same with only small variations.

As shown in Figure 9(c), in case when the database service scales out, the performance degradation is visible for the read, update, and delete operations for the Prototype A. However, this does not affect the Prototype B where the execution time remains the same, when changing numbers of attributes are considered for encryption as illustrated in Figure 9(d).

V. RELATED WORK

Data security is the most important issue of cloud storage and has been a major concern for a long period of time. Recently, there has been extensive research focusing on highlighting the security issues in NoSQL databases [8], [16], [21], [23]. To end this, several research contributions [5], [13], [22], [24] have been made, which provide encryption support for NoSQL databases at different layers to protect outsourced data.

The research conducted by Tian et al. [24] focuses on data encryption at the middleware layer. The authors proposed a middleware for data encryption in the MongoDB. The key differences with our research are: (i) they require to specify the confidential data during the deployment time, whereas our research perspective is on-the-fly (i.e. run-time) data encryption, (ii) our research focuses on providing a support for data encryption for different tenants at various levels of granularity, whereas they do not consider this explicitly, and (iii) their main focus is on transparency, whereas transparency is our secondary focus with a primary focus on supporting fine-grained data encryption, while also achieving good performance and better scalability.

The existing solutions in the state-of-practice [11], [14] to provide encryption support at the middleware layer either (i) offer limited support for the data encryption where the attributes with only specific data types can be encrypted, or (ii) provide solution-specific data types which should be used in the application to encrypt the sensitive data.

A number of libraries are available in different programming languages, providing encryption support in the application layer. For example, one of the easiest ways to encrypt sensitive data in Java is by using custom data types provided by the Java simplified encryption (Jasypt) [11]. Jasypt is a Java library which facilitates developers to add encryption support with minimum effort. However, these libraries have some limitations: (i) they need to be configured to specify sensitive data during deployment time, (ii) they only support attribute-level data encryption and provide their own data types to be used in the application code to specify sensitive data, hence require modifications in the application layer, and (iii) they do not support encryption at various-levels of granularity, which can also be altered during run-time. Although, these libraries can be modified to support encryption for various NoSQL databases, but at this moment, no contribution has been done in this direction and the support is limited to relational databases.

In another related work, the authors proposed TEAL [14] that provides transparent encryption support in the abstraction layer. The main objective of TEAL is to facilitate developers to re-use existing software components and seamlessly migrate to secure storage services. However, they do not consider secure storage at differing levels of granularity and only operate on relational databases, which share the same data model, whereas our research targets NoSQL databases, which follow heterogeneous data models.

To the best of our knowledge, this is the first effort that purposes such data mapping strategy that leverages schema flexibility of NoSQL to achieve scalable, dynamic, and fine-grained data encryption for multi-tenant SaaS applications.

VI. CONCLUSION

Outsourcing data to external third-party cloud storage providers offers a wide array of clear benefits over hosting data in on premise data centers. In practice, however, data confidentiality considerations often prohibit outsourcing confidential data to external and often untrusted storage providers.

This paper presented a data mapping strategy that leverages the data schema flexibility of columnar NoSQL databases to support dynamic yet scalable data encryption that can be enacted at different levels of granularity. Our prototype implementation has validated this data mapping strategy in the context of a multi-tenant SaaS application, a Log Management-as-a-Service (LMaaS) offering, and we have performed a comprehensive evaluation of the performance and scalability benefits. These results can promote further research and development of critical applications based on our proposed solution, where the data confidentiality requirements are much higher, varies greatly, and also the performance is the key factor.

This work fits into our ongoing research on application-level middleware for federated data storage architectures in support of multi-tenant SaaS applications. We envision the creation of policy-driven storage mechanisms that select the most appropriate data mapping strategy dynamically, based on customer SLAs and available storage resources. In future work, we will also explore different data mapping strategies and further investigate performance optimization, by providing improved data distribution support, for example by grouping all sensitive and non-sensitive data belonging to the same entity, and storing them together.

ACKNOWLEDGMENT

This research is partially funded by the Research Fund KU Leuven (project GOA/14/003 - ADDIS), the SBO DeCoMAdS project, and the imec SeClosed project.

REFERENCES

[1] Kundera. https://github.com/impetus-opensource/Kundera. [Last visited on May 02, 2017].

[2] Apache. Cassandra. http://cassandra.apache.org/. [Last visited on May 02, 2017].

[3] D-Base. Decentralized support for Business processes in Application Services . http://www.iminds.be/en/projects/d-base, 2016. [Last visited on June 14, 2016].

[4] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[5] Y. Ding and K. Klein. Model-driven application-level encryption for the privacy of e-health data. In *ARES '10 International Conference on Availability, Reliability, and Security*, pages 341–346, Feb 2010.

[6] [DMS]2. Decentralized Data Management and Migration for SaaS (iMinds ICON project). http://www.iminds.be/en/projects/DMS2, 2016. [Last visited on June 14, 2016].

[7] Katarina Grolinger, Wilson A. Higashino, Abhinav Tiwari, and Miriam Am Capretz. Data management in cloud environments: Nosql and newsql data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):22, 2013.

[8] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. The rise of big data on cloud computing: Review and open research issues. *Information Systems*, 47:98 – 115, 2015.

[9] J. Hu and A. Klein. A benchmark of transparent data encryption for migration of web applications in the cloud. In *Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC '09.)*, pages 735–740, Dec 2009.

[10] Bala Iyer, Sharad Mehrotra, Einar Mykletun, Gene Tsudik, and Yonghua Wu. A framework for efficient storage security in rdbms. In *International Conference on Extending Database Technology*, pages 147–164. Springer, 2004.

[11] Jasypt. Java Simplified Encryption. http://www.jasypt.org/, 2016. [Last visited on July 19, 2016].

[12] L. M. Kaufman. Data security in the world of cloud computing. *IEEE Security Privacy*, 7(4):61–64, July 2009.

[13] Lianzhong Liu and Jingfen Gai. A new lightweight database encryption scheme transparent to applications. In *2008 6th IEEE International Conference on Industrial Informatics*, pages 135–140, July 2008.

[14] Karl Lorey, Erik Buchmann, and Klemens Bohm. Teal: Transparent encryption for the database abstraction layer. In *Proceedings of the CAiSE16 Forum at the 28th International Conference on Advanced Information Systems Engineering*, New York, NY, USA , 2016.

[15] Ulf T. Mattsson. Database encryption - how to balance security with performance. http://papers.ssrn.com/sol3/papers.cfm?abstract_id=670561. [Last visited on June 17, 2016].

[16] L. Okman, N. Gal-Oz, Y. Gonen, E. Gudes, and J. Abramov. Security issues in nosql databases. In *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 541–547, Nov 2011.

[17] A. Rafique, S. Walraven, B. Lagaisse, T. Desair, and W. Joosen. Towards portability and interoperability support in middleware for hybrid clouds. In *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 7–12, April 2014.

[18] Ansar Rafique, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen. Policy-driven data management middleware for multi-cloud storage in multi-tenant saas. In *2015 IEEE/ACM 2nd International Symposium on Big Data Computing (BDC)*, pages 78–84, Dec 2015.

[19] Ansar Rafique, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen. On the performance impact of data access middleware for nosql data stores. *IEEE Transactions on Cloud Computing*, PP(99):1–1, 2016.

[20] Ansar Rafique, Dimitri Van Landuyt, Vincent Reniers, and Wouter Joosen. Towards scalable and dynamic data encryption for multi-tenant saas. In *ACM SIGAPP Symposium On Applied Computing*, April 2017.

[21] A. Ron, A. Shulman-Peleg, and A. Puzanov. Analysis and mitigation of nosql injections. *IEEE Security Privacy*, 14(2):30–39, Mar 2016.

[22] V. Sidorov and W. K. Ng. Transparent data encryption for data-in-use and data-at-rest in a cloud-based database-as-a-service solution. In *2015 IEEE World Congress on Services*, pages 221–228, June 2015.

[23] D. S. Terzi, R. Terzi, and S. Sagiroglu. A survey on security and privacy issues in big data. In *10th International Conference for Internet Technology and Secured Transactions*, pages 202–207, Dec 2015.

[24] Xingbang Tian, Baohua Huang, and Min Wu. A transparent middleware for encrypting data in mongodb. In *2014 IEEE Workshop on Electronics, Computer and Applications*, pages 906–909, May 2014.

[25] Tim Waage and Lena Wiese. *Foundations and Practice of Security: 7th International Symposium, FPS 2014, Montreal, QC, Canada, November 3-5, 2014. Revised Selected Papers*, chapter Benchmarking Encrypted Data Storage in HBase and Cassandra with YCSB, pages 311–325. Springer International Publishing, Cham, 2015.

[26] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data consistency properties and the trade-offs in commercial cloud storages: the consumers' perspective. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR '11)*, pages 134–143, Asilomar, California USA, jan 2011.

[27] B. Wang, S. S. M. Chow, M. Li, and H. Li. Storing shared data on the cloud via security-mediator. In *IEEE 33rd International Conference on Distributed Computing Systems (ICDCS)*, pages 124–133, July 2013.

[28] Yonghong Yu and Wenyang Bai. Enforcing data privacy and user privacy over outsourced database service. *Journal of Software*, 6(3):404–412.