Security&privacy issues and challenges in NoSQL databases

Sabrina Sicari*‡, Alessandra Rizzardi*, Alberto Coen-Porisini*
*Dipartimento di Scienze Teoriche e Applicate, Università degli Studi dell'Insubria,
via O. Rossi 9 - 21100 Varese (Italy)
‡Corresponding author
Email: {sabrina.sicari; alessandra.rizzardi; alberto.coenporisini}@uninsubria.it

***Abstract*—Organizing the storing of information and data retrieval from databases is a crucial issue, which has become more critical with the spreading of cloud and Internet of Things (IoT) based applications. In fact, not only the network's traffic has increased, but also the amount of memory and the mechanisms needed to manage the so-called *Big Data* efficiently. Relational databases, based on SQL, are giving way to the NoSQL ones due to their efficiency in managing the heterogeneous information gathered from IoT environments. Such data can be stored, in a distributed manner, within the IoT network's devices or in the cloud. Hence, security and privacy concerns naturally emerge regarding access control, authentication, and authorization requirements. This paper analyses the current state of the art of security and privacy solutions tailored to NoSQL databases, particularly Redis, Cassandra, MongoDB, and Neo4j stores. The paper also aims to shed light on current challenges and future research directions in the field databases' security in the IoT scenario.**

***Keywords*—NoSQL databases, Internet of Things, Access Control, Authentication, Authorization, Security, Privacy**

I. Introduction

The growth in the amount of data transmitted throughout network systems is ever increasing, mainly due to the spreading of the Internet of Things (IoT) and cloud-based applications. As a consequence, information systems have to process and organize much more information, thus requiring the definition and diffusion of novel and efficient mechanisms to manage and persistently store data [1]. IoT systems usually handle high volumes of heterogeneous information in a distributed manner. Hence, protocols and strategies adopted in such environments must be: (i) scalable, since many end-devices could be involved in data exchanges; (ii) dynamic, in order to be able to process different kinds of information; (iii) reliable, both in terms of robustness and response times.

Concerning data persistency, traditional SQL-based stores, which are commonly referred to as *Relational Database Management Systems (RDBMSs)*, cannot sufficiently address the requirements above, mainly due to their low flexibility in terms of data structures. Instead, NoSQL databases address the emerged challenge by providing high data availability, service reliability, and flexibility to manage the so-called *Big Data* [2]. The term NoSQL was first introduced in 1998 to indicate an open-source relational database that did not use a SQL interface; in such years, NoSQL databases have received increasing attention from companies and organizations for the ease and efficiency in managing high volumes of dynamic, heterogeneous, and often unstructured data. In particular, the main advantages of NoSQL databases are the following: (i) efficient execution of data reading and writing operations; (ii) low latency; (iii) easy to expand; (iv) low costs in terms of management and operations. Moreover, some NoSQL systems automatically perform the $sharding$ task, which assigns a subset of the data to each node/processor belonging to the network. Such a partitioning method refers to $horizontalscalability$, which consists in scaling the database by adding new processors along with their disks/nodes [3]. Due to the features mentioned above, NoSQL databases better satisfy the needs of IoT-related environments, with respect to traditional SQL-based ones.

Apart from scalability and performance, security and privacy requirements represent one of the most difficult challenges NoSQL databases face nowadays. In fact, NoSQL databases were not initially designed and integrated with security and privacy-related functionalities, such as data encryption, authentication, and authorization mechanisms. Furthermore, sharding further poses

security risks due to its highly distributed nature. The main security risks encountered by NoSQL databases are related to not encrypted data storage, unauthorized exposure of both data and backup (or replicated data), insecure communication over the network [4].

Available data models, for NoSQL databases, are classified as: $key - value$, $column - oriented$, $document$ and $graphs$. In this paper, one datastore from each NoSQL database category, just mentioned, is described and compared, with respect to its functionalities and reliability. Hence, the state-of-the-art of the most popular NoSQL databases, namely $Redis$, $Cassandra$, $MongoDB$, and $Neo4j$, is analyzed, mainly focusing on the satisfaction of security and privacy requirements.

Concerning the literature, some available works address topics related to NoSQL databases. For example, the authors of [5] propose a comparative classification model, putting in relation functional and non-functional requirements to techniques and algorithms employed in NoSQL databases; the main goal is to derive a decision tree to guide practitioners and researchers in the choice of the NoSQL database, which is most suitable for their purposes. Classification and evaluation of NoSQL databases in *Big Data Analytics* are reported in [6], aiming to help users, and especially organizations being aware of strengths and weaknesses of various NoSQL database approaches to support applications that process vast volumes of data. With the same scope, the authors of [7] analyze strategies and corresponding features, strengths, and drawbacks of NoSQL databases, highlighting open challenges.

Instead, focusing on reliability, the work presented in [8] summarizes the security-related features supported (or not supported) by some NoSQL databases; while, in [9], an overview of existing access control models, in different NoSQL databases, is proposed. Security issues of sharded NoSQL stores are investigated in [10]; while [11] presents the native security functionalities supported by some NoSQL databases. Finally, the most recent [12] envisions a system, which enables the execution of different types of search and aggregate queries over encrypted data for a wide range of different NoSQL databases without modifying the underlying database engine. Also, the support for built-in data security features in the most popular NoSQL databases is discussed in [12]. Instead, in this work, we aim to provide a more comprehensive description of available solutions,focusing on architectures and pointing out and discussing open challenges in the field of security for NoSQL databases.

The rest of the paper is organized as follows. Section II describes the available NoSQL data models. Section III briefly compares RDBMSs and NoSQL databases in terms of structure and functionalities, while Section IV introduces the security and privacy requirements investigated throughout the paper. Then, Section V analyses the security and privacy-related solutions concerning Redis, Cassandra, MongoDB, and Neo4j databases. Section VI provides a discussion about the outcomes of the conducted research, while Section VII ends the paper.

II. NoSQL data models

NoSQL databases aim to scale in distributed systems horizontally and are classified based on their data model and data retrieval mode, which are: key-value, column-oriented, document, and graph, as detailed in the following sections. Table I associates some existing NoSQL databases with the corresponding data model. Among them, we can find the four databases analyzed in this paper (one for each discussed data model), namely, Redis, Cassandra, MongoDB, and Neo4j.

TABLE I. Data model for existing NoSQL databases

Data model	Database
Key-Value	Redis, Dynamo, Riank
Column-Oriented	HBase, Cassandra, Hypertable
Document	MongoDB, CouchDB, RaverDB
Graph	Neo4j, Titan, Core Data

A. Key-value data model

The key-value model is schema-less and uses a hash table (i.e., a sort of lookup table), where keys are stored as indexes towards data [13]. Hash tables are suitable for quickly and efficiently finding values in large data sets. In addition, with its replication function, it offers fast, secure, and less expensive access to information along with high availability and durability. The only weaknesses may be the lack of a scheme, making it difficult to create customized views of data. Data are stored in the form of rows as structured data, but can also be stored as *JavaScript Object Notation (JSON)* objects. Figure 1 shows a generic example of key-value data model, where a column identifies the keys, which

must be unique, and the other contains the attributed values in the form of key-value pairs.

DATABASE	
KEY	**VALUE**
key_0	*ID*: id_0 *field_name_0*: field_value_a *field_name_1*: field_value_b *field_name_2*: field_value_c
key_1	*ID*: id_1 *field_name_0*: field_value_d *field_name_1*: field_value_e *field_name_2*: field_value_f

Fig. 1. Key-value model

Redis[1] is an example of advanced and open-source key-value store. Today, Redis provides response times which are less than a millisecond, enabling millions of requests per second for real-time applications, such as those related to gaming and IoT fields. Redis is ideal for implementing high-availability in-memory caching, and reducing latency in data accessing. Typical use cases are storing database query results, web pages and frequently accessed (i.e., cached) objects (e.g., images and files).

Unlike other key-value stores [14], Redis offers data structures which are suitable for managing any type of binary data, such as arrays, bytes, numbers, strings, XML documents, images, etc. Hence, each key-value couple is a pair of binary strings. To access key-value pairs, Redis offers the following elementary operations: (i) *set (key-value)*, which adds or modifies a key-value pair in the database; (ii) *get (key)*, which retrieves the value related to the given key, from the database; (iii) *delete (key)*, which deletes the key-value pair from the database, based on the key, given as input. Furthermore, since atomic operations are performed, concurrent accesses by multiple clients will not affect data updates.

KEY	VALUE
collection_name: key_0	{*field_name_0*: field_value_a, *field_name_1*: field_value_b, *field_name_2*: field_value_c, ...}
collection_name: key_1	{*field_name_0*: field_value_d, *field_name_1*: field_value_e, *field_name_2*: field_value_f, ...}

Fig. 2. Redis: key-object representation

There are two data representation strategies. The first one, adopts a single key-value pair for each object. More in detail, the key is a concatenation of the collection's name and the object's identifier; while the value is a serialization of the entire content of the object, as shown in Figure 2. Such a mechanism allows the efficient recovery of the object, which can take place through a single recovery operation. The second one uses several key-value pairs for each field belonging to the object. In this case, the key is composed of the collection's name, the object's identifier and the name of the first level field; while the value is that of the field in the specified object, as shown in Figure 3. Such a strategy offers a further operation, named *key (pattern)*, which identifies all the keys corresponding to the pattern given as input. Moreover, Redis provides hashes to store and query the objects contained in the database. In this way, a hash value can be used to refer to the various fields of an object, in the form of field-value pairs, as shown in Figure 4; then, an object can be retrieved through *hgetall (key)* operator, which finds the field-value pairs associated with the reference key.

KEY	VALUE
collection_name: key_0/field_name_0	field_value_a
collection_name: key_0/field_name_1	field_value_b
collection_name: key_0/field_name_2	field_value_c
...	...
collection_name: key_0/field_name_n	field_value_...

Fig. 3. Redis: key-field representation

KEY	HASH VALUE
collection_name: key_0	*field_name_0*: field_value_a
	field_name_1: field_value_b
	field_name_2: field_value_c
	...

Fig. 4. Redis: key-hash representation

B. Column-oriented data model

Column-oriented databases, also known as *column-family* stores [15], can be considered an evolution of key-value stores, since data are still represented as hash maps, but can reach two or more indexing levels. In fact, information is stored in cells, grouped into columns, which are further grouped into families. Columns' families must be almost pre-defined, thus making such a

[1]Redis database, https://redis.io

data model less flexible than key-value and document-oriented ones. Due to horizontal partitioning, the main advantages are quick access to information, scalability, and data aggregation. $Google$ has introduced such a schema by developing *BigTable*, to manage the amount of storage required by $Google$ applications such as $Gmail$, $GoogleMaps$, website indexing, etc. Figure 5 shows a generic example of this model, where each column family groups a set of data in the form of key-value pairs; each column family group is uniquely identified.

DATABASE	
ID: 0	
column-family: column-family_id_0	*field_name_0*: field_value_a *field_name_1*: field_value_b *field_name_2*: field_value_c
column-family: column-family_id_1	*field_name_3*: field_value_d *field_name_4*: field_value_e
ID: 1	
column-family: column-family_id_0	*field_name_0*: field_value_f *field_name_1*: field_value_g *field_name_2*: field_value_h
column-family: column-family_id_1	*field_name_3*: field_value_i *field_name_4*: field_value_j

Fig. 5. Column-oriented model

Cassandra represents a hybrid key-value/column-oriented database, even if it is de facto considered a column-oriented store, due to its structure based on hash-maps. Originality developed for the social network $Facebook$, Apache Cassandra[2] further obtained contributes from IBM, $Twitter$, $Instagram$, $Spotify$, $Rackspace$ and $DataStax$. Moreover, it has been adopted by $eBay$, $GitHub$ and $Netflix$ to store large volumes of data and to be able to retrieve them promptly. In fact, Cassandra can handle petabytes of information and thousands of concurrent operations per second in both hybrid-cloud and multi-cloud environments. Hence, it is a powerful database, and a lot of development is available as well as the support for their adoption, such as $DataStax$[3], which is collaborating with the Cassandra community to make this store simpler to adopt.

The keys' values are columns and they are grouped into sets of columns, named *columns' families*; in few words, as shown in Figure 6, each key identifies a

KEY	COLUMN_FAMILY_0		COLUMN_FAMILY_1	
key_0	column_0	column_1	column_2	column_3
	value_a	value_b	value_c	value_d
key_1	column_0	column_1	column_2	column_3
	value_e	value_f	value_g	value_h

Fig. 6. Cassandra: column-family representation

row containing a variable number of elements, and applications using Cassandra can specify the order of columns within a family. Such columns are further distinguished in $super$ and $simple$ columns, as follows:

- $Column$: the basic element is a record consisting of a timestamp, a column name and a value
- $Super - Column$: it is a structure, where a column can store a dynamic list of columns
- $Column - Family$: it is a set of columns, where the number of columns can vary over time; Cassandra admits no limitations in the columns' number, but such a schema must be handled at the application level
- $Key - Space$: it identifies a set of $Column - Family$, which is a list of columns or super-columns identified by a row key.

Therefore, in Cassandra, the values, contained in tables, are addressed by the triple *(row key, column key, timestamp)*, where the column key is intended as: (i) *(column family: column)* in case of simple columns contained in the column family; (ii) *(column family: super-column: column)* in case of columns referring to a super-column.

Cassandra's APIs consist of three operations: (i) *get (table, key, column name)*; *insert (table, key, record mutation)*; *delete (table, key, column name)*. Column name identifies a column or a super-column within a column family. All requests made by client applications are routed to a server belonging to a Cassandra cluster, which determines replicas. In fact, Cassandra performs data partitioning among different clusters to balance data and processing loads, through hash functions, which preserve row keys' order. Cassandra provides two replication strategies to guarantee cluster's scalability and durability, which determine the nodes where replicas must be located; the replication factor is defined as the total number of replicas within a cluster and, as a general rule such a number must never exceed the cluster nodes. If such a number is exceeded, write

[2]Apache Cassandra, https://cassandra.apache.org

[3]The Open, Multi-Cloud Stack for Modern Data Apps, https://www.datastax.com

operations are denied. These are the two replication strategies in Cassandra:

- $SimplyStrategy$: it is a replication strategy within a single data center, where the first replica is placed on a node determined by the coordinator, then further replicas are placed on the other nodes in a clockwise direction
- $NetworkTopologyStrategy$: it is a strategy used in the presence of multiple data centers, to store multiple copies of data in a distributed manner.

A Cassandra cluster server runs modules that provide the following functionalities: partitioning, cluster membership, error detection, and storage engine. The storage engine module can perform synchronously and asynchronously write operations. All system control messages are based on UDP; while messages relating to replicas' management and requests' routing are based on TCP [16].

Cassandra uses its query declarative language, named *CQL (Cassandra Query Language)*. Unlike SQL, the CQL model does not support binary operations such as $join$ operations; for such a reason, a set of rules have been further integrated by the literature, which ensures efficiency and scalability, thus making Cassandra suitable for *Big Data* based applications [17].

C. Document data model

Document-based model is the most used NoSQL data model [18], since it is able to manage structured, semi-structured and unstructured data. Documents resemble the rows of relational databases, but are more flexible, since they are schema-less. A document can contain multiple key-value pairs or even nested documents; it has a unique identifier, leading to a disadvantage: a new index must be created for each query type. Figure 7 shows a generic example of a document-based data model, where documents can contain pointers to other documents inside the database.

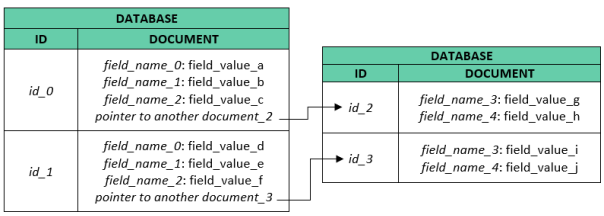

Fig. 7. Document-based model

MongoDB[4] is an example of distributed, document-oriented, and general-purpose database, targeted to modern application developers and to the cloud. It stores data in the form of JSON documents, without defining a particular schema, which makes the system flexible for heterogeneous data management. Note that JSON is a Java object serialization format, particularly suitable for fast reading and writing operations, so it is ideal for real-time applications. MongoDB is ranked in the first place among NoSQL databases according to $DB-Engines$, as presented in Section II. For such a reason, MongoDB provides official drivers for a variety of programming languages and development environments, including $Google$, $Adobe$, $Cisco$, $Bosh$, $Nokia$ and $Flipkart$ [19].

Besides documents organized in key-value pairs, MongoDB adds a further organization level, which consists of the so-called $collections$, aiming at grouping together similar documents. Different documents may be linked by means of identifiers, as shown in Figure 8. A single instance of MongoDB can manage more than one database independent of each other. To manage documents, for example by adding new data, retrieving information, updating or deleting data, MongoDB uses the *CRUD (Create, Read, Update, Delete)* functions [20]. A strength of MongoDB is that it supports all the indexing techniques also adopted in relational databases, to speed up the search and to order the data within the store. Data/documents can also be discarded after a certain elapsed time, by means of the *TTL (Time To Live)* index.

Moreover, MongoDB puts in act a sharding technique, by enabling partitioned clusters, whose components are the following:

- $Snippets$: they represent independent databases
- $Mongos$: they act as routers and are responsible for routing read and write requests from the application to the shards (i.e., the snippets). They also cache useful metadata from the configuration server, in order to manage requests without overloading the server itself
- *Configuration servers*: they represent special $mongos$, which store, in a configuration database, the metadata related to the partitioned cluster. Usually, more than one configuration server is put in place, in order to guarantee the functionalities of the cluster, even if a configuration server is not accessible for some reason.

[4]MongoDB, https://www.mongodb.com/

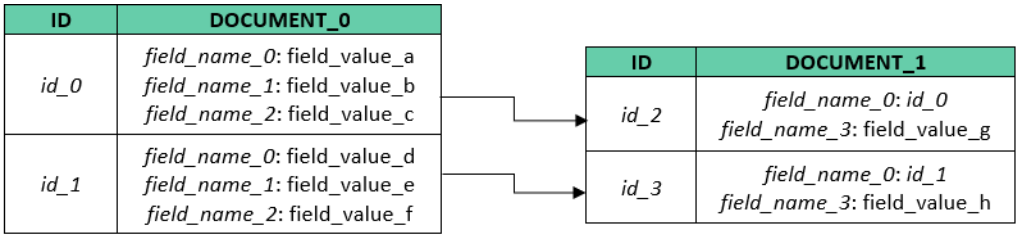

Fig. 8. MongoDB: documents' representation

MongoDB provides two mechanisms for replication: (i) *master-slave*, as presented in Section III; (ii) *replica sets*, which become the standard for MongoDB, since it guarantees the automated *fail-over*. Note that a *replica set* is a set of MongoDB instances, which contains the same information, and the basic behavior is the following: (i) the primary node receives write operations from clients; (ii) the primary node performs the write operations and asynchronously replicates them on the secondary nodes; (iii) if the primary node results offline for any reason, one of the secondary nodes will be automatically promoted/elected to primary, making the system again available with all pre-existing data (i.e., fail-over). *Replica sets* also provide improvements in terms of easier recovery and more sophisticated deployment topologies.

MongoDB provides its own query language, which is named *MongoDB Query Language (MQL)*. Note that, recently, it integrated the $leftjoin$ operations among collections, but it does not yet support the well-known $innerjoin$ operation; as a consequence, data must be stored in a single large collection organized in nested JSON objects. In [21], a novel algorithm is presented to overcome such a limitation.

D. Graph data model

Graph datasets only manage semi-structured data, so there is no pre-defined schema [22]. A graph database (see Figure 9) is a particular document-oriented database, where some documents are treated as relationships to connect other documents. The graph can be partitioned to improve the performance, thus leading this schema to be $ACID$-compatible, offering rollback support and ensuring data consistency. Graph databases are adopted for social networking, bio-informatics, and cloud management applications.

Neo4j[5] is an example of graph database. It is usually employed in health, government and logistics scenarios, but also in the analysis of social networks' behavior, due to its intrinsic nature (i.e., the graph). Also, companies, such as $Walmart$ and $eBay$, adopt Neo4j to understand the behavior and preferences of online customers, in order to make offers and recommendations in real-time; in this way, such companies can rapidly connect products and buyers by promptly understanding customers' needs and product trends. It is considered one of the leading graph databases for being able to manage large data sets; moreover, it provides detailed documentation and many integration tools and drivers, which support programming languages, such as $Java$, PHP, $Node.js$ and $Python$. Neo4j, compared to other graph databases, is easier to use, despite the need for prior knowledge about the query language [23].

In fact, Neo4j is used to manage not only data, but also the relationships among them. Unlike Redis, Cassandra and MongoDB, which organize data into rows, columns and tables, Neo4j provides a highly flexible structure, generated by the relationships stored among records (see Figure 10, which represents some generic collections, relationships and properties), thus allowing a more rapid execution of queries [24]. Neo4j puts data in a series of *store files*, which is included in a single folder; each *store file* contains information about a single part of the graph, thus optimizing and facilitating the *graph traversal*. It is considered an $ACID$ compliant transactional database and is available in three versions: (i) $Community$, which is shared a trial version; (ii) $Enterprise$, which is designed for commercial deployments; (iii) $Aura$, which is a cloud DBaaS to remotely interact with Neo4j, with the support of native engineers.

To improve data retrieval's performance, Neo4j integrates two caching mechanisms: (i) the file system cache, which contains the most requested relationships; (ii) the object cache, which contains not only the relationships, but also the properties of the most vis-

[5]Neo4j database, https://neo4j.com

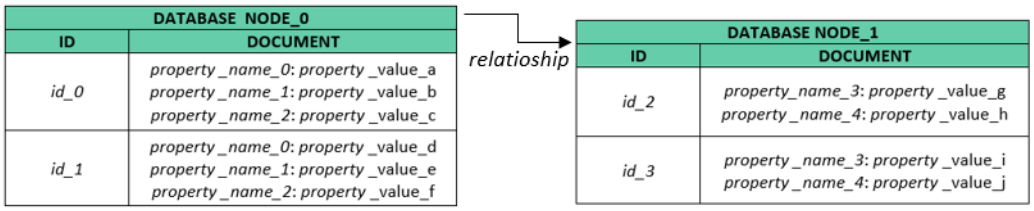

Fig. 9. Graph model

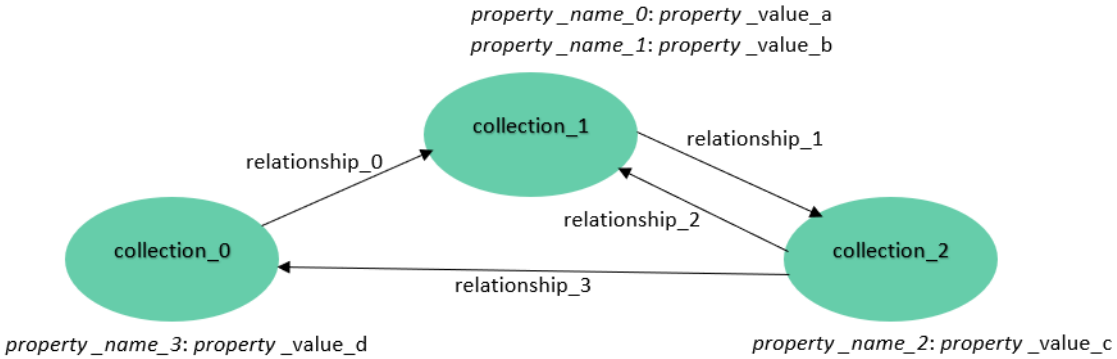

Fig. 10. Neo4j: graph representation

ited nodes. The *HA (High Availability)* cluster module characterizes the clustering capabilities of the system [25].

In general, graph databases do not yet provide any standardization concerning the language for traversing and inserting data within the graph; such a lack leads to definitions and implementations of different frameworks for data management. For example, the work in [26] proposes a framework, which should be suitable with four types of graph databases: $Neo4j$, $OrientDB$, $Titan$ and DEX. $Gremlin$ and $Cypher$ are used as query languages to traverse the Neo4j graphs. More in detail, $Gremlin$ is a programming language specific to traversing and manipulating graphs and it is just adopted by many graph database vendors, so it tries to represent a sort of standard language; while $Cypher$ is a SQL-inspired declarative query language, which tries to avoid the need for executing traversal operation in the application's code [27].

III. RDBMS and NoSQL databases comparison

NoSQL databases are often compared with relational databases, which are essentially based on SQL, considering them as an alternative approach, to be used in scenarios that require scalability, flexibility, and the management of large volumes of data [28]. In general, SQL databases are based on a static schema, where data are structured; each change must be carefully evaluated to avoid introducing errors or maintenance issues. Instead, NoSQL databases handle well-structured, semi-structured, and also unstructured schema, which easily adapts to changes in the data type and/or structure, thus becoming particularly suitable for evolving and heterogeneous scenarios, such as those related to IoT applications [29].

In relational databases, redundancy is avoided by means of normalization, which is a mechanism for splitting data into small logical tables. In this way, the storage space is optimized, but the performance of the database queries will degrade due to the complexity of data retrieval. RDBMSs maintain data consistency among applications, by ensuring that multiple database instances always contain the same data. In NoSQL databases, data are stored in collections usually without relationships or normalization among them, causing redundancy, but improving availability. In fact, NoSQL databases transparently perform data replication across more nodes. As a disadvantage, the performance degrades due to "writing" queries, since they must be replicated on each node. Two main replication models are adopted in NoSQL systems: *master-slave* and *master-master* replication. The former allows the slaves to hold a copy (i.e., *master copy*) of the data, only for reading purposes; while the master is enabled to both

write and read the data, thus ensuring data consistency. The latter allows reading and writing operations to any copy, without guaranteeing data consistency, which should be managed at the application level [30].

Concerning database's transactions, relational databases are characterized by *ACID (Atomicity - Consistency - Isolation - Durability)* properties; while NoSQL databases present *BASE (Basically Available, Soft state, Eventual consistency)* properties. The $ACID$ paradigm points out the four features that a RDBMS must guarantee for considering valid the carried out transactions [31]:

- $Atomicity$: at the end of each transaction, its effects must be totally visible, otherwise, the system returns to the initial state
- $Consistency$: each performed transaction must leave the database in a consistent state
- $Isolation$: each performed transaction must be independent of the others so that possible failures do not affect other transactions
- $Durability$: the results of a completed transaction must be persistent so that results must not be lost; note that almost all DBMSs implement a recovery system in case of failures.

Implementing $ACID$ principles in NoSQL databases is not trivial, due to the strong consistency ensured among data in RDMBS. For such reasons, NoSQL databases refer to the $BASE$ paradigm, which consists of the following three features:

- *Basically Available*: the system must ensure data availability
- *Soft state*: system state can change over time and data consistency must be managed by the developer (i.e., not by the database engine)
- *Eventual consistency*: when new data are added to the distributed NoSQL system, they will gradually spread, still maintaining consistency.

Hence, $BASE$ paradigm aims to guarantee data availability, scalability, fault tolerance, and flexibility, despite leaving the consistency's management to application level's developers.

On the one side, NoSQL databases are designed for horizontal scaling (i.e., to scale out), which occurs if the increase in resources corresponds to the increase in system nodes, so that the system can balance the workload. Such behavior ensures that NoSQL databases can store and process huge amounts of data, making them particularly suitable for *Big Data* based applications. On the other side, relational databases usually support vertical scalability (i.e., to scale up), which is achieved in this way: to increase the performance of the whole system, the resources of a single node of the system are increased, for example, by using a CPU with higher frequency or by increasing the available memory. The disadvantage is the cost, since upgrading the performance of a node could be more expensive than adding a new one (which presents the same performance as the already existing node), as done by NoSQL stores [32].

According to the CAP *(Consistency - Availability - Partitioning)* theorem, formulated by Eric Brewer [33], a distributed system cannot provide all such three properties at the same time:

- $Consistency$: data are always the same in each replica on each server (i.e., once a change has been made, all devices belonging to the same distributed system are modified accordingly)
- $Availability$: the system is always able to give an answer to a query; hence the data are always accessible
- *Partition Tolerance*: the system continues to run even in case of failures.

Basically, CAP theorem shows that a distributed system is able to satisfy at most two of such requirements, generating the following combinations: (i) CP, Coherence and Partition Tolerance; (ii) AP, Availability and Partition Tolerance; (iii) CA Consistency and Availability. NoSQL databases, as shown in Figure 11, rank in the CP Tolerance and AP Tolerance segments. While, RDBMS are placed in the CA segment.

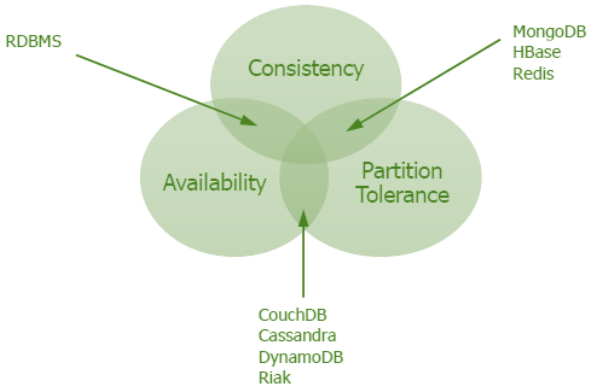

Fig. 11. CAP *(Consistency - Availability - Partitioning)* theorem

Relational databases use the SQL standard concerning the query language, which handles complex queries through a standardized interface. Instead, NoSQL

TABLE II. COMPARISON BETWEEN SQL AND NOSQL RDBMS

Feature	SQL	NoSQL
Schema	structured	structured, semi-structured and unstructured
Redundancy	avoided with normalization	replication management
Atomicity	guaranteed	guaranteed
Consistency	guaranteed	to be managed by developers
Isolation	guaranteed	fault tolerance guaranteed
Durability	guaranteed	data availability guaranteed
Scalability	vertical scaling	horizontal scaling
Query language	standard	custom

databases do not have a standard query language, but provide their own query language; note that many NoSQL databases do not provide $join$ operations [5]. This is one of the reasons for the slow diffusion of NoSQL databases with respect to SQL ones. Table II summarizes the comparison between SQL and NoSQL RDBMS. Instead, Table III shows the *top ten* related to the currently most adopted databases, according to *DB-Engines*[6], which collected information about 358 databases actually existing and used in computer systems; note that seven databases in the *top ten* are relational ones. *DB-Engines* assess the various databases according to the following parameters:

- Number of citations on websites also indented as the number of occurrences of a particular database through search engines, such as $Google$ and $Bing$
- General interest by $Google\ Trends$, which analyses how often a system is searched
- Number of job offers, which mention a particular database as a skill, for example through the most common job search engines, such as $Indeed$ and $Simply\ Hired$
- Number of profiles in professional networks, such as $LinkedIn$, where the database is mentioned
- Relevance in social networks, in particular concerning the number of $tweets$ on $Twitter$, which mention a database.

As emerges from Table III, the *top ten* includes three of the most important NoSQL databases, which will also be treated in this paper: MongoDB, Redis, and Cassandra.

[6]DB-Engines, https://db-engines.com/en/ranking

IV. SECURITY AND PRIVACY REQUIREMENTS

Security and privacy represent critical requirements in databases, in general, and for NoSQL databases, in particular, because non-relational databases are frequently based on $sharding$ (i.e., data are distributed over multiple servers), as hinted in Section I. The majority of NoSQL databases adopt coarse-grained access control mechanisms, while some efforts have been made in literature to provide fine-grained access control extensions to be applied to specific non-relational databases [34]. In the following, three relevant features will be detailed: (i) authentication, authorization and access control; (ii) privacy and policy enforcement; (iii) integrity and confidentiality.

A. Authentication, authorization and access control

Authentication mechanisms include all the methods adopted for recognizing the identity of a generic entity, in order to allow or deny access to a certain system, resource, or set of resources. For example, once an entity is authenticated to a service, the information to be disclosed may depend on specific access control rules, which could prevent access to certain kinds of sensitive or non-authorized data. In databases' context, users' access control is the first step to protect data [35].

Access control policies are classified into three well-known categories, originally conceived for RDBMS [10]: *Discretionary Access Control (DAC)*, *Mandatory Access Control (MAC)* and *Role Based Access Control (RBAC)*. DAC policies grant access based on the requester's identity; the resource's owner establishes access rights to the resource itself for all system users. DAC is extremely flexible and adaptable to various application environments, but it provides no control mechanism on the information flow; in fact, a user, owning an access authorization, can transfer such an authorization (also indirectly) to other users. MAC policies are an evolution of DAC ones, as they are able to handle the information flow, since, access is regulated by defining security classes (i.e., labels), which are associated with users and resources. More in detail, a level of confidentiality is associated with each user and resource, and all the actions, undertaken on data, are regulated by matching such levels with the predefined labels. In MAC, only the administrator can grant access to users; instead, DAC is easier both to use and to manage, since it adopts the principle of minimum privileges [36].

TABLE III. DATABASES' RANKING BY *DB-Engines*

Rank 2019	Rank 2020	Database	Model	Rating 2019	Rating 2020
1	1	Oracle	SQL	1369.36	+22.71
2	2	MySQL	SQL	1264.25	-14.83
3	3	Microsoft SQL Server	SQL	1062.76	-22.30
4	4	PostgreSQL	SQL	542.29	+60.04
5	5	MongoDB	NoSQL	446.48	+36.42
6	6	IBM Db2	SQL	161.24	-10.32
7	8	Redis	NoSQL	1151.86	+9.95
8	7	Elasticsearch	NoSQL	150.50	+1.23
9	11	SQLite	SQL	126.68	+3.31
10	10	Cassandra	NoSQL	119.18	-4.22

$RBAC$ policies introduce the concept of $role$, which simplifies the authorizations' management, since access permissions are not regulated $per - user$, but based on the roles associated to the user himself/herself [37]. Several studies propose various methodologies to support $RBAC$ implementation in NoSQL databases; for example, [38] proposes a purpose-based access control method for MongoDB, [39] suggests to adopt a *K-VAC (Key-Value Access Control)* access control system targeted to key-value stores, [40] and [41] devise an encryption method, which is targeted to column-oriented and graph-based databases. The common goal is to design a solution that ensures the completeness and correctness of data, in a system where users have limited access to the resources, needed to perform defined actions. According to the literature, there is no available solution able to guarantee data completeness and correctness for all the four types of NoSQL databases (i.e., key-value, column-oriented, document and graph); therefore, no results are available to compare $RBAC$ implementation with respect to NoSQL access control [42].

Besides such three conventional models (i.e., DAC, MAC and $RBAC$), other access control systems have been proposed to meet security requirements in NoSQL databases. To this end, *Fine-Grained Access Control (FGCA)* is considered a fundamental requirement in various application scenarios, but, as mentioned above, many NoSQL databases do not support $FGCA$. In column-based databases, [39] proposes to implement access control policies at various levels, such as based on columns' families; in this case, $FGCA$ is designed to work with Cassandra, so it represents a dedicated implementation, which cannot be easily adapted to other databases; while the solution proposed in [43] is assessed on MongoDB.

In *Big Data* context, it emerges the need to have an access control model based on the semantic content of data, named *CBAC (Content Based Access Control)*, where the concept of $content$ plays a fundamental role in the decision-making process for access control. The decision dynamically depends on the similarity between the user's credentials and the data content [44]. However, in this direction, the most suitable access control method is *ABAC (Attribute Based Access Control)*. Such a mechanism allows data owners to encrypt information based on access criteria, so that only the user who owns the permission to access the data can decrypt them. An issue related to such a kind of approach concerns the policies' update: when the data owner wants to change the policy, data must be transferred from the cloud to the local user's device and re-encrypted, according to the newly adopted policy, thus causing a high communication overload. To cope with such a problem, [45] proposes a method for safe policy updating, which consists in outsourcing the policy updating to the server. Another solution is represented by the adoption of $Hadoop$ [46], which is an open-source framework for *Big Data* storage and processing; it relies on *HDFS (Hadoop Distributed File System)* to store data across multiple nodes. $HDFS$ does not have a solid security model and users can directly access data without any authorization. Hence, to solve such an issue, the authors of [8] propose a new access control scheme for data storage in $Hadoop$, which is based on access tokens, in order to simplify the control over resources

by data owners. Finally, [37] proposes a scheme to optimize ABAC in NoSQL databases (more specifically, in MongoDB), where ABAC policies allow specifying context-sensitive conditions that regulate access in a fine-grained fashion. Essentially, the envisioned technique was originally based on the rewriting of queries through the use of SQL, which could appear unsuitable for processing heterogeneous data; hence, the generality of the proposal is limited. To deal with such an issue, [37] exploits SQL++ to be more general, so that it can be used by NoSQL data stores, which support SQL++. Finally, the work in [47] tries to couple ABAC with a new declarative language for fine-grained, attribute-based authorization policies, named *XACML (eXtensible Access Control Markup Language)* for Graph-structured data ($XACML4G$). The motivation lies in the additional path-specific constraints, described in graph patterns; in fact, with respect to other data models, they demand specialized processing of the rules and policies as well as adapted enforcement and decision-making mechanisms in the access control process.

Outcomes. What emerged is that authentication and authorization are widely explored concepts in databases' context. Different paradigms have been studied and put in act in literature, in order to achieve a finer-grained access control over the database's resources. A promising solution is represented by ABAC, since it couples access permissions with encryption, thus adding a further intrinsic security level on data. The main issue is related to the distributed nature of NoSQL database, which could bring to performance degradation caused by the execution of security functionalities. Another crucial aspect regards how access control policies are managed and updated, mainly due to the dynamic nature of IoT environments, where NoSQL databases are adopted. In fact, a policy update must be extended to the whole system for being effective. Surely, some synchronization mechanisms should be integrated to ensure that the same rules are effectively applied throughout the whole system [48].

B. Privacy and policy enforcement

Privacy is intended as the preservation of users' sensitive information, which can be derived from habits, profiling, tracing, or inferred from other information, such as location or services' preferences, which could be stored in databases, besides being transmitted throughout the network [49]. Note that, privacy management has become more complex in systems based on NoSQL databases, which usually run on distributed systems, dealing with heterogeneous data coming from untrusted IoT environments. Some application domains are more affected by the presence of sensitive data, such as health monitoring and services, traffic analysis, smart retail, and so on.

Policy enforcement stands for the methods responsible for actuating well-defined security and privacy policies on data in a particular system. Policy enforcement is often represented by a module or a framework included in the system itself, which should be able to filter the users' requests and enable only the disclosure of authorized resources.

Multiple violations can be undertaken in the databases' context, such as unauthorized access attempts, digital identity theft or attacks on software systems, to hit provider companies, thus damaging users' privacy. To cope with such kinds of attacks, security strategies can be set up through the adoption of privacy policies and policy enforcement mechanisms [50], besides introducing proper encryption schemas to prevent information leakage. Privacy policies can be designed to protect privacy on data and can be specified by data owners, data providers or security administrators, depending on the applications.

The solution defined in [51] couples a policy enforcement framework with an encryption scheme based on privacy homomorphism; while [52], describes a mechanism to apply security-aware policies on the attributes stored in JSON documents, which allow expressing attribute-based policies, thus building an access control mechanism to enforce JSON security policies. Moreover, the envisioned approach also tries to handle privacy policy conflicts, by defining a priority hierarchy among the policy structure. Similarly, the work in [53] envisions a descriptive language, using JSON notation, which enables users to generate a *security plan* application for performing secure query processing over encrypted NoSQL databases, hosted in the public cloud; such a system is targeted to NoSQL stores in general, but has been assessed in a document-store data model. Note that its essential feature resides in policy enforcement's multi-key and multi-level security mechanisms because the encryption key is subject to more frequent changes than the crypto module. Furthermore, keys are assigned for a single data element, while encryption algorithms could be applied for several data elements with several keys. Hence, such a separation allows for more efficient enforcement of security policy and key

management.

Apart from JSON, other solutions [54] [55] exploit XACML for policy specification in NoSQL databases; more in detail, [55] focuses on graph stores.

Outcomes. With respect to privacy, a relevant aspect, which emerged from the above discussion, is related to the choice of the language used for policy definition. The two most adopted solutions are JSON and XACML, which surely provide adequate support for specifying the desired rules. However, a standard has not been defined yet, and many solutions are specifically tailored to a certain datastore or data model. Aspects, which still deserve more attention, are those related to: (i) key management, since encryption strictly affects the possible application of privacy strategies; (ii) identity management, due to the fact that, within the system, proper entities must be responsible for performing the policy enforcement controls. The introduction of privacy-related functionalities often requires the presence of trusted authorities or trusted third parties, which must ensure that the system behaves as expected with respect to resources' release; however, the presence of trusted authorities or trusted third parties could potentially bring single points of failure. To cope with such an issue, a distributed approach should be envisioned, maybe adopting an architecture not only based on the cloud, but integrated with a fog computing architecture [56]. Furthermore, privacy is often based on purposes, but other parameters, like policy revocation or consent options, could be taken into consideration.

C. Integrity and confidentiality

Integrity and confidentiality requirements are guaranteed if the content and the ownership of the data cannot be tampered with or eavesdropped on by a non-legitimate entity. Encryption techniques are usually adopted to deal with such issues.

The authors of [57] propose an *OPE (Order Preserving Encryption)* scheme, which is able to create database's indexes directly and make many kinds of queries on the ciphertext (i.e., without decrypting information), thus contributing to improving the resilience of information against malicious attacks. Similarly, [58] and [59] adopt homomorphic cryptography to also execute aggregations queries, such as the one including SUM and AVG operators. The work presented in [60] uses *REA (Reverse Encryption Algorithm)*, which is based on symmetric encryption, to protect data confidentiality, ensuring, at the same time, query processing performance. A searchable system for encrypted Personal Health Record (PHR), residing on a MongoDB database in the cloud, is envisioned in [61]; it adopts an *Adelson-Velsky Landis (AVL)* tree to construct an index on the 'age' field, and the *Order-Revealing Encryption (ORE)* algorithm to encrypt the AVL tree.

In the context of graph databases, a hashing technique, as presented in [62], is used to encrypt data organized within graphs. The nodes' tree is built through an in-depth search in the graph, after which the pre-order and post-order values are assigned to each visited node. Finally, a collision-resistant hash value is generated for the resulting structure; however, such a proposal does not support graph's dynamics, hence the hash must be re-calculated for the entire structure in case of any changes, thus compromising performance.

As just said, NoSQL databases are widely adopted in cloud environments, due to their horizontal scaling feature. In such a context, *DBaaS (DataBases-as-a-Service)* represents a cloud computing service model that provides users access to a database, without the need to configure physical hardware or install particular software. Three entities belong to a generic DBaaS model: (i) *Data Owner (DO)*, who uploads data to the cloud; (ii) *Cloud Provider (CP)*, which represents a third-party company that provides cloud-based storage services, applications, infrastructure or platform; (iii) *Clients*, who fetch data from the cloud. Usually, the NoSQL approach, adopted in cloud systems, is the column-oriented one. Data integrity focuses on two dimensions: correctness and completeness. The correctness identifies a correct data $insert$ operation in the cloud by the clients; while the completeness identifies a correct data reception upon client requests. Another dimension is related to freshness, which means that clients always get the latest version of data. The work in [63] identifies three approaches to data integrity and confidentiality:

- *Merkle Hash Tree (MHT).* MHT is a binary tree, where each leaf is a hash of a data block, and each internal node is a hash of the concatenation of its two children. In the databases' context, the leaves are records of the table, ordered according to a search key [64]. Such a structure should reduce the cost of I/O operations, both on the client and server sides. The work in [65] introduces a method exploiting MHT, which is used as a structure to guarantee data integrity in DBaaS model.

- *Digital Signature (DS).* DS approach is based on digital signature also to guarantee authenticity to data, and it is adopted in [66]. More in detail, each database table is associated with a new column, which contains a hash of concatenated record values, as it is signed with the private key of the DO; then, clients verify the integrity of records using the public key of the DO. The signature aggregation technique is adopted to combine multiple record signatures into a single one to reduce client-server communication's costs. To ensure completeness, DO signs consecutive pairs of records, hence it must have a local copy of the database or it must be able to gather consecutive records from the cloud.
- *Probabilistic approach*. In probabilistic approach, additional records are uploaded to the cloud along with the original ones. The more additional records are being uploaded, the higher is the probability of detecting data integrity attacks. All data is encrypted on the client-side, hence CP cannot distinguish between the original and the additional records. In this case, the whole database must be necessarily encrypted.

Outcomes. Concerning cryptographic algorithms, the literature seems to be still divergent on the approach to be adopted in NoSQL databases. In fact, a variety of solutions have been proposed, which strictly depend on the system's architecture. Probably, the well-suited approach is the one that supports the resources' constraints of end-devices in IoT scenarios [67], which cannot perform heavy encryption tasks. Another important aspect of being considered is the delay, when the database is distributed (or sharded) within the IoT network, as in fog computing architectures [68]. A trade-off must be established towards an efficient and standardized solution in this direction.

Table IV summarizes the outcomes, in terms of pros and cons, of the investigated security and privacy requirements, as presented in this section.

V. NoSQL databases: comparison

We provide herein a detailed discussion on available solutions concerning the security requirements described in Section IV, specifically tailored to four well-known NoSQL databases: Redis, MongoDB, Cassandra, and Neo4j. As just introduced in Section III, they belong to the four different data models, conceived in NoSQL field.

A. Redis security requirements

Since Redis is a key-value database, any malicious entity can obtain information about the keys and gather the corresponding values. Redis is essentially designed to be accessed by trusted clients inside trusted environments, hence it provides limited built-in security functionalities[7]. In fact, even if Redis does not implement any access control mechanism, it provides a tiny layer of authentication that is optionally turned on editing the $redis.conf$ file. When the authorization layer is enabled, Redis will refuse any query by unauthenticated clients, and a client can authenticate himself/herself by sending the $AUTH$ command followed by the password. Such a password is set by the system administrator in clear text inside the $redis.conf$ file and should be long enough to prevent brute force attacks. Concerning server's configuration, by means of $CONFIG$ command, a client can change the program's working directory and files' name; to cope with such an issue, the Redis authors are still investigating the possibility of adding a new configuration parameter to avoid $CONFIG\ SET/GET$ operations.

The authors in [69] propose the adoption of a Redis client, which provides authentication, authorization and *AES (Advanced Encryption Standard)* encryption facilities for both text data and binary/multimedia ones. As just said, by default, Redis cannot be accessible to everyone, but only to trusted clients belonging to the network; hence, queries from unauthenticated clients will be rejected. Any user wishing to access a Redis system must first register; at the login stage, the generated key, on the client-side, is encrypted before being stored in the databases, thus protecting the information travelling throughout the network. The mechanism proposed in [69] exploits the AES algorithm for eliminating the need for any key exchange protocol; it also extends the Redis system's capability to manage scalable and secure databases in real-time applications.

Future improvements will aim to introduce new commands to access data and to reduce the computational overhead, by limiting the resources' management overload, since data must be encrypted and decrypted at each operation; moreover, considerations on the needed bandwidth are certainly required, in case AES algorithm increases the data length.

[7]Redis Security, https://redis.io/topics/security

TABLE IV. SUMMARY OF SECURITY AND PRIVACY REQUIREMENTS

Security features	Pros	Cons
Authentication, Authorization, Access Control	many mechanisms available in literature	policy updates in distributed contexts
Privacy, Policy Enforcement	support by the language for policy definition	lack of key and identity management strategies
Integrity, Confidentiality	many cryptographic algorithms available	delay in distributed systems and power-constraints of end-devices

Outcomes. As emerged from the analysis mentioned above, Redis is not conceived to run in untrusted environments. Such an aspect represents a limitation mainly in IoT scenarios, where devices taking part in the communication exchanges, within the network, are usually untrusted. Hence, its real applicability for the IoT, besides the excellent performance in terms of query response time and data processing, can be hindered by the lack of security-related functionalities, which are essential to trust and monitor the IoT network's behavior.

B. Cassandra security requirements

Natively, Cassandra stores data without encryption [70] and CQL language could be potentially exposed to injections attacks, as SQL. To deal with injection, Thrift APIs, which exploit *RPC (Remote Procedure Call)*, have been introduced in Cassandra. Another issue is related to the DoS (Denial of Service) attack, because Cassandra runs a thread for each client; hence, if several malicious users create a certain number of malicious connections, they could make Cassandra waste many resources. Moreover, the current implementation does not time out idle connections, so any open connection consumes a thread.

Concerning data confidentiality and authentication, from version 1.2.2, data are protected through access control and encryption; a user can be authenticated using a proper interface, named $IAuthenticate$, as described in Cassandra's documentation[8], following such steps: (i) change the setting of $cassandra.yaml$ file as the authenticator in $PasswordAuthenticator$ and set the authorization to $CassandraAuthorizer$; (ii) configure the replication factor for the key space.

Before enabling user's authentication in the cluster, client applications must be pre-configured with credentials provided by Cassandra. As soon as a server enables authentication, any connection attempts without proper credentials will be rejected. Note that super-users can set up user accounts to be authorized to access database objects using CQL to grant them proper permissions. More in detail, three kinds of authentication are available:

- *Internal authentication*: the column-oriented database engine manages users' access through credentials such as user's ID and password
- *External authentication*: external authentication, such as $Kerberos$ authentication, consists in another proper network protocol in charge to authenticate the user's identity using a ticketing system, where tickets are used to demonstrate users' identity itself
- *Client-Server Encryption*: Client-to-server or node-to-node encryption, coupled with a trusted certificate, should guarantee the reliability of user's information.

Besides authentication, users can be authorized to access certain schemes, tables and columns, depending on their roles and privileges. To define such permissions, the database's administrator must configure the $cassandra.yaml$ file, while to grant permissions to users, the system_auth space must be configured. Cassandra provides an authority interface, named $IAuthority$, which provides methods for discovering the authorizations associated with an authenticated user and a hierarchical list of resources. Note that, each authorization is only specified for existing column families, so no protection is available for newly added columns or column families. In fact, after permissions are granted, the database does not update them; then, the Cassandra process must be restarted.

As hinted in Section IV-A, [39] proposes a $FGCA$ access control policies at column families' level. Instead, the work in [71] point out strengths and weaknesses, quantifying the performance, of three algorithms for searchable encryption in Cassandra, namely the *index-per-keyword based CGK* scheme, the *index-per-document based HK* scheme and the *sequential scan based SWP* scheme.

[8]Cassandra's documentation, https://cassandra.apache.org/doc/latest/

Cassandra does not natively support the auditing task, which is usually fundamental for organizations and companies to monitor, detect anomalies, and make some statistics on the database's activities, as well as to ensure and verify compliance with the applied security policies. To cope with such a lack, the work in [72] proposes to adopt a custom implementation of both $IAuthority$, in order to provide a complete audit of all operations that require authorization, and $IAuthenticate$, in order to audit successful and unsuccessful login attempts.

Outcomes. Summarizing, Cassandra also provides excellent performance in terms of data retrieval and query processing. It supports basic security configurations, ranging from client-to-server or node-to-node encryption, to a simple internal or external authentication. However, more complex mechanisms could be integrated to offer modules, enabling $FGCA$ access control, more robust encryption schemes, a clever key management system, etc. Such features would make Cassandra more suitable for integration in IoT distributed environments, since it just supports replication and partitioning.

C. MongoDB security requirements

As Cassandra, MongoDB also does not encrypt data within the database, but applications are in charge of ciphering information before it is inserted into the store. Note that MongoDB cannot perform encryption at the column level, since documents could be different from each other. To cope with such an emerged issue, the authors of [73] propose the adoption of various symmetric key cryptographic schemes to encrypt MongoDB data at the application level, but do not define the respective keys' size. Instead, the work in [74] provides an analysis of DES-64, AES-128 and Blowfish-64 symmetric cryptographic algorithms simulating different sizes for the randomly generated keys; since encrypted data could require more storage space, a $ZLib$ compression technique[9] could be integrated. The conducted security analysis shows that AES is, naturally, more robust, due to key length. Note that MongoDB is usually configured to cipher documents' fields along with the identifier of the JSON object, and both information is used for authentication to prevent malicious data replication. MongoDB supports the following authentication mechanisms to identify the user:

- *SCRAM (Salted Challenge Response Authentication Mechanism)*: it is the default authentication mechanism for MongoDB, which substantially verifies user's credentials
- *X.509 certificates*: such a mechanism uses X.509 certificates, which are issued by a certification authority

Authentication is made available in two versions, which are the *Community Edition* and the *Enterprise Edition*. The second one provides such further functionalities, with respect to the first one: (i) in-memory storage, which guarantees more predictable latency; (ii) auditing; (iii) $Kerberos$ authentication; (iv) *LDAP Proxy Authentication* and *LDAP Authorization*; (v) *Encryption at rest*, which represents encryption on inactive data; it could be combined with transport level encryption and security policies to guarantee compliance with security standards.

MongoDB integrates the RBAC mechanism to manage authorization, adding support for policies' specifications at the document's level. Since JavaScript functions, which are stored in $db.system.js$ collection, are used by developers to execute internal commands, external entities could inject malicious scripts [75]. A purpose-based access control method for MongoDB is proposed in [38], and the same authors presents: (i) in [37], a scheme to optimize ABAC, exploiting SQL++ language; (ii) in [76], a general approach to evaluate the impact of access control policies on the protected resources within NoSQL system. In particular, the experiments have been done with MongoDB, which has been chosen as it natively supports MapReduce[10], like many NoSQL datastores. The aim of [77] is similar to that of [37], since it proposes a method for incorporating ABAC in NoSQL databases by generating RBAC roles from ABAC policies, providing an implementation for MongoDB. Instead, the authors of [54] also envision an architecture applied to MongoDB for purpose-based access control policies, but policies are represented with XACML.

An encrypted MongoDB, named $CryptMDB$, is described in [78]; it integrates an additive homomorphic asymmetric cryptosystem to encrypt user's data and achieve strong privacy protection, confidentiality, and protection against illegal access to the database.

Outcomes. MongoDB is the most investigated and most famous among the NoSQL databases. For such a reason, and also due to its dynamic structure, it seems

[9]ZLib compression, https://zlib.net/

[10]MapReduce, https://hadoop.apache.org/

more flexible towards the integration with security mechanisms. In fact, it already supports integration with $Kerberos$ authentication, $LDAP$, and X.509 certificates, but it does not provide its own authentication protocol. Also, no standardization is available concerning authorization, access control, and encryption, besides it is adopted in many solutions, also tailored to IoT environment, again due to its documents' flexibility, and it is frequently coupled with policy enforcement frameworks [79].

D. Neo4j security requirements

Neo4j provides a complex security model, which is stored within the graph, in particular in a special database, named *system database* [11]. Neo4j also enables by default the following authentication/authorization providers: (i) a native authentication provider; (ii) an $LDAP$ authentication provider; (iii) an authentication provider integrated with a custom plug-in; (iv) $Kerberos$ and *Single Sign-On* authentication.

Moreover, three access control mechanisms are available:

- *RBAC*: Neo4j defines a set of predefined roles, which are the following:
  - $Reader$: the user enabled for this role can only read data, execute read-only queries, and change his/her password
  - $Editor$: the user enabled for this role can execute queries to modify data, create nodes and relationships, set properties and even delete data; however, it has no control over other users' activities
  - $Publisher$: the user enabled for this role has the same $Editor$'s privileges, but can also create labels, types of relationships and properties
  - $Architect$: the user enabled for this role is a $Publisher$, who can manage the database's indexes and constraints
  - $Admin$: the user enabled for this role is able to perform any operation within the database, such as managing transactions and queries, made by other users.

  In case a user owns more than one role, he/she automatically inherits the permissions from both roles [80].
- *Sub-graph access control*: It is regulated by privileges, which are managed by combining $white-list$ and $black-list$ mechanisms; $GRANT$ and $DENY$ operations are associated to read and write privileges, with respect to the membership of a user to a certain list[12].
- *Property-level access control*: In this case, role-based $black-lists$ can be used to limit the properties, which can be read by a user. In particular, the function $dbms.security.propertylevel.enabled$ enables property-level access control, while the function $dmbs.security.propertylevel.blacklist$ ensures that the properties included in the $black-lists$ for a particular user are the union of the $black-lists$ for all the roles owned by the user himself/herself.

To preserve the integrity and confidentiality of data when they travel throughout the network, Neo4j supports SSL/TLS technology[13], with the following features: (i) the SSL framework must be adequately configured to explicitly state which are the allowed encryption techniques; (ii) a certificate authority must be integrated within the system.

Neo4j can be optionally extended by writing custom code, which can be directly invoked by means of $Cypher$ language. Security to such extensions could be guaranteed through two particular concepts: (i) $sandboxing$, which allows isolating some parts of the system to prevent the use of unsafe APIs; (ii) $white-lists$, which, by means of the function $dbms.security.procedures.allowlist$, limiting some extensions' loading when a wider library is invoked.

Outcomes. Neo4j, as MongoDB, provides a flexible structure to store information, but it is more complex to manage, since proper strategies must be adopted to traverse the graph and gather the requested information. Concerning security, as for MongoDB, integration with $Kerberos$ and $LDAP$ for authentication is still supported; while encryption mechanisms are not covered by adequate studies. Furthermore, a simple $RBAC$ is provided, while more complex access control methods would be required to manage the heterogeneous and dynamic systems based on the IoT concept.

[11] Neo4j documentation on security, https://neo4j.com/docs/cyphermanual/current/administration/security/

[12] Graph and sub-graph access control, https://neo4j.com/docs/cypher-manual/current/administration/security/subgraph/#administration-security-subgraph

[13] SSL and Neo4j, https://neo4j.com/docs/operationsmanual/current/security/ssl-framework/

Table V summarizes the outcomes, in terms of pros and cons, of the investigated security and privacy requirements, as presented in this section.

VI. Discussion

The conducted analysis revealed how the interest in NoSQL databases already involves companies and organizations, due to the distributed nature and dynamic management provided by stores to handle large volumes of data. Some of them use keys, table structures, column families, thus giving the idea that some NoSQL databases can be superficially similar to relational ones, but without giving up their unstructured nature. In this sense, a growing trend is represented by the use of SQL query engines to access NoSQL databases, such as the open-source Apache Drill projects[14]. Such a trend could persuade more companies to choose hybrid SQL/NoSQL data storage solutions in the next future.

Concerning key-value models, managing data relationships and executing transactions with multiple operations is not recommended due to the non-standardized query language. The adoption of such a kind of store is recommended when dealing with e-commerce or for user profiles' and preferences' management, since such systems usually present a fixed structure, which is easily mapped in key-value pairs (e.g., a product in e-commerce is represented by product's name, price, quantity, etc.). As an evolution of the key-value model, the column-oriented one can be used for indexing, counters (e.g., the visitors of a website) or the deadlines' management (e.g., software licenses for the trial period). Instead, the document-oriented model is useful for logging events and web analytics (e.g., user profiling), due to its semi-structured nature; but it is not the best solution when you need to execute complex transactions involving numerous operations (due to the continuous indexing tasks performed). Finally, the use of the graph database model presents poor scalability due to the single server architecture, but it is ideal for obtaining query results in real-time.

As pointed out in the analysis conducted in Section V, security requirements in NoSQL databases still need to be developed and expanded, as well as the ability to define standardized query languages. Table VI summarizes the features belonging to the four NoSQL databases, analyzed in this paper: Redis, Cassandra, MongoDB, and Neo4j.

[14]Apache Drill, https://drill.apache.org

In general, what emerges from the analysis of the state-of-the-art is that many works, which are available in the literature, adopt NoSQL databases, but, instead of directly integrating them with security functionalities, they couple the same stores with security frameworks, in order to keep them reliable both in terms of *data at rest* and when information are queried from or to them. Some pillars must be considered in such a direction, as described in the following.

Data protection. Concerning data protection, a central role is played by encryption mechanisms. As just pointed out, some NoSQL databases do not support data encryption by themselves, but the application, which integrates with its architecture the store, must ensure data confidentiality and integrity. Note that encryption helps in guaranteeing privacy for users' sensitive information. Ciphering operations should take into account the key's dimension as well as the dimension of data after the encryption task, in order to keep under control the storage occupancy. Also, mechanisms which support end-to-end encryption are preferable, mainly in IoT-related and fog scenarios, where information is stored in a distributed way and may be transmitted in hop-by-hop communications [81] [82]. Note that, even in the fog computing field, the research community is investigating new solutions for searching over encrypted data [83] [84]. Another fundamental issue is distributing and protecting the credentials owned by the involved devices and recovering the system in case of violation.

Resource disclosure. In databases' context, it is fundamental to regulate how information is released to requesting users. Hence, access control and authentication mechanisms must be mandatory. Some promising approaches have been cited and could represent relevant starting points for further investigations in the area, such as those based on attributes, which reveal to be the most compliant with the dynamic IoT environment. Traditional public key infrastructure-based authentication schemes may provide the system with identity authentication and conditional privacy protection, which are not enough for assessing the reliability of the information. Hence, access control mechanisms based on attributed may be coupled with the so-called *Attribute Based Encryption (ABE)* [85] [86]. In fact, ABE allows encrypting data for multiple recipients, in such a way that only those recipients whose attributes satisfy a given access policy can decrypt afterward. A distributed architecture, resulting from the combination of a NoSQL database and fog computing, is an ideal

TABLE V. SUMMARY OF SECURITY AND PRIVACY REQUIREMENTS IN THE INVESTIGATED NOSQL DATABASES

NoSQL database	Considerations about security and privacy features
Redis	Unsuitable for running in untrusted environments; a limited support for authentication is provided
Cassandra	Authentication, access control and encryption mechanisms are available
MongoDB	Kerberos authentication, LDAP, X.509 certificates, authorization and encryption mechanisms are available
Neo4j	Kerberos and Single Sing-On authentication, LDAP, RBAC with predefined roles, authorization and SSL/TLS mechanisms are available

TABLE VI. SECURITY FEATURES OF NOSQL DATABASES

Database	Encryption	Authentication/Authorization	Auditing	Query Language
Redis	not supported	username/password credentials	not supported	not available
Cassandra	only in the *Enterprise Edition*	username/password credentials	not supported	CQL
MongoDB	only with SSL/TLS	$Kerberos$ & $LDAP$ in *Enterprise Edition*	only in the *Enterprise Edition*	MQL
Neo4j	only in the *Enterprise Edition* with SSL/TLS	$Kerberos$ & $LDAP$ in *Enterprise Edition*	only in the *Enterprise Edition*	Gremlin, Cypher

candidate to actuate proper measures to grant access tokens to authorized parties, who use them to perform given actions (e.g., data decryption). Also, a fog computing platform can be used as a sort of distributed storage and trust authority to authorize access and disclose data among authorized parties and end-devices. The challenge is how to practically couple the fog computing paradigm and distributed functionalities with the NoSQL store's primitives.

Trust and rogue device detection. The concept of distributed storage makes the end-devices vulnerable to remote and physical attacks, since they are far from the core system and communications take place over an untrusted IoT network. As emerges from Sections IV and V, many NoSQL databases provide mechanisms for data replication, which could be in contrast (i.e., mainly in terms of efficiency) with security-related approaches, but which improve the robustness of the system in the case of data leakage. In general, users should trust the system from which they gather the information of interest; hence, proper mechanisms should guarantee the data reliability and a prompt recovery in case of violations [87].

Auditing. Monitoring, logging, and reporting are essential for detecting attacks and violations in a system promptly. In the case of databases, an audit about the access to resources can be methodically performed, in order to detect anomalies, and make some statistics on the database's activities [88]. Also, the attack's propagation should be inhibited. In this way, the correct application of security policies can be effectively ensured, since the behaviour of the system is continuously verified.

Performance and low-overhead requirements. One of the drawbacks of integrating security-related functionalities in any system is that they will undoubtedly have an impact on performance. Mainly in IoT scenarios, which usually collect and transmit information in real-time, it is fundamental to guarantee a low overhead to reduce network latency. NoSQL databases have been conceived for being very efficient in terms of data retrieval, depending on the specific application and data model; hence, such features must be preserved even in the presence of security checks.

Standardization. A standardization process for NoSQL databases and query languages is definitely needed [89]. Along with security integrations, also policy definition must be specified, in order to represent access permissions in a uniform way, also enabling, in the future, the reliability of data among different datastores, which belong and communicate across multiple domains, as sometimes happens in IoT scenarios.

Finally, Tables VII and VIII resemble the last discussed aspects to be taken into considerations in the definition of solutions targeted to NoSQL databases.

VII. CONCLUSION

The paper started with an analysis of NoSQL data models; key-value, column-oriented, document, and graph. For each data model, technical details for the corresponding datastore have been provided, specifically:

TABLE VII. FEATURES AND OPEN ISSUES OF NOSQL DATABASES - PART 1

Database/Outcomes	Data protection	Resource disclosure
Redis	not guaranteed	limited support
Cassandra	encryption mechanisms integrated	access control mechanism integrated
MongoDB	encryption mechanisms integrated	access control mechanism integrated
Neo4j	SSL/TLS integrated	access control mechanism integrated
Open issues	definition of efficient searching mechanisms over encrypted data distributed management of credentials, revocation systems	integration with fog computing scenarios

TABLE VIII. FEATURES AND OPEN ISSUES OF NOSQL DATABASES - PART 2

Database/Outcomes	Trust and rogue device detection	Performance and low-overhead	Standardization
Redis	default asynchronous replication	efficient access to data	elementary operations
Cassandra	single and multiple data centres replication	efficient access to data	simple APIs
MongoDB	replica sets	heavy authentication/encryption mechanism integrated	RBAC integrated
Neo4j	only standard replication mechanisms supported	complex security model	RBAC integrated
Open issues	definition of intrusion detection systems in presence of replication, policy enforcement's monitoring	save devices' resources by limiting the overhead, reduce latency	definition of standard query languages, security interfaces, policy mechanisms

Redis, Cassandra, MongoDB, and Neo4j. Then, a comparison between NoSQL and SQL RDBMS features has been carried out. Furthermore, the paper analyzed how security requirements (i.e., authentication, authorization, access control, privacy, policy enforcement, integrity, and confidentiality) are addressed in NoSQL databases, considering the different data models mentioned. What emerged is that some NoSQL datastores already provide some authentication, authorization, access control and encryption mechanisms, but some crucial aspects still deserve more attention by the scientific community, such as: the definition of efficient and, possibly, standardized security mechanisms, query and policy languages. Such solutions aim to improve the reliability and robustness of the whole system, mainly in the case of IoT scenarios, characterized by high heterogeneity in terms of involved devices and managed information. Moreover, in a distributed environment, it is fundamental to guarantee that it operates in compliance with security requirements, in a capillary way (e.g., in fog computing contexts); in such a direction, NoSQL databases represent a promising solution, due to their native distributed nature. We hope that the discussion can be of interest to various audiences, including PhD candidates, research consortia and IT industry, to pursue the realization of secure and privacy-aware solutions targeted to NoSQL-based infrastructures NoSQL-related query languages.

REFERENCES

[1] A. Kobusińska, C. Leung, C.-H. Hsu, R. S., and V. Chang, "Emerging trends, issues and challenges in internet of things, big data and cloud computing," *Future Generation Computer Systems*, vol. 87, pp. 416–419, 2018.

[2] V. N. Gudivada, D. Rao, and V. V. Raghavan, "Nosql systems for big data management," in *2014 IEEE World congress on services*. IEEE, 2014, pp. 190–197.

[3] R. Cattell, "Scalable sql and nosql data stores," *Acm Sigmod Record*, vol. 39, no. 4, pp. 12–27, 2011.

[4] A. Tewari and B. Gupta, "Security, privacy and trust of different layers in internet-of-things (iots) framework," *Future generation computer systems*, vol. 108, pp. 909–920, 2020.

[5] F. Gessert, W. Wingerath, S. Friedrich, and N. Ritter, "Nosql database systems: a survey and decision guidance," *Computer Science-Research and Development*, vol. 32, no. 3, pp. 353–365, 2017.

[6] A. Moniruzzaman and S. A. Hossain, "Nosql database: New era of databases for big data analytics-classification, characteristics and comparison," *International Journal of Database Theory and Application*, vol. 6, no. 4, 2013.

[7] A. Davoudian, L. Chen, and M. Liu, "A survey on nosql stores," *ACM Computing Surveys (CSUR)*, vol. 51, no. 2, pp. 1–43, 2018.

[8] E. Sahafizadeh and M. A. Nematbakhsh, "A survey on security issues in big data and nosql," *Advances in Computer Science: an International Journal*, vol. 4, no. 4, pp. 68–72, 2015.

[9] A. A. Alotaibi, R. M. Alotaibi, and N. Hamza, "Access control models in nosql databases: An overview," *JKAU*, vol. 8, no. 1, pp. 1–9, 2019.

[10] A. Zahid, R. Masood, and M. A. Shibli, "Security of sharded nosql databases: A comparative analysis," in *2014 conference on information assurance and cyber security (CIACS)*. IEEE, 2014, pp. 1–8.

[11] W. Zugaj and A. S. Beichler, "Analysis of standard security features for selected nosql systems," *American Journal of Information Science and Technology*, vol. 3, no. 2, pp. 41–49, 2019.

[12] A. Rafique, D. Van Landuyt, E. H. Beni, B. Lagaisse, and W. Joosen, "Cryptdice: Distributed data protection system for secure cloud data storage and computation," *Information Systems*, vol. 96, p. 101671, 2021.

[13] S. Liu, S. Nguyen, J. Ganhotra, M. R. Rahman, I. Gupta, and J. Meseguer, "Quantitative analysis of consistency in nosql key-value stores," in *International Conference on Quantitative Evaluation of Systems*. Springer, 2015, pp. 228–243.

[14] F. Bugiotti and L. Cabibbo, "A comparison of data models and apis of nosql datastores." in *SEBD*, 2013, pp. 63–74.

[15] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, "Column-oriented database systems," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1664–1665, 2009.

[16] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[17] A. Chebotko, A. Kashlev, and S. Lu, "A big data modeling methodology for apache cassandra," in *2015 IEEE International Congress on Big Data*. IEEE, 2015, pp. 238–245.

[18] H. Vera, W. Boaventura, M. Holanda, V. Guimaraes, and F. Hondo, "Data modeling for nosql document-oriented databases," in *CEUR Workshop Proceedings*, vol. 1478, 2015, pp. 129–135.

[19] S. Chickerur, A. Goudar, and A. Kinnerkar, "Comparison of relational database with document-oriented database (mongodb) for big data applications," in *2015 8th International Conference on Advanced Software Engineering & Its Applications (ASEA)*. IEEE, 2015, pp. 41–47.

[20] S. G. Edward and N. Sabharwal, "Mongodb architecture," in *Practical MongoDB*. Springer, 2015, pp. 95–157.

[21] A. Celesti, M. Fazio, and M. Villari, "A study on join operations in mongodb preserving collections data models for future internet applications," *Future Internet*, vol. 11, no. 4, p. 83, 2019.

[22] P. Barceló Baeza, "Querying graph databases," in *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, 2013, pp. 175–188.

[23] F. Holzschuher and R. Peinl, "Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j," in *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, 2013, pp. 195–204.

[24] J. Guia, V. G. Soares, and J. Bernardino, "Graph databases: Neo4j analysis." in *ICEIS (1)*, 2017, pp. 351–356.

[25] H. Huang and Z. Dong, "Research on architecture and query performance based on distributed graph database neo4j," in *2013 3rd International Conference on Consumer Electronics, Communications and Networks*. IEEE, 2013, pp. 533–536.

[26] S. Jouili and V. Vansteenberghe, "An empirical comparison of graph databases," in *2013 International Conference on Social Computing*. IEEE, 2013, pp. 708–715.

[27] J. J. Miller, "Graph database applications and concepts with neo4j," in *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, vol. 2324, no. 36, 2013.

[28] K. Sahatqija, J. Ajdari, X. Zenuni, B. Raufi, and 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.

[29] H. Fatima and K. Wasnik, "Comparison of sql, nosql and newsql databases for internet of things," in *2016 IEEE Bombay Section Symposium (IBSS)*. IEEE, 2016, pp. 1–6.

[30] Y. Gu, X. Wang, S. Shen, S. Ji, and J. Wang, "Analysis of data replication mechanism in nosql database mongodb," in *2015 IEEE International Conference on Consumer Electronics-Taiwan*. IEEE, 2015, pp. 66–67.

[31] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan, "Salt: Combining {ACID} and {BASE} in a distributed database," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 495–509.

[32] V. Abramova, J. Bernardino, and P. Furtado, "Which nosql database? a performance overview," *Open Journal of Databases (OJDB)*, vol. 1, no. 2, pp. 17–24, 2014.

[33] E. Brewer, "Cap twelve years later: How the" rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[34] P. Colombo and E. Ferrari, "Fine-grained access control within nosql document-oriented datastores," *Data Science and Engineering*, vol. 1, no. 3, pp. 127–138, 2016.

[35] N. Gupta and R. Agrawal, "Nosql security," in *Advances in Computers*. Elsevier, 2018, vol. 109, pp. 101–132.

[36] F. Jaidi, "Advanced access control to information systems: Requirements, compliance and future directives," *SECURITY IN COMPUTING AND COMMUNICATIONS*, p. 83, 2017.

[37] P. Colombo and E. Ferrari, "Enhancing nosql datastores with fine-grained context-aware access control: A preliminary study on mongodb," *International Journal of Cloud Computing*, vol. 6, no. 4, pp. 292–305, 2017.

[38] ——, "Enhancing mongodb with purpose-based access control," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 6, pp. 591–604, 2015.

[39] D. Kulkarni, "A fine-grained access control model for key-value systems," in *Proceedings of the third ACM conference on Data and application security and privacy*, 2013, pp. 161–164.

[40] Y. Shalabi and E. Gudes, "Cryptographically enforced role-based access control for nosql distributed databases," in *IFIP*

Annual Conference on Data and Applications Security and Privacy. Springer, 2017, pp. 3–19.

[41] C. Morgado, G. B. Baioco, T. Basso, and R. Moraes, "A security model for access control in graph-oriented databases," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2018, pp. 135–142.

[42] H. Khan, "Analysis of role-based access control methods in nosql databases," Ph.D. dissertation, Middle Tennessee State University, 2019.

[43] H. X. Son and E. Chen, "Towards a fine-grained access control mechanism for privacy protection and policy conflict resolution," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 2, 2019.

[44] W. Zeng, Y. Yang, and B. Luo, "Access control for big data using data content," in *2013 IEEE International Conference on Big Data*. IEEE, 2013, pp. 45–47.

[45] K. Yang, X. Jia, and K. Ren, "Secure and verifiable policy update outsourcing for big data access control in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 12, pp. 3461–3470, 2014.

[46] P. Adluru, S. S. Datla, and X. Zhang, "Hadoop eco system for big data security and privacy," in *2015 Long Island Systems, Applications and Technology*. IEEE, 2015, pp. 1–6.

[47] A. Mohamed, D. Auer, D. Hofer, and J. Küng, "Authorization policy extension for graph databases," in *International Conference on Future Data and Security Engineering*. Springer, 2020, pp. 47–66.

[48] S. Sicari, A. Rizzardi, D. Miorandi, and A. Coen-Porisini, "Dynamic policies in internet of things: enforcement and synchronization," *IEEE Internet of Things Journal*, vol. 4, no. 6, pp. 2228–2238, 2017.

[49] A. F. Westin, "Privacy and freedom," *Washington and Lee Law Review*, vol. 25, no. 1, p. 166, 1968.

[50] S. Tamane, V. K. Solanki, and N. Dey, *Privacy and security policies in big data*. IGI Global, 2017.

[51] H. Hu, J. Xu, C. Ren, and B. Choi, "Processing private queries over untrusted data cloud through privacy homomorphism," in *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 2011, pp. 601–612.

[52] Q. N. T. Thi, T. K. Dang, H. L. Van, and H. X. Son, "Using json to specify privacy preserving-enabled attribute-based access control policies," in *International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage*. Springer, 2017, pp. 561–570.

[53] M. Ahmadian, F. Plochan, Z. Roessler, and D. C. Marinescu, "Securenosql: An approach for secure search of encrypted nosql databases in the public cloud," *International Journal of Information Management*, vol. 37, no. 2, pp. 63–74, 2017.

[54] Z. Kacimi and L. Benhlima, "Xacml policies into mongodb for privacy access control," in *Proceedings of the Mediterranean Symposium on Smart City Application*, 2017, pp. 1–5.

[55] F. P. Diez, A. C. Vasu, D. S. Touceda, and J. M. S. Cámara, "Modeling xacml security policies using graph databases," *IT Professional*, vol. 19, no. 6, pp. 52–57, 2017.

[56] T. Kudo, "Fog computing with distributed database," in *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 2018, pp. 623–630.

[57] R. Agrawal, J. Kiernan, R. Srikant, and 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.

[58] H. Hacıgümüş, B. Iyer, and S. Mehrotra, "Efficient execution of aggregation queries over encrypted relational databases," in *International Conference on Database Systems for Advanced Applications*. Springer, 2004, pp. 125–136.

[59] E. Mykletun and G. Tsudik, "Aggregation queries in the database-as-a-service model," in *IFIP annual conference on data and applications security and privacy*. Springer, 2006, pp. 89–103.

[60] A. Mousa, E. Nigm, E.-S. El-Rabaie, and O. S. Faragallah, "Query processing performance on encrypted databases by using the rea algorithm." *IJ Network Security*, vol. 14, no. 5, pp. 280–288, 2012.

[61] L. Chen, N. Zhang, H.-M. Sun, C.-C. Chang, S. Yu, and K.-K. R. Choo, "Secure search for encrypted personal health records from big data nosql databases in cloud," *Computing*, vol. 102, no. 6, pp. 1521–1545, 2020.

[62] M. U. Arshad, A. Kundu, E. Bertino, K. Madhavan, and A. Ghafoor, "Security of graph data: hashing schemes and definitions," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, 2014, pp. 223–234.

[63] G. Weintraub and E. Gudes, "Data integrity verification in column-oriented nosql databases," in *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 2018, pp. 165–181.

[64] ——, "Crowdsourced data integrity verification for key-value stores in the cloud," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2017, pp. 498–503.

[65] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine, "Authentic data publication over the internet 1," *Journal of Computer Security*, vol. 11, no. 3, pp. 291–314, 2003.

[66] P. Kalpana and S. Singaraju, "Data security in cloud computing using rsa algorithm," *International Journal of research in computer and communication technology, IJRCCT, ISSN*, pp. 2278–5841, 2012.

[67] S. Amghar, S. Cherdal, and S. Mouline, "Which nosql database for iot applications?" in *2018 international conference on selected topics in mobile and wireless networking (mownet)*. IEEE, 2018, pp. 131–137.

[68] S. Sicari, A. Rizzardi, D. Miorandi, C. Cappiello, and A. Coen-Porisini, "Security policy enforcement for networked smart objects," *Computer Networks*, vol. 108, pp. 133–147, 2016.

[69] A. K. Zaki and M. Indiramma, "A novel redis security extension for nosql database using authentication and encryption," in *2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. IEEE, 2015, pp. 1–6.

[70] V. Mishra, "Cassandra data security," in *Beginning Apache Cassandra Development*. Springer, 2014, pp. 61–78.

[71] T. Waage, R. S. Jhajj, and L. Wiese, "Searchable encryption in apache cassandra," in *International Symposium on Foundations and Practice of Security*. Springer, 2015, pp. 286–293.

[72] A. Golhar, S. Janvir, R. Chopade, and V. Pachghare, "Tamper detection in cassandra and redis database—a comparative," in *Proceeding of International Conference on Computational Science and Applications: ICCSA 2019*. Springer Nature, 2020, p. 99.

[73] S. Sathyadevan, N. Muraleedharan, and S. P. Rajan, "Enhancement of data level security in mongodb," in *Intelligent Distributed Computing*. Springer, 2015, pp. 199–212.

[74] M. Mathur and A. Kesarwani, "Comparison between des, 3des, rc2, rc6, blowfish and aes," in *Proceedings of National Conference on New Horizons in IT-NCNHIT*, vol. 3, 2013, pp. 143–148.

[75] P. Aggarwal and R. Rani, "Security issues and user authentication in mongodb." Elsevier Second International Conference on Emerging Research in Computing ..., 2014.

[76] P. Colombo and E. Ferrari, "Evaluating the effects of access control policies within nosql systems," *Future Generation Computer Systems*, vol. 114, pp. 491–505, 2021.

[77] E. Gupta, S. Sural, J. Vaidya, and V. Atluri, "Attribute-based access control for nosql databases," in *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, 2021, pp. 317–319.

[78] G. Xu, Y. Ren, H. Li, D. Liu, Y. Dai, and K. Yang, "Cryptmdb: A practical encrypted mongodb over big data," in *2017 IEEE International Conference on Communications (ICC)*. IEEE, 2017, pp. 1–6.

[79] S. Sicari, A. Rizzardi, D. Miorandi, and A. Coen-Porisini, "Security towards the edge: Sticky policy enforcement for networked smart objects," *Information Systems*, vol. 71, pp. 78–89, 2017.

[80] A. Wahane and Y. Jin, "A graph database approach for xacml role-based access control implementation." SEDE, 2018.

[81] M. Usman, I. Ahmed, M. I. Aslam, S. Khan, and U. A. Shah, "Sit: a lightweight encryption algorithm for secure internet of things," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 1, 2017.

[82] I. Sultan, B. J. Mir, and M. T. Banday, "Analysis and optimization of advanced encryption standard for the internet of things," in *2020 7th International Conference on Signal Processing and Integrated Networks (SPIN)*. IEEE, 2020, pp. 571–575.

[83] Y. Miao, J. Ma, X. Liu, J. Weng, H. Li, and H. Li, "Lightweight fine-grained search over encrypted data in fog computing," *IEEE Transactions on Services Computing*, vol. 12, no. 5, pp. 772–785, 2018.

[84] F. Meng, M. L. Cheng, M. Wang, P. Voulgaris, and H. Wee, "Abdks: attribute-based encryption with dynamic keyword search in fog computing," *Frontiers of Computer Science*, 2020.

[85] S. Namasudra, "An improved attribute-based encryption technique towards the data security in cloud computing," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 3, p. e4364, 2019.

[86] M. Ali, M.-R. Sadeghi, and X. Liu, "Lightweight revocable hierarchical attribute-based encryption for internet of things," *IEEE Access*, vol. 8, pp. 23 951–23 964, 2020.

[87] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis, "The role of trust management in distributed systems security," in *Secure Internet Programming*. Springer, 1999, pp. 185–210.

[88] L. Liu and Q. Huang, "A framework for database auditing," in *2009 Fourth International Conference on Computer Sciences and Convergence Information Technology*. IEEE, 2009, pp. 982–986.

[89] M. Bach and A. Werner, "Standardization of nosql database languages," in *International Conference: Beyond Databases, Architectures and Structures*. Springer, 2014, pp. 50–60.