Information Systems 96 (2021) 101671

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

CryptDICE: Distributed data protection system for secure cloud data storage and computation

Ansar Rafique [*], Dimitri Van Landuyt, Emad Heydari Beni, Bert Lagaisse, Wouter Joosen

imec-DistriNet, KU Leuven, 3001 Leuven, Belgium

A R T I C L E   I N F O

Article history:
Received 11 June 2020
Received in revised form 26 September 2020
Accepted 20 October 2020
Available online 30 October 2020
Recommended by Alysson Neves Bessani

Keywords:
Data security and privacy
NoSQL databases
Search over encrypted data
Database-as-a-Service
Data encryption
Cloud computing
Query processing
Computation over encrypted data

A B S T R A C T

Cloud storage allows organizations to store data at remote sites of service providers. Although cloud storage services offer numerous benefits, they also involve new risks and challenges with respect to data security and privacy aspects. To preserve confidentiality, data must be encrypted before outsourcing to the cloud. Although this approach protects the security and privacy aspects of data, it also impedes regular functionality such as executing queries and performing analytical computations. To address this concern, specific data encryption schemes (e.g., deterministic, random, homomorphic, order-preserving, etc.) can be adopted that still support the execution of different types of queries (e.g., equality search, full-text search, etc.) over encrypted data.

However, these specialized data encryption schemes have to be implemented and integrated in the application and their adoption introduces an extra layer of complexity in the application code. Moreover, as these schemes imply trade-offs between performance and security, storage efficiency, etc, making the appropriate trade-off is a challenging and non-trivial task. In addition, to support aggregate queries, User Defined Functions (UDF) have to be implemented directly in the database engine and these implementations are specific to each underlying data storage technology, which demands expert knowledge and in turn increases management complexity.

In this paper, we introduce CryptDICE, a distributed data protection system that (i) provides built-in support for a number of different data encryption schemes, made accessible via annotations that represent application-specific (search) requirements; (ii) supports making appropriate trade-offs and execution of these encryption decisions at diverse levels of data granularity; and (iii) integrates a lightweight service that performs dynamic deployment of User Defined Functions (UDF) –without performing any alteration directly in the database engine– for heterogeneous NoSQL databases in order to realize low-latency aggregate queries and also to avoid expensive data shuffling (from the cloud to an on-premise data center). We have validated CryptDICE in the context of a realistic industrial SaaS application and carried out an extensive functional validation, which shows the applicability of the middleware platform. In addition, our experimental evaluation efforts confirm that the performance overhead of CryptDICE is acceptable and validates the performance optimizations for achieving low-latency aggregate queries.

1. Introduction

The emergence of Cloud computing has led to a paradigm shift, not only in the technological and business landscape, but also in the database landscape [1,2], as illustrated with the emergence of delivery models such as Database-as-a-Service (DBaaS) [3]. Cloud storage services enables data owners –individuals and organizations– to store their data remotely in a flexible and on-demand manner, without taking on the responsibility for provisioning, configuring, scaling, and maintaining these storage systems [4].

In the context of cloud storage, one of the biggest challenges is to provide data management support for cloud-based applications in an efficient and scalable manner [5]. The need to support data-intensive cloud applications in an efficient and scalable manner have gained substantial interest, and led to the development of cloud-friendly database technologies, commonly known under the umbrella term of *NoSQL*. NoSQL databases are built from the ground up to scale horizontally just by simply adding more nodes. As such, they yield numerous benefits in terms of high availability, elastic scalability, and data model flexibility —concerns that are particularly relevant in cloud computing and more specifically in cloud storage [6].

* Corresponding author.
E-mail address: Ansar.Rafique@cs.kuleuven.be (A. Rafique).
URL: https://distrinet.cs.kuleuven.be/people/ansarr (A. Rafique).

https://doi.org/10.1016/j.is.2020.101671
0306-4379/© 2020 Elsevier Ltd. All rights reserved.

Although cloud data storage provides numerous benefits to organizations, there are also caveats that significantly hindering its rapid and wider adoption. In essence, the DBaaS delivery model requires and assumes a degree of trust in the provider that will not be realistic or desirable in different real-world application contexts. Many applications involve storing sensitive information that when compromised will seriously jeopardize the privacy of individuals and violate data protection laws such as the GDPR. Recent data security breaches and their impact on a large number of individuals and organizations have exacerbated these concerns [7–9]. In practice, data security and privacy protection are among the most important factors when choosing a database for cloud-based applications [10].

NoSQL databases, which are prominently used in a cloud environment do not provide strong built-in security mechanisms and thus rely on developers to engage with a wide range of data protection measures from within the application code [10–15]. Although adopting these measures in application will lead to an adequate protection of the data, many of them impose non-trivial trade-offs: for example, the use of data encryption before persisting data has implications on the ability to execute different types of search (e.g., equality search, full-text search, etc.) and aggregate queries. To address this concern, a number of different data encryption schemes (e.g., deterministic, random, homomorphic, etc.) have been proposed [16–21], which can be used to execute different types of queries and perform complex computation over encrypted data.

However, the approach to adopt specific data encryption schemes to support different types of queries comes with several non-trivial challenges. Firstly, as these data encryption schemes are commonly integrated in the application layer to support different types of search and aggregate queries over encrypted data, an extra layer of complexity is introduced in the application, and a level of expertise is required from application developers. Secondly, as these different data encryption schemes have specific security strengths and weaknesses (e.g., the random encryption scheme offers greater data security strength than other encryption schemes, but has no built-in support for executing queries), trade-offs need to be made between strong security, increased performance, and rich query capabilities. For example, enforcing strong data security requirements can lead to a system that is less performance-oriented and offers limited query capabilities. Similarly, disregarding privacy towards increased performance and rich query capabilities can lead to pushing off critical security requirements. Therefore, making appropriate trade-offs is a non-trivial task which highly depends on the application requirements and on the limitations imposed to sensitive data. This task becomes more complex when different types of data with varying privacy requirements are considered. Thirdly and finally, to support aggregate queries in the application requires User Defined Functions (UDF) to be supported directly within the database engine, which not only demands expert knowledge and introduces additional management complexity, but also raises additional security concerns.

To address the above-mentioned concerns, we present CryptDICE, a flexible, generic, reusable, and distributed data protection system that facilitates building applications that involve encrypted data storage and search, but does not require an in-depth understanding of different data encryption schemes. To address the problems highlighted above, CryptDICE (i) provides built-in support for several different data encryption schemes (by integrating a number of established libraries), yet hides the complexity from the developer via annotations, which steer the selection of the most appropriate scheme for a given (search) requirement; (ii) supports trade-offs between performance and security and enables executing different types of search and aggregate queries over encrypted data for a variety of different NoSQL databases; (iii) incorporates a lightweight service that reduces the management complexity and also mitigates high-security risks by preventing developers from implementing UDF directly in the database engine. The latter service –which has built-in support for heterogeneous NoSQL databases– rather implements UDF in the application code and provides migration transparency (from on-premise to the cloud) in order to perform complex computations next to the database engine purely for the sake of performance, i.e., to realize low-latency aggregate queries and also to avoid expensive data shuffling (from cloud to an on-premise data center).

There exists several individual implementations [22,23] and combined libraries [7,24–26] that can be used by software developers, but to our knowledge, an integrated and developer-friendly framework such as CryptDICE that reduces the implementation and management complexity from a developer point of view and offers performance optimization, is lacking. We have validated a prototype implementation of CryptDICE in the context of a realistic industrial Software-as-a-Service (SaaS) application, carried out an extensive functional validation, and also conducted a thorough experimental evaluation. The evaluation results confirm that CryptDICE significantly reduces the required development time for enabling data encryption and supporting different types of interactive search queries over encrypted data as well as offers performance optimizations for achieving low-latency aggregate queries. We have also conducted a thorough experimental evaluation to analyze the performance overhead of CryptDICE, which is shown to be negligible.

The remainder of this paper is structured as follows: Section 2 provides relevant background information and derives the problem statement motivated by a realistic industrial SaaS application case. Section 3 presents the design of our proposed CryptDICE system, while Section 4 details the prototype based on CryptDICE. Section 5 presents our extensive evaluation of the CryptDICE system in three different aspects. We contrast our solution with related works in Section 6. Finally, Section 7 concludes this paper and indicates directions for future research.

2. Motivation

The motivation for this work is based on our experiences with large-scale Software-as-a-Service (SaaS) applications, which stem from several applied research projects. These projects have been carried out in active collaboration with industrial SaaS application providers. For simplicity of illustration, we focus on one such application case from the financial domain, a Billing-as-a-Service document management SaaS application, which is introduced in Section 2.1. More specifically, we highlight the problem of lack of trust in cloud storage services that are used by this SaaS application. Then, Section 2.2 first gives a high-level overview of NoSQL databases that are increasingly being used in cloud-based storage services and then provides a comparison of data security features supported in popular NoSQL databases. Next, Section 2.3 presents the specific details of different data encryption schemes that are used to perform search and computation over encrypted data. Finally, Section 2.4 elaborates on the problem statement.

2.1. Billing-as-a-Service SaaS application

Cloud computing in general and SaaS, in particular, have become increasingly popular and equally important for many ICT actors in both business-to-business (B2B) and business-to-consumer (B2C) markets. Take an example of a billing process, which is an unavoidable, time-consuming, and recurring task for any organization. Therefore by outsourcing the billing process

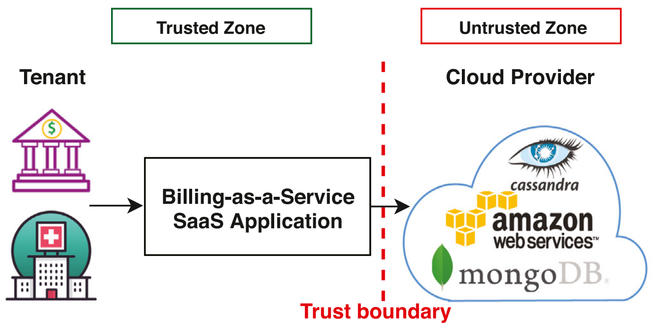

Fig. 1. The use of third-party storage services involves traversing a trust boundary between the users, tenants and SaaS provider on the one hand and the cloud-based storage services on the other hand.

to specialized third-party solution providers, a reduction in operating expenses can be achieved. The Billing-as-a-Service SaaS application is a B2B cloud offering that makes the recurring billing process as seamless as possible and provides a wide range of financial services –including the management of financial documents such as invoices, bills, accounts, payments– to its customer organizations (aka tenants). The tenants of this SaaS application are customer organizations of all sizes from different application domains (e.g., banks, hospitals, telecom operators etc.), which impose strict requirements on data protection and privacy. The SaaS application allows tenants to securely manage their financial documents and yet enables them to perform several different types of search operations.

Lack of trust in cloud storage services. Despite the evidential popularity and growing trend of adopting cloud storage for provisioning the persistence of data, the users, and tenant organizations (e.g., banks or hospitals); service providers are essentially required to trust these cloud-based storage services and their providers (as shown in Fig. 1). To manage such trust concerns, for example when storing sensitive data, appropriate data protection steps must be taken before outsourcing data to external storage providers.

2.2. NoSQL databases

In recent years, a fairly large number of database technologies (> 225) have emerged that are collectively referred to as NoSQL. NoSQL databases are non-relational, distributed, horizontally scalable, highly available, and schema-free in nature [27]. Due to these well-known benefits, NoSQL databases are increasingly popular in cloud storage and are widely supported by multiple cloud service providers [7,28]. NoSQL encompass a wide range of technologies that can store different types of data; such as structured, semi-structured, unstructured, and polymorphic data. Most of the NoSQL databases do not support Atomicity, Consistency, Isolation, and Durability (ACID) properties and are instead relying on the basically available, soft state, and eventual consistency (BASE) principles [29]. They sacrifice strong consistency in exchange for high availability and scalability [30] and are classified based on four different data models: key-value databases, wide-column stores, document stores, and graph databases.

Data protection support in NoSQL databases. NoSQL databases have not been designed with security aspects in mind [12,22,31] and most of the NoSQL databases initially did not provide any built-in data security mechanism. However, in recent years, security has been identified as a major area of concern and the issue has prompted the interest and development of data security features in NoSQL databases [32,33]. Table 1 provide a comparison of popular NoSQL databases based on their current support for data security features.

As shown in the third column of Table 1, most of the NoSQL databases support for authentication, for example to gain administrator access. However, the level of support is highly dependent on a certain type of NoSQL database(s). For example, Apache Cassandra, MongoDB, and Neo4J[1] support both internal and external authentication mechanisms, whereas Apache HBase only supports external authentication. The fourth column of Table 1 indicates that most of the NoSQL databases either employ Role-Based Access Control (RBAC) or manage Access Control List (ACL) to give different permissions to users based on their roles and to govern access to a database system. As an example, Apache Cassandra, MongoDB, and Neo4J employ RBAC to restrict access to users through assigned roles, whereas Apache HBase, CouchDB, Redis, and Riak manage ACL that provides users access to particular portions of data and as such limits their access in terms of the commands that they can execute. The fifth column of the table gives information about the communication protocol that these NoSQL databases rely on to establish secure communication between application and database servers. As shown, most of the NoSQL databases either use the Transport Layer Security (TLS) or the Secure Shell (SSH) to protect data-at-transit from client machines to a database cluster and as such prevent accidental or deliberate attempts to read data.

The final column of Table 1 summarizes the support for data protection in NoSQL databases. In relational databases, there are multiple levels at which encryption is supported. For example, most relational databases provide a Transparent Data Encryption (TDE) feature that encrypts the entire database at rest at the lowest level –the storage-level– and hence requires no modification in the database or application code. This enables data to be transparently encrypted and decrypted for database users and as such allowing easier administration of day-to-day encryption operations. As the schema is designed up-front where all the rows have the same columns, relational databases also support Field Level Encryption (FLE), which is a server-side facility that encrypts only selected columns.

In contrast, most of the community editions of NoSQL databases lack such built-in data protection support and store data as plaintext, which incurs many security risks. As shown in the final column of Table 1, there are several enterprise NoSQL database products (e.g., Cassandra, MongoDB, etc.) that use the TDE feature to support encryption in which the entire database files are encrypted at the file system or block level. Similarly, some enterprise NoSQL databases such as CouchDB and Neo4J use TDE provided by third-party on-disk encryption software vendors to support data-at-rest encryption. However, applying encryption at the storage level or the database level is problematic for four main reasons. Firstly, there is a trust boundary between the client and the database (deployed and fully controlled by an external cloud provider), and thus for example, an encryption key is also stored alongside the encrypted data. Secondly, both encryption at storage and database levels only accomplish coarse-grained level encryption and do not address the encryption requirements at the finest level of granularity, which is FLE. For example, they do not offer the flexibility to encrypt only a specific set of columns, which is the most preferred approach since less data is encrypted thus improving on latency. Thirdly, they do not provide the means to enable search on encrypted data, hence lacking the ability to support different types of queries (e.g., equality search, full-text search, etc.) and the potential to perform complex computation. Finally, data is only encrypted when writing to the disk, which

[1] https://neo4j.com/.

Table 1
Comparison of popular NoSQL databases on the basis of their support for built-in data security features.

Database	Data model	Authentication[a]	Authorization	Communication protocol	Data protection[b]
Apache Cassandra	Columnar	Supports both internal and external authentication mechanisms	Employs Role-Based Access Control (RBAC)	TLS	TDE
MongoDB	Document	Supports both internal and external authentication mechanisms	Employs Role-Based Access Control (RBAC)	SSL/TLS	TDE
Apache HBase	Columnar	Supports only external authentication mechanism	Manages Access Control List (ACL)	SSH	TDE
CouchDB	Document	Supports Basic authentication, Cookie authentication, and Proxy authentication	Manages Access Control List (ACL)	TLS	TDE (third-party)
Redis	Key-value	Supports only username/ password authentication	Manages Access Control List (ACL)	SSL/TLS	TDE
Riak	Key-value	Supports S3 authentication and Keystone authentication	Manages Access Control List (ACL)	TLS	Not supported
Voldemort	Key-value	Not supported	Not supported	HTTP	Not supported (?)
MonetDB	Columnar	Supports only username/ password authentication	Not supported	SSH	Not supported
Amazon DynamoDB	Key-value/ Document	Supports S3 authentication	Handles by AWS Identity and Access Management (IAM)	HTTP	TDE
Neo4J	Graph	Supports both internal and external authentication mechanisms	Employs Role-Based Access Control (RBAC)	TLS	TDE (third-party)

[a]External authentication is only supported in enterprise NoSQL database products.
[b]Data protection support is only available in enterprise NoSQL database products that encrypt the whole database files using Transparent Data Encryption (TDE) at the storage level.

implies that the data is not encrypted in RAM and as such enables a root user to read data in RAM (e.g., by attaching the GNU debugger (GDB) [34] to a database).

2.3. Data encryption schemes

Data encryption enables the protection of sensitive data and therefore addresses many privacy-related issues in cloud computing. However, as discussed in the previous section, most of the community editions of NoSQL databases lack such support and thus rely on developers to engage with data protection measures in the application code. As a result, not only complexity is substantially increased, but also executing search queries and performing computation over encrypted data becomes highly inefficient: a simple search query over an encrypted database first involves a full database scan where all the encrypted data is retrieved from the database by the application. Then, each record needs to be decrypted in the application and compared against the conditions of the specified query. Only then a set of matching results can be obtained.

To alleviate this, several different data encryption schemes have emerged, which can be used to execute interactive search queries and also to perform complex computations over encrypted data, without the need to decrypt each database record. Table 2 provides an overview of data encryption schemes and summarizes their built-in support for different types of queries.

Random/probabilistic encryption. Random (RND) is an encryption scheme in which encrypting the same plaintext results in a random ciphertext. This is one of the strongest data protection schemes and can be achieved, for example, by applying the Advanced Encryption Standard (AES) with a random initialization vector (IV) [14]. This scheme, as such, prevents data from being compromised through plaintext attacks and is also considered to be computationally secure [35]. However, this scheme is not designed to produce ciphertexts over which meaningful computations can be performed [22]. The applicability of this scheme is, therefore, limited to the protection of data in cases where no queries are to be performed directly over encrypted data, such as data transmission or data storage [14,22]. As a result, retrieving data encrypted using this encryption scheme would require a full database scan and subsequent decryption, which is not feasible in larger-scale databases.

Deterministic encryption. Deterministic (DET) encryption is a type of encryption scheme in which the resulting converted information (aka ciphertext) can be repeatedly produced, given the same source text (aka plaintext) and key. This can be achieved, for example, by using the AES with fixed IV [14]. In comparison to the RND encryption scheme, which is considered to be highly (computationally) secure, DET encryption is known to be less secure as it provides the information of plaintext-ciphertext pair [14]. This scheme facilitates to perform queries with equality checks over encrypted data. However, to perform range queries over encrypted data using this encryption scheme, first a full database scan, then data retrieval, and finally decryption is required, since equality search does not suffice [22].

Order-preserving encryption. In the order-preserving (OP) encryption scheme, the order relationship between plaintext is preserved after the encryption process. As such, the order relations among the encrypted data is revealed, not the actual data itself. The order comparison is a well-known operation in the database world and it is commonly used for performing operations such as ORDERBY, MIN, MAX, SORT, etc [23]. This encryption scheme is considered to be less secure than the DET encryption scheme [14]. The order-revealing nature of this data encryption scheme allows for comparisons (i.e. range queries that require the order of data) to be performed over encrypted data. This is also the main characteristic that makes it vulnerable to inference attacks where knowledge of data distribution may lead to extraction of confidential data [22,36].

Table 2
Summary of existing data encryption schemes and their potential support for different type of queries, which is denoted by (×) symbol. The (◯) symbol represents the security strength of these encryption schemes, ranging from the strongest (●) to the weakest (◔).

Support	Data encryption schemes			
	Random (RND)	Deterministic (DET)	Order-Preserving (OP)	Homomorphic (HOM)
Data confidentiality	×	×	×	×
Security strength	●	◕	◑	◔
Equality search		×		
Complex boolean search		×		
Range queries			×	
Aggregate queries (sum and average)				×
Full-Text search				
Implementation[a]	Application code	Application code	Application code	Application code/ database function

[a]These data encryption schemes need to be implemented in the application code with the exception of HOM, which can also be implemented and deployed as a database function directly in the underlying database engine. However, such a feature (where a database function can be implemented and deployed directly in the database engine) is not available in all NoSQL databases.

Homomorphic encryption. Homomorphic (HOM) encryption schemes support queries that require computation (i.e. aggregate queries) to be performed directly on the encrypted data without requiring access to a secret key (i.e. private key) and decrypting the encrypted data [37,38]. The result of such a computation also remains in an encrypted form and can be revealed by the owner of the secret key [39]. In recent years, research on fully homomorphic (HOM) encryption has resulted in significant advances that make it possible to perform arbitrary computations on encrypted data. However, the computation involved in fully homomorphic encryption is still prohibitively high, which makes it unsuitable for resource-constrained systems [35].

2.4. Problem statement

Due to the lack of trust in cloud-based storage services, data confidentiality must be protected or well-preserved before outsourcing sensitive data to the cloud. The current state of support in NoSQL databases further substantiates this view and leads to the conclusion that the application-level data protection (e.g., encryption) is required. This approach allows (i) addressing the trust-related issues between the client and the database, (ii) achieving the desired level of flexibility in terms of supporting encryption at the finest level of granularity (attributes), and (iii) offering support for executing different types of search queries and performing complex computation over encrypted data.

However, it also poses additional challenges regarding both *implementation complexity* and *management complexity*. We further elaborate on these challenges in the context of the Billing-as-a-Service SaaS application –which enables customers to perform several different operations (i.e. interactive search queries and complex computation) over their securely-managed financial documents– discussed in Section 2.1. To perform these operations efficiently, without the need to decrypt each record in the database, the application will need to integrate and combine different encryption schemes, leveraging their intrinsic characteristics (as discussed in Section 2.3). For example, finding all paid/unpaid invoices that belong to a specific customer requires an equality search, which can be supported by using an implementation of the deterministic encryption scheme (DET). Similarly, computing the sum of all invoices that belong to a specific customer requires executing an aggregate query and can be supported by adopting the homomorphic encryption scheme (HOM). As another example, finding all paid/unpaid invoices that belong to a customer whose name starts with the letter "X" requires full-text search functionality and is not directly supported by any of the data encryption schemes.

To support the above-mentioned queries, different data encryption schemes –that typically require expertise and custom coding– need to be combined and used within the application, which is not always easy to deploy or maintain and tends to be a time-consuming and error-prone task. The implementation complexity is substantially increased especially when such queries (e.g., full-text search) are supported in the application, which are not directly addressed by any of the data encryption schemes. Moreover, combining different data encryption schemes also requires making trade-offs between security, performance, and query capabilities, which is a non-trivial task and increases complexity from the application developer's point of view. In addition, to perform the complex computation over encrypted data, User Defined Functions (UDF) need to be implemented directly in the database engine for each underlying database, which not only demands expertise, but in turn also introduces additional management complexity.

3. CryptDICE: a distributed data protection system

CryptDICE hides the complexity of different data encryption schemes and performs their adaptive selection in order to provide data protection support, yet enables the execution of (search and aggregate) queries over encrypted data. This section provides an in-depth overview of the design objectives and the architecture of our proposed system. At its core, the system is designed with several objectives in mind:

- Support data protection guarantees at different levels of granularity (e.g., object-level encryption, field-level encryption, etc.).
- Transparent from the developers/end-users perspective and refrain them from engaging in the complexity of different data security mechanisms.
- Perform an adaptive selection of different (encryption) schemes to ensure data protection and also to enable execution of (search and aggregate) queries.
- Avoid transmission of unencrypted data over public communication channels.
- The cost of enabling data encryption support and executing interactive search queries should be minimal and must be achieved without making significant changes in the application code in order to make the system viable for a wide range of applications.
- Perform complex computation (i.e. aggregate queries) on the encrypted data as close as possible to the underlying database engine in order to avoid expensive data shuffling and to obtain optimal performance.
- Do not introduce any modification in the underlying database engine in order to make the system compatible with a variety of different NoSQL databases.

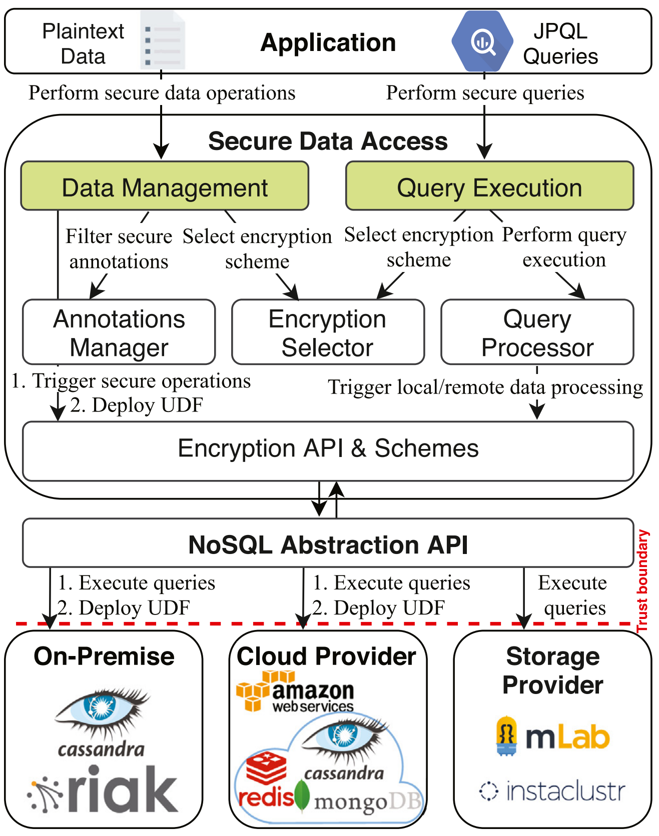

Fig. 2. Architecture of our proposed CryptDICE system.

These objectives led us to design CryptDICE, a distributed data protection system that ensures data confidentiality and enables the execution of interactive search queries for different NoSQL databases with no modification in the underlying database engine and minimum required changes (in the form of annotations) on the client-side applications. As such, it allows service providers to seamlessly outsource their sensitive data to the third-party cloud storage services while ensuring data confidentiality and yet enabling them to operate on their outsourced data. Fig. 2 depicts the organization of major components in our proposed system.

CryptDICE is mainly composed of three layers (from top to bottom): the *Application* layer, the *Secure Data Access* layer, and the *NoSQL Abstraction API* layer. The system can be used on top of different cloud deployment models: on-premise, cloud provider, and cloud storage provider (aka Database-as-a-Service deployment). The overall deployment strategy consists of a trusted zone in which all three layers of the CryptDICE system are deployed along with a private encryption key and an untrusted zone, which is composed of different cloud deployment models. The private encryption key is used to encrypt/decrypt data transmitted between the application and an untrusted zone. This deployment strategy ensures that data is always transmitted in an encrypted form over public communication channels. The remainder of this section presents all three layers of the proposed system and also further discusses different deployment models. In particular, we shed more detailed light on different subsystems and components of CryptDICE.

3.1. Application layer

The *Application* layer communicates directly with the CryptDICE system (i.e. the *Secure Data Access* layer) and performs different operations. For example, the application provides data that need protection support and also issues different queries to the CryptDICE system. The data protection in the system involves three phases: (i) the application uses built-in annotations of the CryptDICE system and performs different operations on plaintext[2] data using the Java Persistence API (JPA) specification, (ii) the data protection support is enabled by the CryptDICE system, which accordingly selects appropriate data encryption schemes for satisfying different data protection requirements, (iii) a pipe-and-filter approach is taken in which the transformation of data is applied before actual storing it and the database itself is fully agnostic of this. Hence, data protection in CryptDICE is enabled by an unmodified NoSQL database engine. Similar to the approach used for data protection, the query execution in the system also involves three phases. First, the application issues queries based on Java Persistence Query Language (JPQL)[3] Backus-Naur Form (BNF)[4] grammar. Second, the query parsing, encryption, decryption, and rewriting processes are done by the designed components of the CryptDICE system. Last, server-side query execution and complex computation over encrypted data is performed by an unmodified NoSQL database engine.

3.2. Secure data access layer

The *Secure Data Access* layer consists of two core subsystems (as shown in green color): the *Data Management* system and the *Query Execution* system. As implied by the names, the former is responsible to perform encrypted data management operations (e.g., encrypted CRUD operations) in our proposed system, while the latter is responsible to generate and coordinate as well as to initiate the execution of search and aggregate queries over encrypted data. The rest of this section covers each of these subsystems in more detail. However, to provide a brief overview of the roles and responsibilities of different components visible in Fig. 2, both these systems communicate with the `Encryption Selector` component, which selects the appropriate data encryption schemes based on the specific data storage and search requirements. The `Annotations Manager` component filters different types of annotations at run-time specified on an entity object (e.g., JPA-specific annotations and built-in annotations of the CryptDICE system) and stores the meta-data in the external cache of the CryptDICE system (not depicted). The `Query Processor` component is mainly responsible to perform different types of queries (e.g., equality-search queries, full-text search queries) and also manipulates local and remote processing of aggregate queries.

The `Encryption API & Schemes` component supports two key functions with respect to data protection and search. First, it provides a uniform API to perform encryption and decryption operations across various NoSQL databases. To realize this support, the component implements the standard interfaces of the *NoSQL Abstraction API* layer and overrides all the methods with the practical implementation of data protection principles. Second, the component encapsulates the complexity and supports the plugin-based implementation of different third-party data encryption schemes (e.g., random, deterministic, homomorphic, etc.) discussed in Section 2.3. As such, based on the scheme selected by the `Encryption Selector` component, the `Encryption API & Schemes` component uses the right implementation of the encryption scheme to encrypt and decrypt data as well as to

[2] In this rest of this paper, whenever we mention plaintext, we actually refer to the object, which could also be the unstructured data and not necessarily only the structured data.

[3] https://docs.oracle.com/javaee/7/tutorial/persistence-querylanguage.htm.

[4] http://itdoc.hitachi.co.jp/manuals/3020/30203Y0710e/EY070508.HTM.

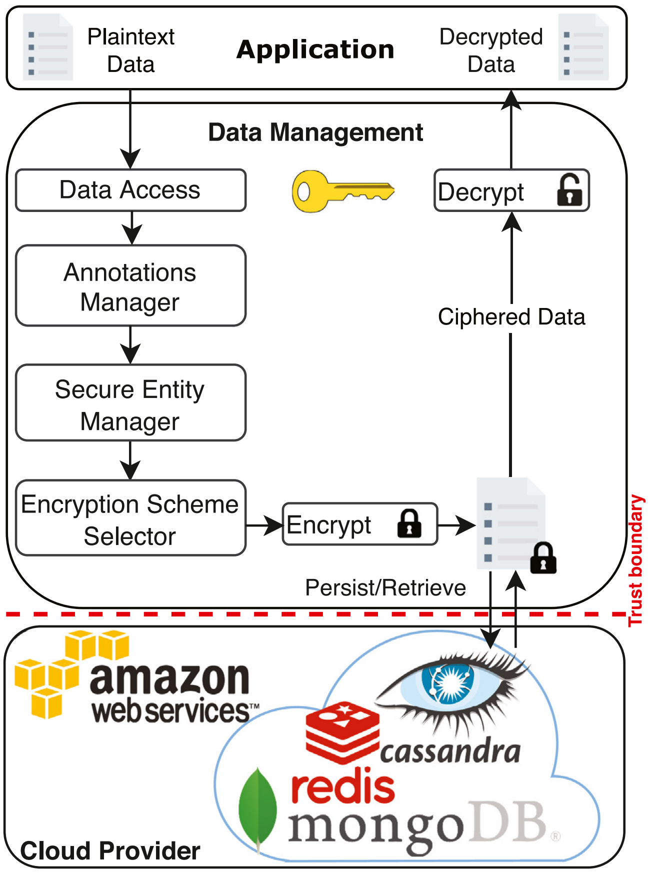

Fig. 3. Detailed overview of different components of the *Data Management* system.

perform data processing. The latter component, in turn, consists of multiple subcomponents (not depicted), which are used by both subsystems (the *Data Management* system and the *Query Execution* system) and onward remain center of focus. In the rest of this section, different components of both subsystems are described concisely.

3.2.1. Data management

A more detailed view on the *Data Management* system, which follows the pipe-and-filter architecture [40] to offer data protection support is depicted in Fig. 3. As shown, the system is comprised of a number of components, each performing different tasks: the `Data Access` component, the `Annotations Manager` component, the `Secure Entity Manager` component, the `Encryption Scheme Selector` component, the `Encrypt` component, and the `Decrypt` component. As shown in Fig. 3, the application communicates directly with the *Data Management* system and performs several operations (e.g., secure data storage operations) on the plaintext data.

The component responsible to act on behalf of the *Data Management* system is the `Data Access` component. This component provides an abstraction API (e.g., JPA interface) for the applications to interact with the *Data Management* system and perform encrypted create, read, update, delete (CRUD) operations and as such also hides the underneath complexity of implementing different cryptographic primitives from the application developers. As the proposed system provides JPA as a standard interface, it supports built-in annotations defined in JPA and also implements additional annotations, which represent metadata. The `Data Access` component communicates with the `Annotations Manager` component, which accesses different annotations of a class, method, and field at runtime. The latter component filters all the annotations to determine different security mechanisms required to provide data protection support.

As the system supports encryption at different levels of granularity, it is important to determine the level at which data protection is required. For example, if object-level data protection is required, data (as a whole object) is encrypted in a single operation. This approach provides higher storage performance with limited query capabilities as a full database scan is required to process different queries, which takes considerable amounts of time. On the other hand, if field-level data protection is required, each field of an object is encrypted individually. This approach provides rich query capabilities, however, at the cost of increased storage performance. To ensure the selection of appropriate data protection level, the `Annotations Manager` component reads all annotations at runtime using Reflection[5] on a per request level. This impacts the performance drastically and therefore, special precautions are taken to minimize the adverse impact on performance (e.g., caching). As a result, the `Annotations Manager` component only filters the annotations once, on the first request, and stores the metadata in the external cache of the CryptDICE system. This means when the latter requests are found in the cache, i.e., there is a cache hit, the requests are served directly from the cache.

The request is then directed to the `Secure Entity Manager` component (a subcomponent of the `Encryption API & Schemes` component), which implements the `EntityManager`[6] interface of JPA and overrides all the methods that deal with CRUD operations. This enables all CRUD methods of the `EntityManager` interface to be implemented with security features in mind for potential data compromise, hence ensuring the required level of data protection. This component also enables existing JPA-based applications to use the proposed system, which has built-in data protection features for a wide range of different NoSQL databases without making any modifications in the application code. To address different requirements for data protection, the `Secure Entity Manager` component uses the `Encryption Scheme Selector` component, which selects the appropriate data encryption schemes[7] for given data storage and search requirements. For example, if object-level protection is required, the component selects the data encryption scheme (e.g., RND) different than the one used for field-level protection (e.g., DET). Similarly, if a specific field/member of an entity also requires complex computation to be performed on, the component selects the data encryption scheme (e.g., HOM) different than when the full-text search queries are required.

The `Secure Entity Manager` component then interacts with the `Encrypt` component (a subcomponent of the `Encryption API & Schemes` component) and requests to apply the appropriate encryption operations according to the selected encryption schemes. This is the key component of the *Data Management* system, which not only applies encryption operations but also performs additional functions. For example, if a specific field of an entity also requires full-text search, which is not supported by any of the existing data encryption schemes, the component builds encrypted custom indexes and stores them in the configurable index database of CryptDICE (e.g., Apache Lucene, Elasticsearch) to facilitate full-text search queries (cf. the second column of Table 2 to follow a number of steps that CryptDICE takes to support different types of queries). The output of the

[5] https://www.oracle.com/technical-resources/articles/java/javareflection.html.

[6] https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html.

[7] Table 2 presents an overview of different data encryption schemes.

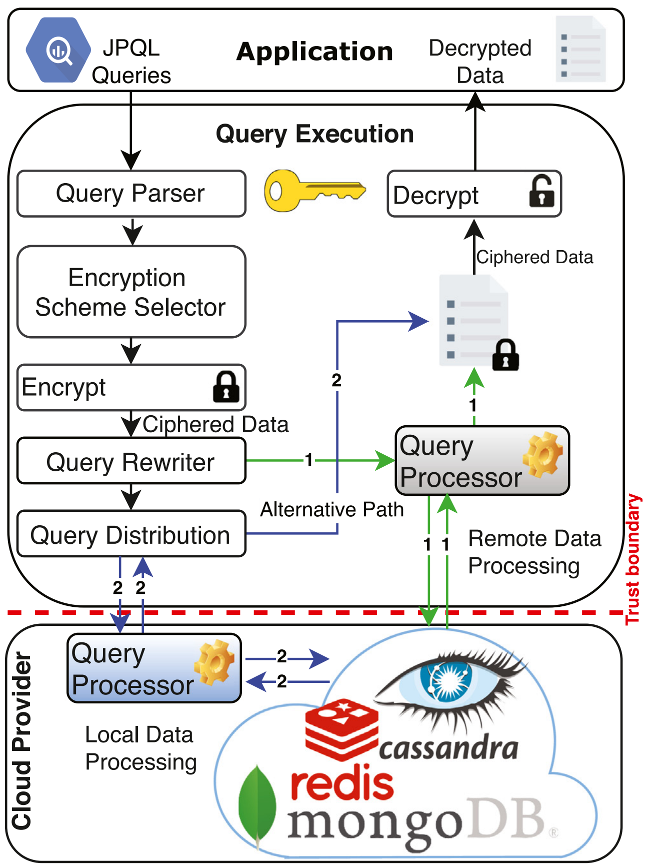

Fig. 4. Detailed overview of different components involved in the *Query Execution* system. CryptDICE supports two alternative strategies for the deployment of the `Query Processor` component. The component is (i) deployed in an on-premise environment () and thus complex aggregate queries are processed remotely (remote to the database engine), and (ii) migrated from an on-premise environment to the cloud provider and thus complex aggregate queries are processed locally (local to the database engine).

`Encrypt` component is the ciphered data, which is stored in external third-party cloud providers. The counterpart of the `Encrypt` component, the `Decrypt` component (a subcomponent of the `Encryption API & Schemes` component) is used by the *Data Management* system to reverse the operation, by converting ciphered data into plaintext data (e.g., an entity object). The `Decrypt` component selects the appropriate decryption operations on each field of an entity according to the applied data encryption schemes. The output of this component (i.e. decrypted plaintext data) is then sent back to the application.

3.2.2. Query execution

A more detailed view on the *Query Execution* system is presented in Fig. 4. Similar to the deployment setup of the *Data Management* system, the deployment setup of *Query Execution* consists of a trusted zone and an untrusted zone. The core of the *Query Execution* system consists of six components: the `Query Parser` component, the `Encryption Scheme Selector` component, the `Encrypt` and `Decrypt` components, the `Query Rewriter` component, and the `Query Processor` component. As shown, the application communicates with the *Query Execution* system and issues different (non-encrypted) JPQL queries.

The `Query Parser` component acts as a proxy respondent on behalf of the *Query Execution* system. The component consumes JPQL queries submitted to the *Query Execution* system, analyzes every single query, and emits the results. More specifically, the `Query Parser` component processes queries in two steps. The first step transforms a query from a raw string of characters into an abstract syntax tree (AST) representation. The second step scans the AST and finds all the column identifiers and their respective values as well as filters different clauses of the query. The extraction of clauses helps the `Query Parser` component to determine the feasibility of executing different queries. To confirm that, the component checks if column identifiers are encrypted using the appropriate data encryption schemes. For example, if a full-text search query is performed, the `Query Parser` component checks if the column identifiers and their corresponding values are encrypted and stored in the database using the data encryption scheme that supports full-text search.

After that, the component interacts with `Encryption Scheme Selector` and `Encrypt` components. The roles and responsibilities of both of these components have been discussed in detail in Section 3.2.1. However, to briefly describe their functions, the `Encryption Scheme Selector` component selects the appropriate data encryption schemes, whereas the `Encrypt` component (a subcomponent of the `Encryption API & Schemes` component) performs various encryption operations according to the selected data encryption schemes. In the specific case of query execution, the `Encryption Scheme Selector` component selects data encryption schemes for each corresponding value of the column identifiers. The `Encrypt` component, on the other hand, performs the actual encryption operations on the values of the column identifiers based on the selected data encryption schemes. The output of this component is the ciphered data, which is then passed to the `Query Rewriter` component (a subcomponent of the `Encryption API & Schemes` component). The latter component rewrites the query, and as such replaces the non-encrypted parts of the JPQL query with the encrypted parts. This component ensures that the modified JPQL query is secure in the sense that it does not reveal any confidential data, hence, it is safe to be sent over unsecured communication channels. The updated query is then passed to the `Query Processor` component, which is considered to be the heart of the CryptDICE system and also works as the core component of the *Query Execution* system.

Query processor. The `Query Processor` component executes different types of search and aggregate queries (e.g., equality search queries, full-text search queries, complex computation, etc.) over encrypted data. The component (locally/remotely) connects to the database engine depending upon its deployment strategy and executes the query. Most of these queries (i.e. non-aggregate queries) are executed when the component is deployed in an on-premise environment (i.e. trusted zone). This involves remotely connecting to the database engine deployed in third-party cloud providers (i.e. untrusted zone), executing different types of non-aggregate queries over encrypted data, and returning all the ciphered data from the untrusted zone to the trusted zone. As depicted in Fig. 4, the component receives ciphered data, which is then passed to the `Decrypt` component of the *Query Execution* system. The returned ciphered data (column identifiers and their corresponding values of each row) is then decrypted according to the applied data encryption schemes. The output (i.e. decrypted plaintext data) of the latter component is then sent back to the application.

In the case of performing a complex computation over encrypted data (e.g., SUM, MAX, MIN, AVERAGE), there are two alternative ways: *database-side* computation and *client-side* computation. For the database-side computation, User Defined Functions (UDF) needs to be implemented inside the database engine for each underlying database technology, which introduces several key challenges[8] regarding database-specific performance

[8] We discuss each of these challenges in detail in the comparative analysis section (Section 5.3.3).

optimality, limited applicability, and increased maintainability of code. The other way is to perform computation at the client-side, inside the CryptDICE system. In this context, the `Query Processor` component of the CryptDICE system provides support for two alternative deployment strategies: remote data processing and local data processing.

Remote data processing. In the first deployment strategy (1 in Fig. 4), the `Query Processor` component is deployed along with all other components of the *Query Execution* system in the trusted zone (client-side). Similar to executing different types of queries discussed above, this strategy also requires remotely connecting to the database engine. However, instead of executing queries inside the database, this strategy requires (i) retrieving all encrypted data from the remote database (deployed in an external cloud provider) to the `Query Processor` component (deployed in an on-premise environment), (ii) performing complex computation over encrypted data (e.g., computing sum or average) inside the *Query Execution* system (a subsystem of CryptDICE), and (iii) returning the computed result (e.g., total sum or average) to the `Decrypt` component. The computed result is then decrypted using the homomorphic (HOM) data encryption scheme and sent back to the application. This deployment strategy is less adequate and certainly more costly in terms of performance as it involves remote data shuffling from a cloud server to a private on-premise network. The performance issue of data shuffling is one of the areas of major concern that is particularly relevant for applications that deal with large data volumes.

Local data processing. To minimize the amount of data shuffling that occurs between the cloud provider and an on-premise network, the `Query Processor` component is designed as a lightweight service, which also provides migration transparency. As such, the component can be migrated from one location (on-premise environment) to another (cloud provider) without the clients being notified about the relocation and the application being aware of the location of the component. In addition, the `Query Processor` component is designed with compatibility in mind in order to make it work with different NoSQL databases. Therefore, this component is also built upon an abstraction API for NoSQL databases and requires no modification in the underlying database engine. In the second strategy (2 in Fig. 4), the `Query Processor` component is migrated from an on-premise environment to the cloud provider (e.g., AWS cloud) in order to avoid expensive data shuffling and execute low-latency aggregate queries. As shown, the `Query Distribution` component interacts with the `Query Processor` component, which is deployed in a public cloud provider to perform computation over encrypted data. The latter component completes local data processing, performs computation next to the database engine, and returns only the computed result (e.g., total sum or average) back to the `Query Distribution` component. The computed result is decrypted using the homomorphic (HOM) data encryption scheme and sent back to the application. In comparison to the former strategy, this deployment strategy achieves low-latency aggregate queries, hence results in performance enhancement.

3.3. NoSQL abstraction API layer

The proposed architecture aims to be generic, in order to be compatible with most of the existing NoSQL databases without requiring any modifications in the underlying database engine. For this reason, and as also depicted in Fig. 2, we use the database-agnostic *NoSQL Abstraction API* layer (e.g., Impetus Kundera, Hibernate OGM, DataNucleus, EclipseLink etc.), which is commonly referred to as Object-NoSQL Database Mapper (ONDM) framework. The layer addresses the heterogeneity problem (in terms of different APIs) by providing a uniform API to applications for using different NoSQL databases. The `NoSQL Abstraction API` layer communicates with the underlying database engine, deployed in an untrusted environment, and performs different operations. Besides, data remains encrypted when it leaves the trusted zone and always stored in the underlying database engine in an encrypted fashion.

3.4. Deployment models

As depicted in Fig. 2, the CryptDICE system can be used on top of three different deployment models: on-premise deployment, cloud provider, and cloud storage provider (aka Database-as-a-Service deployment). These deployment models are differentiated depending on who owns and manages them and can be used without making any changes in the application code and as such by just configuring the `persistence.xml` configuration file of the CryptDICE system. First, CryptDICE can be deployed on top of an on-premise deployment model in which databases are deployed and managed locally in an on-premise environment, but the database administrators cannot necessarily be trustworthy enough. Second, the system can be deployed and used on top of a cloud provider in which databases are managed by third-party cloud providers (e.g., Amazon AWS). In both of these deployment models, most of the requests from the CryptDICE system involve performing CRUD operations, executing different types of queries, and migrating the lightweight service of CryptDICE –that performs dynamic deployment of User Defined Functions (UDF)– next to the database engine in order to achieve low-latency aggregate queries. Third, the system can also be deployed and used on top of a cloud storage provider. CryptDICE can be used on top of this deployment model to perform CRUD operations and execute different types of queries over encrypted data. As the database is managed as-a-Service by a third-party cloud storage provider, this deployment model does not offer enough flexibility to perform dynamic deployment of UDF next to the database engine.

4. Prototype implementation

A proof-of-concept implementation of CryptDICE is developed and made available to the community.[9] We choose to implement the prototype of CryptDICE on top of Impetus Kundera, an open-source abstraction layer (aka Object-NoSQL datastore mapper (ONDM) framework) so we could avoid dealing with heterogeneity in terms of different APIs to communicate with several NoSQL databases and thus reduce the implementation complexity. Besides, Impetus Kundera also introduces the least performance overhead, compared to the state-of-practice ONDM frameworks [41]. Hence, we are able to evaluate its competence as a data protection system as well as the compatibility with state-of-the-art and -practice systems and ONDM frameworks.

While the design of CryptDICE is decoupled from any particular database system, the application case (cf. Section 2.1 for more information about the Billing-as-a-Service application case) we have built upon CryptDICE is configured to use the popular column-oriented database such as Apache Cassandra and the document-based database such as MongoDB. However, it supports a wide range of other databases: *in-memory databases* such as Redis, *full-text search engines* such as Elasticsearch, *big data processing systems* such as Apache Spark, *relational databases* such as MySQL, and *NoSQL databases* such as Oracle NoSQL, Apache HBase, CouchDB, etc. In addition to the wide range of databases supported, the prototype also makes use of technologies such

[9] The prototype implementation is freely available at: http://people.cs.kuleuven.be/ansar.rafique/CryptDICE.zip.

as Apache Lucence and Elasticsearch for indexing purposes in order to facilitate full-text search queries over encrypted data. As full-text search is not supported by any of the data encryption schemes, we have built custom encrypted indexes and employed a key-value datastore such as Redis in a semi-persistent manner to take advantage of basic constructions such as persistent sets and maps. Although the prototype –which supports data protection and enables search and computation over encrypted data– was implemented on top of specific database versions, it works with older and newer versions without requiring any modifications to applications or the databases.

In order to communicate with various back-end databases in a uniform way, the prototype is integrated with standardized middleware based Java Persistence API (JPA) and Java Persistence Query Language (JPQL) and we augment this with a layer of data protection and different data encryption schemes. Consequently, it relies extensively on built-in annotations provided by the JPA and also defines a number of custom annotations, which are used to perform data storage, search, and computation operations. CryptDICE is transparent to the client application and transforms JPA queries to secure JPQL queries internally. In Table 3, we give an overview of custom annotations specific to CryptDICE that are currently supported and for each annotation, we also describe a sequence of internal actions performed by CryptDICE with a summary of their advantages and disadvantages.

```
db.invoices.mapReduce(
function() {
emit(this.name, this.amount);
}, function(key, values) {
var total = values[0];
for (var i = 1; i < values.length; i++){
total = he_add(total, values[i]);
}
return total;
}
);
```

Listing 1: An example MongoDB MapReduce query using *he_add* function which implements Paillier HOM addition.

CryptDICE supports two modes of execution: local and remote, is implemented in Java 8, and consists of over 13K lines of Java code. Concerning the necessary cryptographic primitives (e.g., AES), we have used the implementation of the Java Cryptography Extension (JCE). We further leveraged jPaillier [42], a Java implementation of the Paillier [43] homomorphic cryptosystem to perform computation in CryptDICE.

To perform a thorough comparative analysis of different possible ways of performing computation over encrypted data, we have also implemented the homomorphic addition of the Paillier cryptosystem as User Defined Functions (UDF) directly in the underlying database engine. In the current implementation, UDF are employed for two popular NoSQL databases: Cassandra and MongoDB. The high-level Hom addition function, as shown below, entails multiplication of the encrypted values m_1 and m_2. For brevity, keys and random values are omitted.

$$\mathsf{Enc}(m_1) \cdot \mathsf{Enc}(m_2) \bmod n^2$$

The implementation is based on Javallier [44], a Java library for the Paillier [43] homomorphic scheme. These UDF can be deployed in MongoDB and Cassandra through both the middleware automatically and the command-line interface manually. In MongoDB, UDF must be implemented in JavaScript since the multiplication of large integer values requires special types such as `BigInteger` and MongoDB does not support such types for UDF. Hence, we employed a custom implementation of this missing type [45]. The function *he_add* is added in the *system* database via a user with privileged access rights. To incorporate these UDF in queries, we employed Map-Reduce[10] functionality. The `map` function in MongoDB applies to each input document and emits key-value pairs. For keys with multiple values, MongoDB applies the `reduce` function, which collects and condenses the aggregated data. Listing 1 illustrates an example query that sums up the amounts of invoices per person in the Billing-as-a-Service SaaS application case discussed in Section 2.1.

In the case of Cassandra, we implemented the Hom addition as a user-defined aggregate function (UDA). A UDA is typically composed of two functions: a `state` function to compute the multiplications on each row and update the query state with the results for the next row, and a `final` function to perform some actions at the end such as division for calculation of averages. A typical UDA is applied to data stored in a table as part of query results. To illustrate, the following query goes through all of the invoices in the Billing-as-a-Service use case and calculates the sum of all amounts.

```
SELECT homsum(invoice_amount, invoice_nsquared)
FROM invoices
```

Cassandra provides the developers with a wider spectrum of programming languages. We developed the function in Java and leveraged its native `BigInteger` type.

5. Evaluation

This section describes the techniques and choices made to evaluate the efficiency and effectiveness of CryptDICE as well as to analyze its impact on the overall performance of the application. Section 5.1 describes the application setups and discusses the different deployment setups in which we tested CryptDICE along with details on software and hardware used for the evaluation. Then, our research focuses on a series of experiments, which are conducted to evaluate CryptDICE in three different dimensions.

More precisely, first, Section 5.2 examines the development effort required to enable data encryption support and also to execute interactive search queries in the application. Then, Section 5.3 focuses on the performance analysis of implementing database functions in order to perform complex computation over encrypted data. Final, Section 5.4 evaluates the performance impact, more specifically the performance overhead introduced by CryptDICE. As the financial sector deals with different types of sensitive information and needs to comply with several rules and regulations concerning data security and privacy, we selected the Billing-as-a-Service SaaS application discussed in Section 2.1. Therefore, all these evaluations are conducted in the context of our implementation of the billing SaaS application.

5.1. Application setup

In order to evaluate different aspects of our proposed system, we have implemented two application prototypes, which are based on the Java Persistence API (JPA) standard and perform the same set of operations. These prototypes are implemented to provide secure data storage services (i.e. encrypted CRUD operations) and to execute interactive search queries as well as to perform complex computation over encrypted data. To validate our approach, in all experiments, we compare *CryptDICE^{+ES}*, an application prototype of the Billing-as-a-Service SaaS application built on top of CryptDICE and has a built-in support for different data encryption schemes with *Baseline*, an application

[10] https://docs.mongodb.com/manual/core/map-reduce/.

Table 3
An overview of custom annotations, which are currently supported in CryptDICE.

Annotation	CryptDICE	Advantages	Disadvantages
@Entity({ @MetaInfo(key = "data", value = "Confidential"), @MetaInfo(key = "type", value = "Customer")})	1. Encrypt the full entity using AES or any other configurable encryption algorithm. 2. Store the encrypted entity in the database.	A one-time encryption operation, hence efficient for write performance.	Expensive search operations as a full database scan is required.
@Entity({ @MetaInfo(key = "data", value = "Confidential"), @MetaInfo(key = "type", value = "Customer")}) @Confidential(members = {"firstName", "lastName"})	1. Encrypt each specified member of an entity individually using AES or any other configurable encryption algorithm. 2. Store each encrypted member of an entity in the database.	Efficient search operations as a full database scan is not required.	1. Multiple encryption operations are required to encrypt each member of an entity individually. 2. Inefficient for write performance.
@Entity({ @MetaInfo(key = "data", value = "Confidential"), @MetaInfo(key = "type", value = "Customer")}) @FullTextSearch(members = {"address"})	1. Tokenize each specified member of an entity. 2. Encrypt each token individually. 3. Store each encrypted token in the index database (e.g., Elasticsearch). 4. Encrypt each specified member of an entity individually using AES or any other configurable encryption algorithm. 5. Store each encrypted member of an entity in the primary database (e.g., Cassandra/MongoDB).	Enables to perform a full-text search over the encrypted data.	1. Multiple encryption operations are required to encrypt each member of an entity individually. 2. Multiple encryption operations are required to encrypt multiple tokens of an individual member. 3. Inefficient for write performance.
@Entity({ @MetaInfo(key = "data", value = "Confidential"), @MetaInfo(key = "type", value = "Document")}) @Aggregate(members = { "amount", function = Function.ALL})	1. Encrypt each specified member of an entity individually using Paillier asymmetric algorithm. 2. Store each encrypted member of an entity in the database.	Enables to perform aggregate functions (e.g., SUM, MAX, MIN, AVG) over the encrypted data.	1. Database space is increased significantly. 2. It is computationally intensive and also expensive.

Table 4
Table schema for storing personal information of customers of the Billing-as-a-Service SaaS application.

Key	Identification			Contacts		Tenant
	CustomerNo	Surname	Name	Address	Contact	TenantID
	@EqualitySearch			@FullTextSearch		@EqualitySearch
	DET			CUSTOM		DET

prototype of the Billing-as-a-Service SaaS application built on top of Impetus Kundera (i.e. data access middleware platform) albeit without any inherent support for data encryption. Therefore, CryptDICE^{+ES} has built-in capabilities to provide secure data storage services, to execute different types of interactive search queries, and also to perform computation over encrypted data, whereas such capabilities need to be manually implemented for Baseline.

To evaluate our application prototypes in a more realistic setup, two table schemas are designed with a one-to-many relationship between them. This association between the tables leads to a realistic representation of many use cases from several different application domains. The first table schema, `Customer`, is shown in Table 4 that stores personal information from different customers. As shown, for each customer, the table stores an application generated key (CustomerID), while each row is composed of a set of tuples (Identification, Contacts, Tenant) that group distinct column qualifiers holding customer's information (CustomerNo, Surname, Name, Address, Contact, TenantID). Table 4 also shows a number of annotations, which are used to specify different types of search requirements. Based on these requirements, CryptDICE selects the appropriate data encryption schemes to be applied on each column in order to ensure the privacy of customer's personal information, while still enabling interactive search queries on encrypted data. The last row of Table 4 proposes possibles data encryption schemes, which are used by CryptDICE to encrypt different columns of the table. As shown, a CUSTOM data encryption scheme is implemented to encrypt the column qualifiers *Address* and *Contact* in order to support full-text search operations on these columns. The remaining column qualifiers of the table are encrypted with the deterministic (DET) encryption scheme.

The second table schema, `Document`, is shown in Table 5 that stores various types of financial documents (e.g., invoices, bills) for different customers of the application. The key (DocumentID) is a unique identifier generated by the application and each row is composed of a set of tuples (Document Information, Customer) grouping distinct columns holding relevant information for every document of the customer (DocumentNo, Type, Status, Date, Amount, CustomerID). The last row of Table 5 shows the possible data encryption schemes, which are used by CryptDICE to encrypt different columns of the table schema. For example, the column qualifier *Amount* is encrypted with the homomorphic (HOM) encryption scheme in order to perform complex computation (e.g., SUM, MAX, MIN, AVERAGE) over the encrypted amount, while the remaining columns of the table are encrypted with the deterministic (DET) encryption scheme.

5.2. Cost of enabling data encryption support

In this part of the evaluation, we examine the development cost in terms of lines of code added to enable data encryption support and also quantify the estimated development effort needed to implement different data encryption schemes in order to support various types of queries. To enable data encryption

Table 5
Table schema for storing documents belonging to different customers of the Billing-as-a-Service SaaS application.

Key	Document information					Customer
	DocumentNo	Type	Status	Date	Amount	CustomerID
	@EqualitySearch				@Aggregate	@EqualitySearch
	DET				HOM	DET

support, we consider encryption at different levels of granularity for both application prototypes discussed in Section 5.1. For example, encryption is supported at the level of individual object (which is represented as $\text{Encrypt}^{\text{Obj}}$) and also at the level of a specific field (which is represented as $\text{Encrypt}^{\text{F}}$). From a database perspective, an object represents a row of the table, whereas a field corresponds to a column of the table. Similarly, for both application prototypes, different data encryption schemes are considered to process various types of queries. The results of this aspect of the evaluation are presented in Table 6.

As shown in the third row of Table 6, to provide data encryption support, the Baseline prototype (that has no built-in provisions for data encryption) needs to implement 123 lines of code to achieve object-level encryption (i.e. $\text{Encrypt}^{\text{Obj}}$) and 192 lines of code to perform field-level encryption (i.e. $\text{Encrypt}^{\text{F}}$). In the case of object-level encryption, it is a one-time encryption process in which the whole object is encrypted, whereas each field of an object needs to be encrypted separately for the field-level encryption. Therefore, more lines of code are required to be implemented for encryption at the level of specific fields as compared to the object-level encryption. To support different types of interactive search queries, the number of lines of code increases substantially for Baseline. For example, 921 lines of code are implemented to support equality search queries. This includes the cost of enabling data encryption support on specific fields of an object which require search operations (i.e. 192 lines of code) and also involves the development cost of supporting equality search queries (i.e. 729 lines of code). Similarly, 1231 lines of code are added to support full-text search queries, which include the cost of implementing equality search queries (i.e. 729 lines of code) and also involve the additional development cost of building and maintaining custom indexes (i.e. 502 lines of code). Finally, to support aggregate queries in the Baseline prototype, 408 lines of code are implemented.

On the other hand, $\text{CryptDICE}^{+\text{ES}}$ (an application prototype of the Billing-as-a-Service SaaS application that runs on top of the CryptDICE system, which has built-in support for different data encryption schemes) only requires the introduction of appropriate annotations in order to enable data encryption support and also to perform interactive search queries. This involves a single line of code in the form of an annotation to be added to the application. As shown in the last row of Table 6, to achieve the object-level encryption (i.e. $\text{Encrypt}^{\text{Obj}}$) in $\text{CryptDICE}^{+\text{ES}}$, the @MetaInfo annotation is used. However, to perform the field-level encryption (i.e. $\text{Encrypt}^{\text{F}}$), the @Confidential annotation is used where each member of an entity, which needs to be encrypted can be specified. Similarly, to support different types of search queries, different security annotations are supported. For example, if the @Confidential annotation is used and each field of an entity is specified, equality search queries can be performed on the specified fields without introducing an additional annotation. To facilitate full-text search and aggregate queries in $\text{CryptDICE}^{+\text{ES}}$, @FullTextSearch and @Aggregate annotations are used respectively.

5.3. Comparative analysis and performance evaluation of implementing database function

This group of experiments evaluates the performance of CryptDICE in terms of its ability to process queries over encrypted data when compared to the query processing mechanism that executes directly from the database. Therefore, in this part of the experimentation, we consider and evaluate three different modes of implementing database functions. First, the database function is implemented as User Defined Functions (UDF) in the *database engine* that executes directly from inside the database daemons. Second, the database function is implemented in the CryptDICE system in which all the data is fetched and processed inside CryptDICE and thus remotely to the database engine (i.e. *remote data processing*). Third, the database function is implemented within a lightweight service of CryptDICE, which performs dynamic deployment of database functions in the cloud and thus allows data to be processed as close as possible to the database engine (i.e. *local data processing*).

We present detailed comparative analysis of these three modes and also report on their performance evaluation with respect to the execution time. Section 5.3.1 presents an overview of the experimental setup, while Section 5.3.2 discusses the results. Finally, in Section 5.3.3, we close with a critical assessment and conduct a comparative analysis of the efficiency of all three different modes of implementing database functions.

5.3.1. Experimental setup

The experiments are performed for the table schemas (discussed in Section 5.1) under different workload conditions. More specifically, we start our measurements with 5000 financial documents of type invoices and increase that number up to 500,000 documents. We first insert the information of a number of customers in the `Customer` table. Then, we insert a number of documents of type invoices in the `Document` table that belong to specific customers. Finally, we perform the aggregate queries on the *Amount* column of the `Document` table, which is encrypted using the homomorphic encryption scheme.

Table 7 shows a number of aggregate queries, which perform complex computation over a different set of encrypted documents (i.e. invoices), depending upon the search conditions. However, for evaluation purposes, we consider the worst-case scenario where the computation is performed over the entire encrypted invoices stored in the database. Therefore, we have considered SUM and AVG aggregate functions for the evaluation as shown in the second and fifth rows of Table 7. We specifically measure the performance in terms of the execution time to process different types of aggregate queries, considering all three modes of implementing database functions. In order to mitigate the effect of randomness and also to increase the reliability of the presented results, each experiment is repeated three times independently and the average values are reported. After the completion of each run, the entire database engine is emptied and repopulated.

The database functions are evaluated in a client-server environment where the client node (running the benchmarking application) interacts with the server node (running the database engine). In our experimental setup, the client node is equipped with Intel(R) Core(TM) i7-865U CPU @1.90 GHz @ 2.60 GHz processors with 16 GB RAM and Windows 8 operating system installed. The server node, which is running the database engine is deployed on two different cloud setups: a private cloud setup and public cloud setup. First, the server node is deployed on

Table 6
An overview of total number of lines of code implemented for both application prototypes to (i) support data encryption at the level of individual object (Encrypt^{Obj}) and also at the level of a specific field (Encrypt^{F}) (ii) enable different data encryption schemes to facilitate various types of interactive queries.

Prototypes	Data encryption support		Types of interactive search queries			Dynamics of modification[a]	
	Encrypt^{Obj}	Encrypt^{F}	Equality search	Full-text search	Aggregate query	Application code	Annotation
Baseline[b]	123	192	921	1231	408	×	
			1				
CryptDICE^{+ES}	@Entity({ @MetaInfo(key = "" value = "")})	@Confidential(members ={})		@FullTextSearch()	@Aggregate()		×

[a]In the Baseline prototype, a number of lines of code are implemented in the application to provide data encryption support and also to offer interactive search queries, whereas only built-in annotations are used for the CryptDICE^{+ES} prototype.

[b]Baseline is developed on top of an existing ONDM framework (i.e. Impetus Kundera), which has no built-in support for data encryption.

Table 7
A number of aggregate queries, which are used to perform computation on the encrypted data.

Aggregate	Description
SUM	Calculate the sum of all invoices
SUM(X)	Calculate the sum of all invoices for a specific customer X
SUM(X, Paid)	Calculate the sum of all paid invoices for a specific customer X
AVG	Calculate the average of all invoices
AVG(X)	Calculate the average of all invoices for a specific customer X
AVG(X, Paid)	Calculate the average of all paid invoices for a specific customer X

a private Infrastructure-as-a-service (IaaS) cloud, which is built using OpenStack and all the experiments are performed and the corresponding results are reported. As the server node is deployed in a private cloud and the latency between the client node and the server node is lower, this may not reflect a realistic environment and therefore may obscure the performance benefits of the lightweight service of CryptDICE. Therefore, the server node is then deployed on the Microsoft Azure public cloud[11] and the experiments are repeated for the public cloud setup and the corresponding results are gathered.

In the case of the private IaaS cloud, the server node has an Intel(R) 4 Core @ 2.60 GHz processor, 8 GB RAM and is hosted on a compute node of OpenStack. The compute node consists of 40 Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60 GHz processor with 120 GB RAM and runs the Linux/Ubuntu operating system. In the case of the Microsoft Azure cloud, we instantiated a Standard D2s v3 virtual machine (2 vcpus, 8 GiB memory). To ensure the robustness and comprehensiveness of our analysis and evaluation, we have considered both Cassandra and MongoDB as the back-end database engines.

5.3.2. Results

The results of all the experiments in which the server node is deployed in a private cloud and the aggregate functions are computed on top of the Cassandra database engine are shown in Fig. 5. The x-axis represents the number of encrypted invoices on which the aggregate function is performed, whereas the y-axis shows the total execution time in seconds to compute the aggregate function for a different number of encrypted invoices.

As shown, the mode in which the database function is implemented as User Defined Functions (UDF) and the computation is performed inside the *database engine* performs better than the other two approaches as it takes less time to compute the aggregate functions.[12] For example, to compute SUM on 500K encrypted invoices, the mode in which the computation is performed inside the database engine takes 26,471 s, the mode in which the computation is performed inside CryptDICE and remotely to the database engine (*remote data processing*) takes 60,681 s, and the mode in which the computation is performed local to the database engine (*local data processing*) takes 53,587 s. In the case of remote data processing, data shuffling is required that leads to higher-latency in performing the aggregate queries. This induces an additional cost in terms of the higher execution time of fetching all the data from a remote database engine into CryptDICE and then performing the computation locally. To avoid expensive data shuffling, the lightweight service of CryptDICE allows data to be processed next to the database engine. Therefore, as shown in Fig. 5 the mode in which the computation is performed local to the database engine (*local data processing*) performs much better than the mode in which the computation is performed remotely to the database engine.

The results of experiments in which the server node is deployed in a public cloud (Microsoft Azure cloud) and the aggregate function is computed on top of the Cassandra database engine are shown in Fig. 6. As shown, the mode in which the database function is implemented as User Defined Functions (UDF) and the computation is performed inside the *database engine* takes more or less the same amount of time to perform the aggregate queries as compared to when the server node is deployed in a private cloud (see Fig. 5). For example, to compute SUM on 500K encrypted invoices, the server node which is deployed in a private cloud takes 26,471 s, whereas the server node which is deployed in a public cloud takes 25,784 s. The reason that different deployment setups have no major impact on the performance of aggregate queries is mainly because the computation is performed inside the database engine. The small increase in latency, which is mostly visible between 5K to 100K invoices is due to extra overhead required to communicate with the public cloud setup.

Similarly, the mode in which the computation is performed inside CryptDICE and thus remotely to the database engine (*remote data processing*) takes more time to process different number of encrypted invoices as compared to when the server node is deployed in a private cloud. The variation is mainly because of an extra latency, which is introduced in case of using a public cloud for data shuffling (from public cloud to on-premise environment). However, the performance improvement is clearly evident in case of a public cloud deployment setup for the mode in which the computation is performed local to the database engine (*local data

[11] https://azure.microsoft.com/en-us/.

[12] This mode of computation also presents several challenges and limitations, which are subsequently analyzed and discussed in Section 5.3.3.

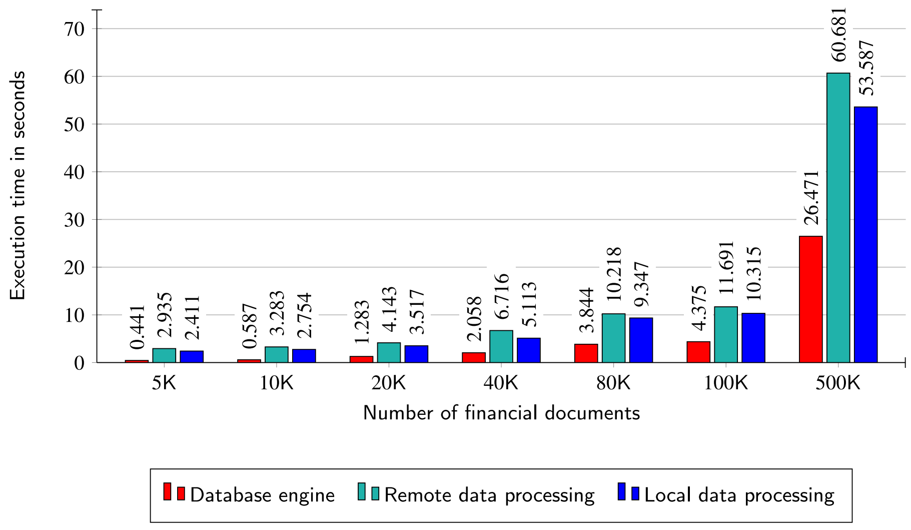

Fig. 5. Total time in seconds required to compute the SUM aggregate function for the data size ranging from 5K up to 500K financial documents using all three modes of implementation the database functions. The server node is deployed in a *private cloud* and the SUM aggregate function is computed on top of the *Cassandra* database engine.

processing). As an example, to compute SUM on 500K encrypted invoices, the server node which is deployed in a private cloud (see Fig. 5) takes 53,587 s, whereas the server node which is deployed in a public cloud (see Fig. 6) takes 49,051 s. In case of a public cloud setup, data shuffling (from public cloud to on-premise environment) is not required as computation is performed inside CryptDICE, next to the database engine.

Beyond these experiments, we also run some additional benchmarks as confirmatory runs where the average is calculated for all the encrypted invoices (run AVG aggregate function). However, we found the results of the AVG aggregate function to be largely consistent with the results of the SUM aggregate function and they lead to the same conclusions as those reported earlier. Therefore, we have omitted the results of the AVG aggregate function from the paper.

5.3.3. Comparative analysis

As is obvious from the results of the previous section, we achieve a significantly better performance when the computation is performed as close as possible to data. That means, aggregate queries, which are executed over homomorphically encrypted data from inside the *database engine* are considered to be better prospects for achieving high performance both on private and public cloud environments. For this reason, the approach has already been used by other prototyped systems. For example, CryptDB [23], the seminal work in this area, implemented the majority of its functionalities such as adjustable encryption and HOM as User Defined Functions (UDF) in the MySQL database. However, our extensive analysis show that the approach is sub-optimal due to concerns regarding database-specific performance optimality, limited applicability, and increased maintainability of code.

Database-specific performance optimality. The performance is database-specific as the approach where the computation is performed from inside the *database engine* does not guarantee the best performance for all NoSQL databases. For example, we have also employed UDF in MongoDB and performed all of the experiments again. The results are presented in Table 8, which show that this mode of computation performs worst in most cases, and even in some cases, it also leads to the "out-of-memory" problem. For instance, to compute SUM on 40K encrypted invoices, this approach takes 502.592 s, which is about ∼250 times slower than when compared to the results of the Cassandra database. Similarly, we encountered the "out-of-memory" problem when we tried to compute SUM on more than 40k (i.e. from 80K up to 500K) encrypted invoices. The reason lies in the fact that values encrypted by homomorphic encryption schemes become considerably larger than their plaintext. Therefore, it is crucial how a database system manages the memory and the states in case of an evergrowing homomorphic addition. That varies from database to database depending on the encryption scheme and the architecture of databases.

On the other hand, the mode in which the computation is performed inside CryptDICE (both for *remote data processing* and *local data processing*), the performance is not so much dependent on the underlying database. To confirm that, we run the same experiments again, but instead of Cassandra, used the MongoDB database. The results, which are presented in Table 8 show that the underlying database has no significant impact on the performance of the mode in which computation is performed inside CryptDICE.

Limited applicability. The implementation of UDF in the *database engine* for cryptographic protocols is not always straightforward. Apart from probable security risks (e.g., side-channel attacks), the set of programming primitives offered by programming languages in databases is sometimes limited for our purpose. For example, MongoDB functions should be written in JavaScript. The Paillier HOM addition involves multiplication of two relatively large integers, and the size of these integers is dependent on the key size. A safe implementation should use big integers to avoid overflows. At the time of implementing *he_add*, JavaScript did not support such primitives. Therefore, we were obliged to use a custom implementation of arbitrary-length big integers. Furthermore, gaining access to cryptographically secure randomness can also be a challenge. In the MongoDB case, employing external libraries is infeasible too, and if external libraries are needed, the source code must be brought to the function implementation. On the other hand, CryptDICE is implemented in Java, which has an extensive set of libraries and supports typical primitives require to implement different cryptographic protocols.

Increased maintainability of code. The introduction of external libraries as function implementation within databases may cause

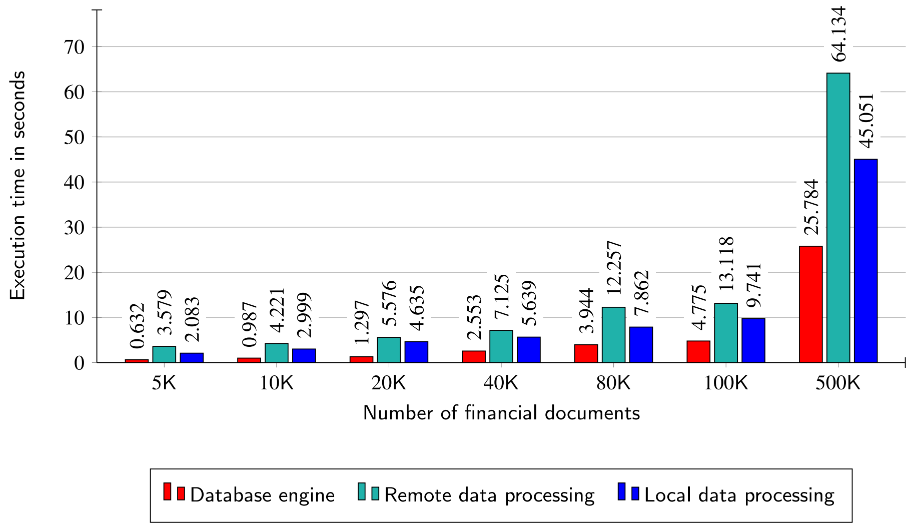

Fig. 6. Total time in seconds required to compute the SUM aggregate function for the data size ranging from 5K up to 500K financial documents using all three modes of implementation the database functions. The server node is deployed in a *public cloud* and the SUM aggregate function is computed on top of the *Cassandra* database engine.

Table 8
Total execution time in seconds to compute the SUM aggregate function using all three modes of implementation the database functions. The server node is deployed in a *private cloud* and the SUM aggregate function is computed on top of both *Cassandra* and *MongoDB* databases. The mode in which the SUM aggregate function is executed over homomorphically encrypted financial documents from inside the *database engine* for the *MongoDB* database leads to the "out-of-memory" problem.

# of documents	**Cassandra**			**MongoDB**		
	Database engine	***Remote data processing***	***Local data processing***	***Database engine***	***Remote data processing***	***Local data processing***
5K	0.441	2.935	2.411	55.238	2.421	2.012
10K	0.587	3.283	2.754	105.994	2.975	2.241
20K	1.283	4.143	3.517	255.207	3.724	2.917
40K	2.058	6.716	5.113	502.592	5.051	4.141
80K	3.844	10.218	9.347	Out of memory error	8.975	7.296
100K	4.375	11.691	10.315	Out of memory error	10.251	8.547
500K	26.471	60.681	53.587	Out of memory error	55.987	48.180

unwanted implementation bugs leading to security vulnerabilities, and on top of that, it may make the debug and update process cumbersome at scale. Besides, monitoring the database functions at run-time through logging is not a trivial task too. Programming language diversity of database functions causes vendor lock-in and hinders implementation portability and re-usability. Each database comes with its requirements, environment and programming language. Therefore, user-defined functions must be developed and maintained per database. For example, MongoDB supports JavaScript; Redis supports Lua; and Cassandra supports several languages including Java.

In a cloud-native setting where service providers want to employ hosted databases, most NoSQL databases do not offer custom functions due to practical and security reasons. An interesting direction for research would be employing Function-as-a-Service (FaaS) paradigm although FaaS do not run within databases. However, exploring FaaS-based approaches is not in the scope of this paper.

5.4. Performance impact

In this section, we assess the impact of CryptDICE on the performance of an application. The experimental results of Sections 5.2 and 5.3 demonstrate the effectiveness of CryptDICE in terms of (i) low development effort required to support data encryption and to execute interactive search queries over encrypted data, and (ii) the ability to perform complex computation as close as possible to data for multiple NoSQL databases in order to achieve optimal performance, respectively. This section sheds light on the other side of the coin by showing that these benefits come with an associated performance overhead. Section 5.4.1 provides details about the experimental setup, while the results are summarized in Section 5.4.2.

5.4.1. Experimental setup

The experiments are performed for application prototypes (the *Baseline* prototype and the *CryptDICE*$^{+ES}$ prototype) of the Billing-as-a-Service SaaS application discussed in Section 5.1. We run our experiments in a typical client-server fashion where the client process runs both application prototypes and communicates directly with the server process, which runs a database engine. More specially, the server process runs the MongoDB database service (version 3.6.4) with its default configuration on a single node, which is deployed over the private cloud infrastructure using OpenStack. In our experimental setup, the client node is equipped with Intel(R) Core(TM) i7-865U CPU @1.90 GHz @ 2.60 GHz processors with 16 GB RAM and Windows 8 operating system installed. The server node has Intel(R) 4 Core @ 2:60 GHz processor, 8 GB RAM, and is hosted on a compute node. The compute node consists of 40 Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60 GHz processor with 120 GB RAM and runs the Linux/Ubuntu operating system.

The experiments –to measure the performance in terms of execution time– were conducted on both application prototypes by executing the CRUD transactions under different workload conditions. We start our measurements with 5000 financial documents of type invoices and increase that number up to 1,000,000 (million) documents. We noticed a significant decrease in response time for the read performance after the first execution, an indication that MongoDB is optimized for read and caches the results of the most recent read requests. To obtain more reliable results and to eliminate the effects of randomization, we repeat each experiment for 3 times and record average results. In addition, after the completion of each run, we emptied the entire database.

5.4.2. Performance results

The results presented in Fig. 7 show the relative performance overhead introduced by CryptDICE for only large data size, involving 500K and 1000K encrypted invoices. As shown, the performance overhead introduced by CryptDICE is higher for the insert operation as compared to read, update, and delete operations. For example, CryptDICE introduces 6,5 % overhead to insert 500K encrypted invoices, whereas the performance overhead decreases to 4,5 % to insert 1000K encrypted invoices. Similarly, the relative performance overhead of CryptDICE is 2,2 % for 500K read operations and 1,1 % for 1000K read operations. The relative performance overhead of CryptDICE is 2,3 % and 1,3 % for 500K and 1000K update operations respectively. For the deleted operation, CryptDICE introduces 2,2 % for 500K encrypted invoices, while the overhead decreases to 1,3 % for 1000K encrypted invoices.

The results show that the overhead introduced by CryptDICE is negligible for read, update, and delete operations. However, we have noticed that CryptDICE incurs considerable overhead for the insert operation. This can be explained as follows: performing an insert operation in CryptDICE requires inspecting and filtering a number of additional annotations (cf. Table 3 for annotation supported in CryptDICE) using reflection at runtime. As CryptDICE supports encryption at different levels of granularity, it first inspects the annotation to decide if the object as a whole to be encrypted or specific members of an object are required to be encrypted. In the latter case, first, CryptDICE persists members in a way that equality-search queries can also be performed on them. Then, it filters the annotation to determine if the full-text search is required in the application. Again, if the full-text search is required, it store members of an object in such a way that full-text search quires can be performed on them (cf. row 4 in Table 3 to examine different steps CryptDICE takes to support full-text search). Finally, CryptDICE filters the annotation to decide if the aggregate queries need to be performed and plans accordingly (cf. the last row of Table 3 to examine different steps CryptDICE takes to support aggregate queries). This all adds up to the additional overhead on the performance of the insert operations.

On the other hand; read, update, and delete operations do not deal with annotations at runtime and therefore they incur low performance overhead. Besides, the experiments conducted in this evaluation perform read operations on the unique object identifier generated by the application (i.e. ID). However, the relative performance overhead can be considerably high if interactive search queries (e.g., equality search on non-identifier columns, full-text search, aggregate queries, etc.) are performed over encrypted data. Similarly, as update and delete operations are also dependent on the read operation, we expect the relative performance overhead for such operations to be higher if read operations are performed on all non-identifier columns. Another interesting conclusion appears from the results that the performance overhead of CryptDICE decreases with the increase in data size. This can be explained as follows: as the absolute performance overhead of CryptDICE is constant and the execution time of the baseline increases gradually with the increase in data size, the relative performance overhead of CryptDICE decreases when the data size increases (e.g., 1000K insert operations).

6. Related work

In the last few years, considerable research has been conducted to mitigate the security challenges in NoSQL databases. This section summarizes related work, which can be broadly classified into two categories: (i) advanced data encryption techniques, and (ii) systems and middleware for protecting sensitive data. In Section 6.1, we give a brief overview of related work on advanced data encryption techniques. Section 6.2 then describes recent research on systems and middleware for protecting sensitive data, with a special emphasis on NoSQL databases.

6.1. Advanced data encryption techniques

Searchable encryption. The research on searchable encryption has been initiated with the seminal work of Song et al. [46]. Following that, the security notions of the symmetric family of such constructions have been defined thoroughly [47–49]. And, over the last decade, these primitives have gone through tremendous improvements in terms of (i) functionalities, e.g. conjunctive [50] and disjunctive [51] queries, (ii) performance, e.g., data locality [52,53], and last but not least (iii) security, e.g., forward and backward privacy [54–56], and more.

In general, these schemes entail three high-level functions: *Token* to generate search tokens based on a secret key, *Query* to search the encrypted index using the generated token, and *Resolve* to decrypt the search outcome. Our system, in particular, the underpinning encryption component, abstracts away these functionalities in a way that the majority of these encrypted search constructions can be plugged in and employed by the query rewriting module.

Homomorphic encryption. Some encryption schemes enable untrusted environments to operate on encrypted data without decrypting the values. These operations typically comprise multiplications and additions. Some schemes only support one of the operations, which are called partially homomorphic encryption, such as El Gamal [20] for an unbounded number of multiplications and Paillier [43] for an unbounded number of additions. In 2009, Gentry proposed [21] the first fully homomorphic encryption (FHE) scheme using lattice-based cryptography, which can perform both operations. There have been tremendous research effort to make FHE more efficient [57–59]. In a high-level description, their execution flows have been investigated, and therefore, our aggregation modules are compliant with their API requirements [60], which means any of the algorithms can get integrated into our system.

Property-preserving encryption. Different searchable encryption constructions have been proposed based on the determinism property of various cryptographic primitives [61–65]. Although determinism introduces equality leakage compared to searchable symmetric encryption, it enables a wide variety of operations for practical systems. A different line of property-preserving schemes aims at encrypted range queries. Order-preserving encryption (OPE) was first introduced by Agarwal et al. [16], and later on, formal security notions and better constructions have been presented by other researchers [66–68]. The security of such schemes has been improved by introduction of the order-revealing encryption (ORE) schemes [17–19]. Although there have been several attacks against the OPE and ORE schemes, efficient encrypted range queries likely to require such primitives. Therefore, the research to achieve more efficient and secure OPE/ORE schemes is an on-going subject of research.

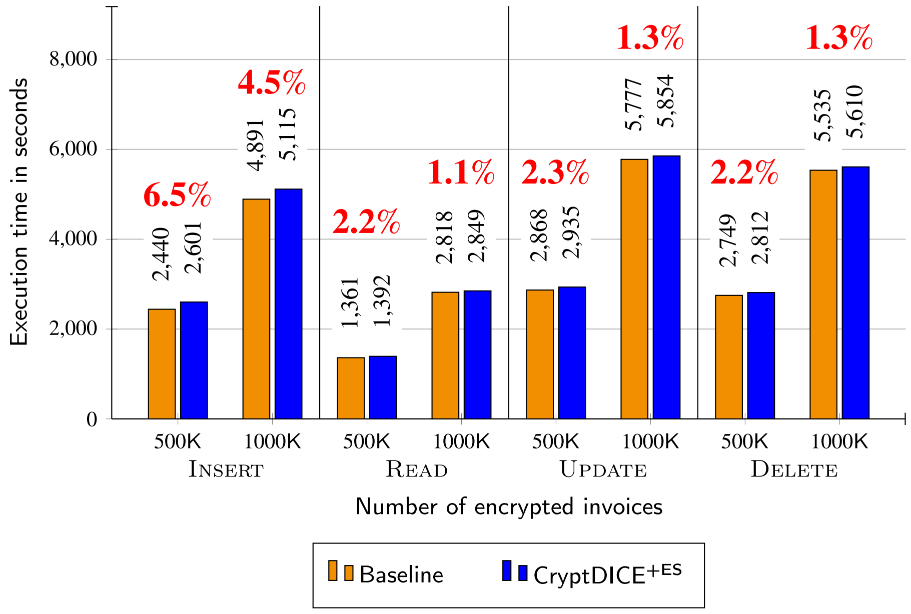

Fig. 7. Total execution time in seconds to perform CRUD operations for 500K and 1000K encrypted invoices. The setup consists of 1 node MongoDB service deployed and managed in a private IaaS cloud using OpenStack. The numbers (%) on top of the bars indicate the performance overhead introduced by CryptDICE for different operations.

6.2. Systems and middleware for protecting sensitive data

Transparent database encryption. The majority of relational databases have built-in encrypted storage engines and support transparent data encryption (TDE), such as Oracle [69], MySQL [70], PostgreSQL [71] and MariaDB [72]. Likewise, most, but not all, enterprise NoSQL database products also offer TDE, such as MongoDB [73], Apache Cassandra [74], Apache HBase [75], Amazon DynamoDB [76]. Almost all of the native TDE support of these databases offers encryption at a coarse-grained key and encryption granularity. For example, they use one or very few keys for the encryption of the entire database. In contrast, our system focuses on client-centric key provisioning and offering encryption at the finest level of granularity, which is field level. The enterprise edition of MongoDB (4.2) recently announced [77] that it supports field-level encryption. Moreover, DynamoDB Encryption SDK client [78] supports field-level data encryption, but it is not natively a part of DynamoDB. Moreover, excluding MongoDB client-side field-level data encryption, which only relies on deterministic data encryption scheme for searching, none of these databases offer searchability and aggregation on encrypted data when encryption keys are managed per client at the application level.

Encrypted database systems. In recent years, several systems (e.g., CryptDB [23], SafeNoSQL [22], Seabed [24], Monomi [25], Opaque [79], EnclaveDB [80], HardIDX [81], Enckv [26]) based on either relational or NoSQL databases have been proposed to query and compute on encrypted data. In this paper, we aim to design a secure data-access middleware with a special focus on enterprise level software development. In that regard, Alves et al. [37] and CloudProtect [82] present frameworks with a set of strictly predefined schemes. SafeNoSQL [22] presents a generic framework with a modular and extensible design that enables data processing over multiple cryptographic techniques applied on the same database schema. DataBlinder [83] presents a distributed and extensible middleware architecture with conceptual abstraction models for searchable encryption. However, this paper aims to go beyond the conceptual models and modularity of the architecture. The key differentiating objective is the reification of these concepts at the middleware and the programming language level through annotations, e.g., within the JPA ecosystem. That has a potential impact on the enterprise software development community.

CryptDB [23] presents a practical encrypted database based on MySQL by offering SQL-aware encryption techniques, adjustable encryption (onion encryption), and chaining encryption keys derived by users' password. The majority of the server-side implementation in CryptDB is done as User Defined Functions (UDF). In addition, CryptDB focuses mainly on relational databases, MySQL in particular, and the query rewriting technique is based on SQL. In contrast, our work aims at NoSQL databases (which are heterogeneous in data models) and uses JPQL as query rewriting abstraction. To perform complex computation, CryptDICE supports both the database-side computation (UDF can be implemented directly inside the database engine) and also the client-side computation (the lightweight service of CryptDICE performs complex computation inside the system and it has support for two modes of execution: remote mode and local mode). Seabed [24] presents big data analytics over encrypted datasets by an additive symmetric homomorphic encryption, which is prototyped on Apache Spark. Monomi [25] employs per-row precomputation, space-efficient encryption, grouped homomorphic addition, and pre-filtering to securely execute analytical workloads over sensitive data. These databases present different techniques to search and perform efficient computation on encrypted data that are tailor-made and strict to their architecture; however, we present a *middleware* rather than a database, which encapsulates different classes of techniques and it is not tied to a particular encrypted search scheme. Opaque [79], EnclaveDB [80], and HardIDX [81] use trusted execution environments, namely Intel SGX, to enable encrypted analytics and searchable encryption by hiding access patterns. Most of these research efforts either modify (or build on top of) the existing databases or offer low-level system security. In contrast, we present a software-only middleware approach to provide a practical encrypted database from a software developer perspective.

Middleware approaches. Prior work by Rafique et al. [84] present PERSIST, a data access middleware which relies on the data mapping strategy proposed in [85,86] to offer scalable and dynamic support for encryption of sensitive data at different levels of granularity such as fields, rows, and tables. Alves et al. [37] present a framework for search and computation on encrypted data, they employ ORE and HOM as their underpinning cryptographic primitives. However, their framework defines and fine-tunes a set of very specific cryptographic primitives and the focus is not primarily on the data access middleware layer. Similarly, CloudProtect [82] is a middleware that transparently encrypts clients' data and relies solely on deterministic encryption for searching. Heydari Beni et al. [83] present DataBlinder, a distributed middleware aiming at crypto agility by abstracting away the cryptographic constructions and presenting three conceptual abstraction models for the leakage profiles, functionalities, and performance for both the security experts and software developers. To do so, the middleware employs an adaptive and reflective architecture such as [87,88] to reify these aspects. SafeNoSQL [22] presents a generic, modular, and extensible architecture enabling data processing using various cryptographic schemes applied on the same database schema for existing NoSQL database engines. However, the primary focus of these architectures is on integration with NoSQL databases and most importantly, on the modularity and extensibility of the cryptographic tactics. CryptDICE, building on top of these future proof architecture contributions, aims to (i) support different categories of NoSQL databases (e.g., columnar, document oriented, etc.), (ii) provide much-needed support for encryption at a different level of granularity (e.g., per table, per object, or per field), and (ii) specify how these abstraction models can be reified in a data access middleware such as Hibernate, a programming language such as Java, and a query language such as JPQL. In addition, none of these research efforts focus on the encrypted full-text search queries, which are not directly supported by the integrated data encryption schemes.

Client-side encryption for Amazon DynamoDB [78] presents an application-level middleware for the encryption of sensitive data with a field-level granularity. However, it does not provide an encrypted search. Azure Always Encrypted database engine [89] provides a protection mechanism for sensitive data not only through encryption but also via hardware enclaves. The engine is capable of deriving the value of initialization vectors (IV) from the content of data, and as a result, enables searchability via the deterministic data encryption scheme.

7. Conclusion

In this paper, we have proposed CryptDICE, a flexible and generic data access system that runs in a distributed fashion and ensures fine-grained protection on application data. The system enables the execution of different types of search and aggregate queries over encrypted data for a wide range of different NoSQL databases with absolutely no modification in the underlying database engine and minimum changes by using the built-in annotations to the client-side applications. The lightweight service of CryptDICE reduces the management complexity of implementing User Defined Functions (UDF) directly in the database engine for each underlying database technology. As such, the service rather implements UDF in the application code and provides migration transparency, i.e., enabling the service to be migrated from an on-premise environment to public clouds and perform complex computations next to the database engine in order to realize low-latency aggregate queries.

We have implemented a prototype of the proposed solution in the context of a realistic industrial Software-as-a-Service (SaaS) application, in order to evaluate different aspects of the CryptDICE system. The evaluation results confirm that CryptDICE significantly reduces the development time and effort for enabling data encryption and supporting different types of interactive secure search queries. Moreover, it offers performance optimizations for achieving low-latency aggregate queries. In addition, the performance overhead induced by CryptDICE is also reported to be modest. The preliminary evaluation we now present focuses primarily on functional aspects, performance criteria, and latency considerations. We will extend the observations of other important dimensions, such as reliability, resilience, and scalability conditions, as future-work.

CRediT authorship contribution statement

Ansar Rafique: Conceptualization, Data Curation, Investigation, Methodology, Software, Validation, Writing - Original Draft Preparation, Writing - Review & Editing. **Dimitri Van Landuyt:** Project Administration, Resources, Supervision, Validation, Visualization, Writing - Review & Editing. **Emad Heydari Beni:** Validation, Writing - Review & Editing. **Bert Lagaisse:** Supervision, Validation, Writing - Review & Editing. **Wouter Joosen:** Funding Acquisition, Supervision, Visualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven and the Cybersecurity Initiative Flanders (CIF) project.

References

[1] I. Arora, A. Gupta, Cloud databases: a paradigm shift in databases, Int. J. Comput. Sci. Issues (IJCSI) 9 (4) (2012) 77.
[2] C. Curino, E.P.C. Jones, R.A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, N. Zeldovich, Relational cloud: A database-as-a-service for the cloud, 2011, https://sungsoo.github.io/articles/cmu-course-papers/CIDR11_Paper33.pdf. [Last visited on September 09, 2020].
[3] O. Ünay, T.İ. Gündem, A survey on querying encrypted XML documents for databases as a service, ACM SIGMOD Rec. 37 (1) (2008) 12–20.
[4] H. Hacigumus, B. Iyer, S. Mehrotra, Providing database as a service, in: Proceedings 18th International Conference on Data Engineering, IEEE, 2002, pp. 29–38.
[5] G. Feuerlicht, J. Pokornỳ, Can relational DBMS scale up to the cloud? in: Information Systems Development, Springer, 2013, pp. 317–328.
[6] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with YCSB, in: Proceedings of the 1st ACM Symposium on Cloud Computing, ACM, 2010, pp. 143–154.
[7] M. Ahmadian, F. Plochan, Z. Roessler, D.C. Marinescu, SecureNoSQL: An approach for secure search of encrypted NoSQL databases in the public cloud, Int. J. Inf. Manage. 37 (2) (2017) 63–74.
[8] J. Silver-Greenberg, M. Goldstein, N. Perlroth, Jpmorgan chase hack affects 76 million households, New York Times 2 (2014).
[9] N.E. Weiss, R.S. Miller, The target and other financial data breaches: Frequently asked questions, in: Congressional Research Service, Prepared for Members and Committees of Congress February, Vol. 4, 2015, p. 2015.
[10] B. Huang, S. Liang, D. Xu, Z. Wan, A homomorphic searching scheme for sensitive data In NoSQL Database, in: 2018 IEEE Smart Data (SmartData), IEEE, 2018, pp. 575–579.
[11] N. Gupta, R. Agrawal, NoSQL security, in: Advances in Computers, Vol. 109, Elsevier, 2018, pp. 101–132.
[12] L. Okman, N. Gal-Oz, Y. Gonen, E. Gudes, J. Abramov, Security issues in NoSQL databases, in: 10th International Conference on Trust, Security and Privacy in Computing and Communications, IEEE, 2011, pp. 541–547.
[13] R. Sellami, B. Defude, Using multiple data stores in the cloud: Challenges and solutions, in: International Conference on Data Management in Cloud, Grid and P2P Systems, Springer, 2013, pp. 87–98.
[14] M.-H. Shih, J. Chang, Design and analysis of high performance crypt-NoSQL, in: 2017 IEEE Conference on Dependable and Secure Computing, IEEE, 2017, pp. 52–59.

[15] X. Tian, B. Huang, M. Wu, A transparent middleware for encrypting data in MongoDB, in: 2014 IEEE Workshop on Electronics, Computer and Applications, IEEE, 2014, pp. 906–909.

[16] R. Agrawal, J. Kiernan, R. Srikant, Y. Xu, Order preserving encryption for numeric data, in: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, 2004, pp. 563–574.

[17] D. Boneh, K. Lewi, M. Raykova, A. Sahai, M. Zhandry, J. Zimmerman, Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation, in: Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2015, pp. 563–594.

[18] D. Cash, F.-H. Liu, A. O'Neill, M. Zhandry, C. Zhang, Parameter-hiding order revealing encryption, in: International Conference on the Theory and Application of Cryptology and Information Security, Springer, 2018, pp. 181–210.

[19] N. Chenette, K. Lewi, S.A. Weis, D.J. Wu, Practical order-revealing encryption with limited leakage, in: International Conference on Fast Software Encryption, Springer, 2016, pp. 474–493.

[20] T. ElGamal, A public key cryptosystem and a signature scheme based on discrete logarithms, IEEE Trans. Inf. Theory 31 (4) (1985) 469–472.

[21] C. Gentry, Fully homomorphic encryption using ideal lattices, in: Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing, 2009, pp. 169–178.

[22] R. Macedo, J. Paulo, R. Pontes, B. Portela, T. Oliveira, M. Matos, R. Oliveira, A practical framework for privacy-preserving NoSQL databases, in: 2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS), IEEE, 2017, pp. 11–20.

[23] R.A. Popa, C.M.S. Redfield, N. Zeldovich, H. Balakrishnan, CryptDB: Protecting confidentiality with encrypted query processing, in: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, in: SOSP 1́1, ACM, New York, NY, USA, 2011, http://dx.doi.org/10.1145/2043556.2043566.

[24] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, S. Badrinarayanan, Big data analytics over encrypted datasets with seabed, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016, pp. 587–602.

[25] S.L. Tu, M.F. Kaashoek, S.R. Madden, N. Zeldovich, Processing analytical queries over encrypted data, URL: https://vm-web.pdos.csail.mit.edu/papers/tu-monomi-cr-vldb13.pdf. [Last visited on September 09, 2020].

[26] X. Yuan, Y. Guo, X. Wang, C. Wang, B. Li, X. Jia, Enckv: An encrypted key-value store with rich queries, in: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, 2017, pp. 423–435.

[27] E. Barbierato, M. Gribaudo, M. Iacono, Performance evaluation of NoSQL big-data applications using multi-formalism models, Future Gener. Comput. Syst. 37 (2014) 345–353.

[28] A. Turner, A. Fox, J. Payne, H.S. Kim, C-mart: Benchmarking the cloud, IEEE Trans. Parallel Distrib. Syst. 24 (6) (2012) 1256–1266.

[29] R. Cattell, Scalable SQL and NoSQL data stores, ACM SIGMOD Rec. 39 (4) (2011) 12–27.

[30] J.R. Lourenço, B. Cabral, P. Carreiro, M. Vieira, J. Bernardino, Choosing the right NoSQL database for the job: a quality attribute evaluation, J. Big Data 2 (1) (2015) 18.

[31] L. Wiese, T. Waage, M. Brenner, CloudDBGuard: A framework for encrypted data storage in NoSQL wide column stores, Data Knowl. Eng. (2019) 101732.

[32] A.M. Eassa, M. Elhoseny, H.M. El-Bakry, A.S. Salama, NoSQL Injection Attack Detection in Web Applications Using RESTful Service, Program. Comput. Softw. 44 (6) (2018) 435–444.

[33] K. Sahatqija, J. Ajdari, X. Zenuni, B. Raufi, F. Ismaili, Comparison between relational and NOSQL databases, in: 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), IEEE, 2018, pp. 0216–0221.

[34] T. Müller, F.C. Freiling, A. Dewald, TRESOR Runs Encryption Securely Outside RAM, in: USENIX Security Symposium, Vol. 17, 2011.

[35] H. Shafagh, A. Hithnawi, A. Droescher, S. Duquennoy, W. Hu, Talos: Encrypted query processing for the Internet of Things, in: Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems, in: SenSys '15, ACM, New York, NY, USA, 2015, pp. 197–210, http://dx.doi.org/10.1145/2809695.2809723, URL: http://doi.acm.org/10.1145/2809695.2809723.

[36] M. Naveed, S. Kamara, C.V. Wright, Inference attacks on property-preserving encrypted databases, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, ACM, 2015, pp. 644–655.

[37] P.G. Alves, D.F. Aranha, A framework for searching encrypted databases, J. Internet Serv. Appl. 9 (1) (2018) 1.

[38] G. Yubin, Z. Liankuan, L. Fengren, L. Ximing, A solution for privacy-preserving data manipulation and query on NoSQL database, J. Comput. 8 (6) (2013) 1427–1432.

[39] S.S. Sathya, P. Vepakomma, R. Raskar, R. Ramachandra, S. Bhattacharya, A review of homomorphic encryption libraries for secure computation, 2018, arXiv preprint arXiv:1812.02428.

[40] J. Philipps, B. Rumpe, Refinement of pipe-and-filter architectures, in: International Symposium on Formal Methods, Springer, 1999, pp. 96–115.

[41] A. Rafique, D. Van Landuyt, B. Lagaisse, W. Joosen, On the performance impact of data access middleware for NoSQL data stores: a study of the trade-off between performance and migration cost, IEEE Trans. Cloud Comput. 6 (3) (2015) 843–856.

[42] kunerd, JPaillier, 2020, https://github.com/kunerd/jpaillier. [Last visited on January 13, 2020].

[43] P. Paillier, Public-key cryptosystems based on composite degree residuosity classes, in: International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 1999, pp. 223–238.

[44] B. Thorne, et al., A Java library for Paillier partially homomorphic encryption based on python-paillier, 2017, https://github.com/n1analytics/javallier.

[45] Peter Olson, BigInteger.js, 2020, https://github.com/peterolson/BigInteger.js/. [Last visited on May 18, 2020].

[46] D.X. Song, D. Wagner, A. Perrig, Practical techniques for searches on encrypted data, in: Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000, IEEE, 2000, pp. 44–55.

[47] R. Curtmola, J. Garay, S. Kamara, R. Ostrovsky, Searchable symmetric encryption: improved definitions and efficient constructions, J. Comput. Secur. 19 (5) (2011) 895–934.

[48] E.-J. Goh, et al., Secure indexes, in: IACR Cryptology ePrint Archive, Vol. 2003, 2003, p. 216.

[49] S. Kamara, C. Papamanthou, T. Roeder, Dynamic searchable symmetric encryption, in: Proceedings of the 2012 ACM Conference on Computer and Communications Security, 2012, pp. 965–976.

[50] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, M. Steiner, Highly-scalable searchable symmetric encryption with support for boolean queries, in: Annual Cryptology Conference, Springer, 2013, pp. 353–373.

[51] S. Kamara, T. Moataz, Boolean searchable symmetric encryption with worst-case sub-linear complexity, in: Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2017, pp. 94–124.

[52] D. Cash, J. Jaeger, S. Jarecki, C.S. Jutla, H. Krawczyk, M.-C. Rosu, M. Steiner, Dynamic searchable encryption in very-large databases: data structures and implementation, in: NDSS, Vol. 14, Citeseer, 2014, pp. 23–26.

[53] D. Cash, S. Tessaro, The locality of searchable symmetric encryption, in: Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2014, pp. 351–368.

[54] R. Bost, $o\varphi o\varsigma$: Forward secure searchable encryption, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 1143–1154.

[55] M. Etemad, A. Küpçü, C. Papamanthou, D. Evans, Efficient dynamic searchable encryption with forward privacy, in: Proceedings on Privacy Enhancing Technologies, Vol. 2018, Sciendo, 2018, pp. 5–20.

[56] J. Ghareh Chamani, D. Papadopoulos, C. Papamanthou, R. Jalili, New constructions for forward and backward private symmetric searchable encryption, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018.

[57] Z. Brakerski, C. Gentry, V. Vaikuntanathan, Fully Homomorphic Encryption without Bootstrapping, 2011, Cryptology ePrint Archive, Report 2011/277, https://eprint.iacr.org/2011/277.

[58] I. Chillotti, N. Gama, M. Georgieva, M. Izabachène, TFHE: fast fully homomorphic encryption over the torus, J. Cryptol. 33 (1) (2020) 34–91.

[59] C. Gentry, S. Halevi, N.P. Smart, Fully homomorphic encryption with polylog overhead, in: Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2012, pp. 465–482.

[60] M.R. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K.E. Lauter, et al., Homomorphic encryption standard., IACR Cryptol. ePrint Arch. 2019 (2019) 939.

[61] M. Bellare, A. Boldyreva, A. O'Neill, Efficiently-searchable and deterministic asymmetric encryption, Cryptol. ePrint (2006).

[62] M. Bellare, A. Boldyreva, A. O'Neill, Deterministic and efficiently searchable encryption, in: Annual International Cryptology Conference, Springer, 2007, pp. 535–552.

[63] M. Bellare, M. Fischlin, A. O'Neill, T. Ristenpart, Deterministic encryption: Definitional equivalences and constructions without random oracles, in: Annual International Cryptology Conference, Springer, 2008, pp. 360–378.

[64] A. Boldyreva, S. Fehr, A. O'Neill, On notions of security for deterministic encryption, and efficient constructions without random oracles, in: Annual International Cryptology Conference, Springer, 2008, pp. 335–359.

[65] F. Hahn, F. Kerschbaum, Searchable encryption with secure and efficient updates, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, 2014.

[66] A. Boldyreva, N. Chenette, A. O'Neill, Order-preserving encryption revisited: Improved security analysis and alternative solutions, in: Annual Cryptology Conference, Springer, 2011, pp. 578–595.

[67] F. Kerschbaum, Frequency-hiding order-preserving encryption, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 656–667.
[68] Z. Liu, X. Chen, J. Yang, C. Jia, I. You, New order preserving encryption model for outsourced databases in cloud environments, J. Netw. Comput. Appl. 59 (2016) 198–207.
[69] Oracle, Oracle TDE, 2020, https://docs.oracle.com/database/121/ASOAG/introduction-to-transparent-data-encryption.htm. [Last visited on April 30, 2020].
[70] MySQL, MySQL Transparent Data Encryption, 2020, https://www.mysql.com/products/enterprise/tde.html. [Last visited on April 30, 2020].
[71] PostgreSQL, PostgreSQL Transparent Data Encryption, 2020, https://wiki.postgresql.org/wiki/Transparent_Data_Encryption. [Last visited on April 30, 2020].
[72] MariaDB, Data-at-Rest Encryption, 2020, https://mariadb.com/kb/en/data-at-rest-encryption-overview/. [Last visited on April 30, 2020].
[73] MongoDB, Encryption at Rest, 2020, https://docs.mongodb.com/manual/core/security-encryption-at-rest/. [Last visited on April 30, 2020].
[74] DataStax, Apache cassandra transparent data encryption, 2020, https://docs.datastax.com/en/security/6.7/security/secEncryptEnable.html. [Last visited on April 30, 2020].
[75] Apache HBase, Securing access to your data, 2020, https://hbase.apache.org/book.html. [Last visited on April 30, 2020].
[76] AWS, DynamoDB Encryption at Rest, 2020, https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/EncryptionAtRest.html. [Last visited on April 30, 2020].
[77] MongoDB, Client-side field level encryption, 2020, https://docs.mongodb.com/manual/core/security-client-side-encryption/. [Last visited on April 30, 2020].
[78] Amazon Web Services, What is the Amazon DynamoDB Encryption Client? 2020, https://docs.aws.amazon.com/dynamodb-encryption-client/latest/devguide/what-is-ddb-encrypt.html. [Last visited on April 30, 2020].
[79] W. Zheng, A. Dave, J.G. Beekman, R.A. Popa, J.E. Gonzalez, I. Stoica, Opaque: An oblivious and encrypted distributed analytics platform, in: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), 2017, pp. 283–298.
[80] C. Priebe, K. Vaswani, M. Costa, Enclavedb: A secure database using SGX, in: 2018 IEEE Symposium on Security and Privacy (SP), IEEE, 2018, pp. 264–278.
[81] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, A.-R. Sadeghi, HardIDX: Practical and secure index with SGX, in: IFIP Annual Conference on Data and Applications Security and Privacy, Springer, 2017, pp. 386–408.
[82] M.H. Diallo, B. Hore, E.-C. Chang, S. Mehrotra, N. Venkatasubramanian, Cloudprotect: managing data privacy in cloud applications, in: 2012 IEEE Fifth International Conference on Cloud Computing, IEEE, 2012, pp. 303–310.
[83] E.H. Beni, B. Lagaisse, W. Joosen, A. Aly, M. Brackx, DataBlinder: A distributed data protection middleware supporting search and computation on encrypted data, in: Proceedings of the 20th International Middleware Conference Industrial Track, 2019, pp. 50–57.
[84] A. Rafique, D. Van Landuyt, W. Joosen, Persist: Policy-based data management middleware for multi-tenant SaaS leveraging federated cloud storage, J. Grid Comput. 16 (2) (2018) 165–194.
[85] A. Rafique, D. Van Landuyt, V. Reniers, W. Joosen, Leveraging NoSQL for scalable and dynamic data encryption in multi-tenant SaaS, in: 2017 IEEE Trustcom/BigDataSE/ICESS, IEEE, 2017.
[86] A. Rafique, D. Van Landuyt, V. Reniers, W. Joosen, Towards scalable and dynamic data encryption for multi-tenant SaaS, in: Proceedings of the Symposium on Applied Computing, 2017.
[87] E.H. Beni, B. Lagaisse, W. Joosen, WF-Interop: adaptive and reflective rest interfaces for interoperability between workflow engines, in: Proceedings of the 14th International Workshop on Adaptive and Reflective Middleware, 2015, pp. 1–6.
[88] E.H. Beni, B. Lagaisse, W. Joosen, Adaptive and reflective middleware for the cloudification of simulation & optimization workflows, in: Proceedings of the 16th Workshop on Adaptive and Reflective Middleware, 2017, pp. 1–6.
[89] Microsoft, Always encrypted engine, 2020, https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/always-encrypted-database-engine?view=sql-server-ver15. [Last visited on April 30, 2020].