Data Sci. Eng. (2016) 1(3):127–138
DOI 10.1007/s41019-016-0015-z

Fine-Grained Access Control Within NoSQL Document-Oriented Datastores

Pietro Colombo[1] · **Elena Ferrari**[1]

Received: 28 July 2016 / Accepted: 1 August 2016 / Published online: 30 August 2016

Abstract The recent years have seen the birth of several NoSQL datastores, which are getting more and more popularity for their ability to handle high volumes of heterogeneous and unstructured data in a very efficient way. In several cases, NoSQL databases proved to outclass in terms of performance, scalability, and ease of use relational database management systems, meeting the requirements of a variety of today ICT applications. However, recent surveys reveal that, despite their undoubted popularity, NoSQL datastores suffer from some weaknesses, among which the lack of effective support for data protection appears among the most serious ones. Proper data protection mechanisms are therefore required to fill this void. In this work, we start to address this issue by focusing on access control and discussing the definition of a fine-grained access control framework for document-oriented NoSQL datastores. More precisely, we first focus on issues and challenges related to the definition of such a framework, considering theoretical, implementation, and integration aspects. Then, we discuss the reasons for which state-of-the-art fine-grained access control solutions proposed for relational database management systems cannot be used within the NoSQL scenario. We then introduce possible strategies to address the identified issues, which are at the basis of the framework development. Finally, we shortly report the outcome of an experience where the proposed framework has been used to enhance the data protection features of a popular NoSQL database.

✉ Pietro Colombo
pietro.colombo@uninsubria.it

1 DiSTA, University of Insubria, Via Mazzini, 5, Varese, Italy

Keywords Fine-grained access control · Document stores · NoSQL · MongoDB

1 Introduction

NoSQL datastores are getting popularity in a variety of scenarios, and their diffusion is growing especially within the data management back-end of modern web applications, and the data storage and analysis layer of Internet of Things platforms. The reasons of NoSQL datastores diffusion range from outstanding performance and scalability, to the provided support for handling high volumes of data, as well as to the ease of interaction with external applications. As a matter of fact, NoSQL datastores outperform relational database management systems (RDBMSs) with respect to the efficiency of data analysis, the flexibility, and the scalability of data management. Current surveys[1] show that MongoDB,[2] which is the current most popular NoSQL datastore, immediately follows, in terms of diffusion, widely used RDBMSs, such as Oracle Database[3] or MySQL.[4] This shows that NoSQL datastores are affirmed solutions which compete in terms of diffusion with RDBMSs.

Different from RDBMSs, characterized by a standard reference data model and query language, a variety of proprietary query languages have been proposed for NoSQL datastores, as well as different data models have been introduced for them. Recent surveys have classified NoSQL databases into three main categories on the basis of

1 http://db-engines.com/en/ranking.
2 https://www.mongodb.com.
3 https://www.oracle.com.
4 https://www.mysql.com.

the adopted data model, namely key value, wide column, and document-oriented datastores [7]. Each of these classes is characterized by features that make the related datastores suited to specific application scenarios. Key-value datastores (e.g., Redis)[5] handle data modeled as pairs of keys and values. Data can be of primitive type or complex objects and are uniquely identified by a key. Such systems allow executing basic queries which retrieve values corresponding to given keys. They are very efficient in terms of used computational resources. Wide column stores (e.g., Cassandra)[6] are an evolution of key-value datastores, with more advanced data organization and analysis features. Data are collected into flexible tables, and they are modeled as heterogeneous records of variable size. Tables are flexible in that each row can be composed of a different set of columns, and columns, in turn, can be organized into column families. Finally, document-oriented datastores (e.g., MongoDB) model data as heterogeneous, hierarchical records, denoted as documents, which in turn are composed of sets of key-value pairs, each specifying a document field. Documents are grouped into collections, which in turn compose a database. Document-oriented datastores provide complex data management and analysis features and query languages and appear as the most flexible and complex currently available NoSQL datastores.

Even though the advanced data analysis and management features of NoSQL datastores are making them very popular, these platforms show several shortcomings, and, as highlighted in [18], one of the most serious is related to the poor data protection mechanisms they currently offer. For instance, Okman et al. [18] analyze the basic authentication and authorization features of MongoDB and Cassandra and propose possible strategies to enhance them.

In this work, we focus on access control features of NoSQL datastores, since access control is the core data protection module of any DBMS. Most of NoSQL systems adopt basic access control mechanisms operating at coarse-grained level. For instance, within document-oriented datastores, access control is enforced at the level of database or at the level of collection of documents. Even MongoDB, the most popular NoSQL datastore, integrates a role-based access control model operating at collection level only. While collection level protection is a good step forward with respect to several other systems operating at database level, it is still not sufficient to provide customized data protection levels, which could further raise the usability and diffusion of these systems.

It is well recognized that data management systems that handle sensitive data could greatly benefit from the integration of fine-grained access control (FGAC) features. FGAC has been recognized as a fundamental requirement in a variety of application scenarios, which range from data management and analysis systems (e.g., [8, 9, 22]), to social networks (e.g., [3, 5, 14]), and service oriented and mobile applications (e.g., [14]). Few NoSQL datastores provide a native support for FGAC, such as the key-value datastore Accumulo,[7] which enforces access control at cell level. However, the great majority of the existing systems do not enforce FGAC, and in this work, we aim at starting to fill this void.

Recent surveys on database popularity[1] rank document-oriented NoSQL datastores in the top position. This is probably due to the flexibility of these systems, the provided advanced analysis features, and the native support for the management of JSON[8] data, which, at present, is among the most common data exchange format of modern applications. For these reasons, in this work, we target document-oriented datastores. Unfortunately, as we will discuss throughout the paper, the schemaless data model of document-oriented datastores do not allow to straightforwardly reuse the FGAC enforcement mechanisms defined for RDBMSs. Moreover, so far no standard NoSQL query language has emerged yet (neither in general nor for a specific datastore category), and each datastore adopts a different language. This reduces the interoperability among the existing systems, for instance, up to now, it is not possible to write a query, even of basic type, which can be executed within several systems. Similarly, data portability can be problematic. For instance, even though within MongoDB and Counchbase[9] data are serialized as JSON objects, the importing of a MongoDB dataset into Couchbase requires preliminary data manipulation activities. The heterogeneity of NoSQL systems as well as of their query languages make the definition of a general FGAC enforcement solution a complex and ambitious task.

In this paper, we survey issues and challenges related to the development of FGAC enforcement monitors and their integration into document-oriented NoSQL datastores. The analysis of the literature lead us to identify possible strategies to address issues related to the definition of policy specification criteria, enforcement strategies, the implementation of the proposed mechanisms by an enforcement monitor, and aspects related to integration of the monitor into existing document-oriented datastores. The analysis described in this paper is partially based on early research experiences on NoSQL datastores that we did with MongoDB [10, 12], as well as on ongoing research activities finalized to the generalization of the approach in [10, 12].

5 http://redis.io.

6 http://cassandra.apache.org.

7 https://accumulo.apache.org.

8 http://www.json.org.

9 https://www.couchbase.com.

The remainder of the paper is organized as follows. Section 2 surveys related work. Section 3 describes the main issues related to the definition of a FGAC framework for document-oriented datastores. Section 4 discusses FGAC enforcement strategies, describing possible ways to address the previously identified issues. Section 5 shortly presents an application that shows how the proposed strategies can be actually applied for the enhancement of the access control features of MongoDB with FGAC. Finally, in Sect. 6, we conclude the paper shortly describing the state of our current research on access control within NoSQL systems, as well as introducing future research goals.

2 Related Work

FGAC has been integrated into several relational access control models, such as, for instance, the purpose-based model proposed in [6], and the action aware access control model in [9]. It has been also successfully deployed into some commercial RDBMSs (e.g., Oracle Virtual Private Database),[10] as well as in modern non-relational systems (e.g., Accumulo[7]).

Oracle Virtual Private Database (VPD) [4] is among the most known fine-grained access control framework for relational database management systems. Oracle VPD regulates the access to table rows by means of access control policies, which specify content- and context-based predicates that refer to properties of the protected data and the execution environment. Policy enforcement is achieved by means of query rewriting, appending the specified policy predicates to the *where* clause of the submitted SQL query. In [20], the enforcement approach used by Oracle VPD has been classified as a Truman model, where each data analyst has a partial view of the database. Rizvi et al. [20] claim that the user view may be inconsistent with respect to the information included in the database and propose an enforcement mechanism which only authorizes the execution of queries whose rewritten version do not bring to inconsistent views. Other approaches for RDBMSs enforce access control at a finer granularity level. For instance, LeFevre et al. [17] propose a SQL-based query rewriting approach which allows enforcing FGAC by means of dynamically generated authorized views of database tables. In [17], access control is enforced at cell level, generating views where all unauthorized cells are nullified. Agrawal et al. [1] describe an approach to transform RDBMSs into privacy-aware DBMSs, which relies on a language that supports the specification of grant commands at cell level.

[10] http://docs.oracle.com/database/121/DBSEG/vpd.htm.

Research efforts have also been recently focused on the integration of FGAC into NoSQL datastores (e.g., [10, 12, 16]) and map-reduce analytics platforms (e.g., [22]). For instance, Kulkarni [16] has proposed a fine-grained access control model for key-value systems denoted K-VAC, which has been first designed to operate with Cassandra,[11] and then extended for the integration into HBase.[12] However, the proposed solution is an ad hoc implementation and cannot be easily ported or adapted to other systems.

In [10], we have proposed the integration of a purpose-based model operating at document level into MongoDB. The successful experience brought us to refine the granularity and generalize the supported policies. Thus, in [12], we have proposed an access control model operating at field level which supports content- and context-based access control policies similar to those of Oracle VPD.

In [22], we have considered the enforcement of FGAC policies within map-reduce systems. The pairs key-value extracted from an accessed resource by a map-reduce job are dynamically modified on the basis of the specified FGAC policies, before the mapping phase starts the processing.

Overall, the research on the integration of FGAC into NoSQL datastores is still in the early stages. More specifically, for what document-oriented datastores are concerned, although the initial experiences that we had with MongoDB allowed us to identify some approaches to the definition and integration of FGAC into NoSQL systems, the proposed solutions need to be generalized to increase their applicability.

3 FGAC Within NoSQL Document-Oriented Datastores: Issues and Challenges

As briefly introduced in Sect. 1, the goal of this paper is to discuss how FGAC can be deployed within document-oriented NoSQL datastores. Although the goal is similar to the one already addressed within traditional DBMSs, intrinsic characteristics of the NoSQL scenario make this a challenge for data security researchers. Table 1 summarizes the main reasons for which we believe that the enhancement of document-oriented datastores with FGAC features is a far more complex and challenging tasks than designing a FGAC framework for RDBMSs.

In the remainder of this section, we shortly consider each of these points.

Generality A first aspect that should be taken into consideration is the heterogeneity of the existing NoSQL

[11] http://cassandra.apache.org.

[12] https://hbase.apache.org.

Table 1 Aspects affecting the complexity of FGAC solutions for data management systems

Aspect	Relational DBMSs	Document-oriented NoSQL datastores
Generality	Eased by a reference data model	Multiple declinations of the same data model
	Eased by a reference standard query language	Multiple proprietary query languages
FGAC granularity	Access control granularity up to cell level	Field level granularity is often a must due to data modeling choices
	Cell level policy specification based on a priori known tables schema	No a priori assumption on documents structure for specification purposes
	Cell level enforcement mechanisms based on a priory known tables schema	No a priori assumption on documents structure for enforcement purposes
Performance	Efficiency is important, but no very strict constraint, due to the size of traditional datasets	Efficiency is a must due to the high volumes of data. A proper trade-off between performance and security is needed, taking into consideration that performance is among the reasons for which NoSQL datastores are getting popularity
Enforcement mechanisms	Query rewriting to enforce access control at row and cell level	Techniques in the literature not applicable at field level, due to the schemaless nature of documents
	Native support for views	No systematic support for views

datastores and the need to define a general solution rather than an ad hoc solution operating with a unique NoSQL database (e.g., [10]). The complexity of the problem is partially due to the lack of a standard query language. Indeed, the enforcement approaches defined for RDBMSs rely on the presence of the relational model and SQL as unique data model and query language. In contrast, the variety of NoSQL datastores that have been defined so far, most of which operating with a different query language, make the definition of a general approach a very ambitious task. In addition, the lack of a reference standard has caused the definition of multiple implementations of the document-oriented data model, which differ for data organization features and terminology. For instance, some document stores do not integrate the concept of collection (e.g., CouchDB).[13]

FGAC granularity Let us now start to consider why the FGAC solutions developed for RDBMSs cannot be reused for the NoSQL scenario. To make the discussion more concrete, let us consider Oracle VPD, one of the most popular FGAC solutions developed for RDBMSs. Oracle VPD considers table rows as the finest protection objects. From a data management perspective, table rows of relational databases correspond to documents of document-oriented NoSQL datastores, even though documents model data resources in a less abstract way than rows, as they do not abstract from the intrinsic structure of a resource, and thus they do not require one to flatten the resource content.

Example 1 Let us consider a dataset of emails. An email has a structure providing meta information related to the message content. For instance, it includes a header and a body, where the header is in turn characterized by properties specifying, among others, the email sender, all the receivers, and the email subject. Within a document store, emails can be straightforwardly modeled as a document whose fields are hierarchically organized to match the email structure. In contrast, the modeling of the same dataset with the relational model requires to flatten the structure of an email by removing fields hierarchy.

At a first sight, due to the parallelism of concepts between the relational and document-oriented data model, fine-grained enforcement mechanisms operating at document level could be defined starting from the mechanisms proposed for RDBMSs. However, additional important aspects need to be considered. To be more concrete, let us consider again Oracle VPD. Within Oracle VPD, the key element of an access control policy is a boolean expression specified over table attributes and contextual properties. For instance, referring to the application scenario in Example 1, an access control policy could grant the access only to those emails that have been sent to a specific email address. Within the relational model the fields to, cc, and bcc are attributes of the email table scheme. In contrast, the schemaless data model of document-oriented NoSQL datastores brings to the definition of documents that include these fields only when the modeled email specify a receiver of that type, as for instance, an email may not have a bcc receiver or a cc receiver. As a consequence, a content-based policy could refer to fields which may not be included in all the documents. This implies the need to specify content-based access control policies under a different perspective, that is, not only considering fields values, but also even considering the structural characteristics of a document, such as the presence of a field.

[13] http://www.couchdb.apache.org.

For what access control granularity is concerned, it is worth noting that depending on the application context and the adopted modeling choices, document level granularity may be too coarse grained. As above mentioned, the hierarchical structure supported by the document-oriented data model allows representing data without abstracting from their structural characteristics. This brings one to define documents with a potential complex structure and many fields, and, as a consequence, access control policies can be defined to protect the access to a single field of a complex document with a hierarchical structure.

Example 2 Let us consider the application scenario introduced in Example 1, and an access control policy that regulates the access to field *from* of an email.[14] Let us suppose that a query that aims at accessing such an email is submitted for execution, and the access control policy that regulates the access to *from* is not satisfied. An access control framework similar to Oracle VPD cannot prevent the access to a single unauthorized sender field. In contrast, it would prevent the access to the whole document containing such a field. Such a mechanism is too restrictive as all other fields of the considered document could be freely accessed.

Within relational databases, the limits of row level access control brought researchers to define cell level access control mechanisms. For instance, Lefevre et al. [17] proposed to enforce cell level access control policies by means of query rewriting. The idea of the proposed mechanism is that a query q submitted for execution is rewritten as q′, in such a way that q′ integrates a subquery that derives an authorized view of each table t accessed by q and performs the analysis tasks of q on such a view. The subquery either projects or nullifies the value of cells on the basis of the satisfaction of the policies specified for them [17]. This technique requires to know in advance the scheme of the accessed data, as well as the name of the attributes that should be projected. In contrast, the schemaless nature of NoSQL datastores prevents the systematic use of similar techniques, as each document in a collection can be characterized by a different set of fields.

Example 3 Let us suppose to specify an access control policy which prevents the access to the fifth bcc receiver of an email when a given condition is not satisfied. The policy is applied to a single email, whose structure is potentially different from all other documents of the collection as it may be the only email with 5 bcc receivers. According to the approach in [17], one should know in advance the existence of an email with 5 bcc receivers within the dataset, in order to rewrite the query.

[14] *from* is a sub-field of the email header.

On the basis of the above-mentioned considerations, we believe that the heterogeneity of the documents collected within a NoSQL database makes the definition of a field level access control mechanism a challenging problem.

Performance An additional challenging aspect is related to the strict performance requirements that commonly characterize NoSQL systems, as the access control enforcement overhead should not compromise the efficiency of the considered systems. Indeed, NoSQL systems are often used in the back-end of applications where performance and scalability are first class requirements. We believe that reasonable trade-offs among security, performance, and scalability of the proposed enforcement mechanisms need to be identified, as secure systems with poor performance may suffer from low usability, but the same applies to highly efficient insecure databases.

Enforcement mechanisms The literature presents two main categories of enforcement approaches for FGAC, namely view-based and query rewriting mechanisms. The view-based mechanism consists in deriving authorized views of a resource, on the basis of the specified access control policies, and granting the permit to access that views instead of the original data resources. This approach suffers from several drawbacks. Indeed, different from relational databases, views are not supported by all NoSQL datastores, and thus ad hoc implementations are required within several NoSQL datastores.

The most straightforward solution probably consists in defining views as temporary collections, which store copies of authorized documents. Although this naive approach allows satisfying security requirements, from the engineering perspective the generation and storage of multiple views of the same resource appears quite impractical, both in terms of memory and time required for view generation and serialization. This naive approach suffers from low efficiency, large memory usage, difficulties to handle resource updates, and it is not scalable. On the other hand, disk view serialization may not be a practical solution due to the variety of views of the protected resources that must be generated and to the time required for write operations. Indeed, write on disk operations typically suffer from high latency, and the definition of multiple views of the same collection may not be possible for the collection size. In addition, even assuming that this naive approach can be used in some application scenarios, this solution requires to regenerate all views every time the protected resource is updated. As such, the cost of handling updates depends on the number of views that have been generated for a protected resource, and the number of documents that must be modified within each view.

Let us consider, for instance, the dataset of emails introduced in Example 1, and let us suppose that the collected emails are stored within a collection cl. The authorized view of cl is derived by considering all access control policies specified for cl documents and related fields. Several different views could be defined for the same dataset, which differ for the number of documents characterizing the protected collection, and the number of fields that characterize any generated image of cl documents. In addition, every time a document of cl is updated, it is also necessary to update the corresponding document of each derived view.

On the basis of the above-mentioned considerations, we believe that the view-based naive approach can only work with small datasets, in scenarios with a low number of stakeholders, which are inherently static.

A second type of enforcement mechanism is based on query rewriting. To the best of our knowledge, in the literature on RDBMSs, two different approaches have been proposed, which operate at row and cell level, respectively. The first one, implemented by Oracle VPD, operates by modifying the where clause of a submitted query q through the conjunction of the selection criteria of q with policy compliance predicates. This solution does not require any serialization, as in practice the rewritten query generates an authorized view at query execution time restricting the selection criteria of the original query. Oracle VPD operates at row level, and thus in the NoSQL counter part, it can only work at document level.

The second mechanism, introduced by Lefevre et al. [17], operates at cell level, either projecting or nullifying the value of each cell. The rewritten query is defined in such a way to include a subquery for each accessed table t, which substitutes the content of t with an authorized view of this resource, generated starting from the cell level access control policies. However, the schema of the accessed tables must be known in order to perform the rewriting, and thus, it cannot be directly applied to document-oriented datastores, as each document structure is potentially different from the ones of all other documents belonging to the same collection.

On the basis of the previous considerations, neither the view based, nor the query rewriting approaches proposed for RDBMSs can be directly applied with NoSQL document-oriented datastores.

4 Enforcement Strategies

The enhancement of NoSQL datastores with FGAC requires to identify proper engineering solutions for the encoding of fine-grained access control policies, the definition of enforcement monitors and the monitors integration into a target NoSQL system. In the remainder of this section, we discuss possible strategies to address the above-mentioned open issues.

4.1 Policy Encoding

The first considered issue is related to the approach to be used for the specification of FGAC policies. In the literature on relational DBMSs, several approaches have been proposed. For instance, in [6], purpose-based policies operating at different granularity levels are specified in dedicated tables. We believe that within NoSQL datastores, a similar specification approach can be used for access control policies operating at the collection level. The policies can be either coarse grained, thus implicitly regulating the access to the whole referred collection, or fine grained, thus regulating the access to documents of the referred collection which satisfy given selection criteria. However, we believe that FGAC policies should not be specified within dedicated collections, as, currently, join operations are not systematically supported by NoSQL datastores. In contrast, they can be stored within dedicated fields of the protected documents. In our previous work, we have used this approach for document level [10] and field level policies [12].

4.2 Enforcement

Abstracting from language and platform dependent aspects, we believe that a promising strategy to enforce FGAC within document-oriented NoSQL datastore consists in combining query rewriting with in memory view generation. Aware of the efficiency and consistency issues that affect naive implementations of the view-based approach (cfr. Sect. 3) and aligned with [17] principles, we believe that an effective enforcement approach should combine view generation and query rewriting, taking maximum benefit from the two mechanisms. According to the proposed approach, the views: (1) should not be serialized on disk, but derived in memory at run time and (2) should be directly generated by the rewritten queries, on the basis of the execution environment of the access request. More precisely, let q be a query that is submitted for execution. At an high level of abstraction, the overall goal of the approach consists in rewriting q as a query q' which derives an authorized view cl' of each collection cl accessed by q and performs the same analysis tasks as q accessing the derived views instead of the original collections. Given a query q, the idea is to first derive the selection criteria sc of q and the set of collections to be accessed by q. For each collection cl accessed by q, the criteria specified by sc as well as the access control policies specified for cl are used to select candidate documents of cl to be stored into cl'. Denoted with cl'', the set of candidate documents, for each document d in cl'', the approach prunes out from d any field f of d such that f is referred to by at least a policy p specified for d fields, but no policy referring to f is satisfied.

At an high level of abstraction, the rationale of view generation is aligned with the basic principles proposed in [17], but the schemaless nature of the document-based model requires a different rewriting mechanism as well as a different view derivation approach. As pointed out in Sect. 3, Lefevre et al. approach [17] requires to know in advance the scheme of the accessed tables. This cannot occur within the NoSQL scenario, where the structure of each accessed document is potentially unique and thus may differ from the ones of all other accessed documents. For instance, within the email dataset previously considered, the emails can have a varying number of to/cc/bcc receivers, whereas some of them may have no to/cc/bcc receiver. The goal is thus carrying out the projection, without knowing in advance the fields which characterize the accessed documents. As such, a possible solution is the one that operates by analyzing the structure of each document at execution time and thus differs from the methodology in [17], which relies on the a priori knowledge of the accessed tables schema. The idea is to consider the candidate documents as JSON objects, characterized by properties representing the document fields. These properties, which are modeled as key-value pairs, are iteratively accessed and modified by means of JSON manipulation functions. Different from [17], where the attributes to be projected are referred to by name within the rewritten queries, a possible approach for the NoSQL scenario consists in referring to the fields to be projected by position, iteratively considering any property of the documents.

In order to exemplify the rationale of the proposed approach and the differences with the one proposed for the relational model, let us consider again the dataset of emails referred to in Example 1 and the followings. Let us consider a query q that derives all types of receivers and the body of the emails that specify a given object. For the sake of simplicity and the lack of a standard query language for NoSQL databases, let us consider the SQL representation of q, which can be straightforwardly defined as *select to, cc, bcc, body from emails where object like "party"*.

Let us now consider how q can be rewritten on the basis of Lefevre et al. approach [17] within a RDBMS. For the sake of simplicity, let us assume that the scheme of an email is characterized by the fields from, to, cc, bcc, obj and body, and let us suppose that the access to any email field is regulated by a policy specified within field p. Let us suppose the existence of a function compliesWith that evaluates whether q execution complies with the access control policies specified for each email field. Listing 1 shows the pseudocode of the rewritten SQL query that derives the authorized view on email.

Listing 1 Query rewriting for FGAC at table cell level

```
select av.cc, av.to, av.subject, av.body from
(select
  case when compliesWith(p) then from
   else null end as from,
  case when compliesWith(p) then to
   else null end as to,
  case when compliesWith(p) then cc
   else null end as cc,
  case when compliesWith(p) then bcc
   else null end as bcc,
  case when compliesWith(p) then obj
   else null end as obj,
  case when compliesWith(p) then body
   else null end as body
 from emails where obj like "party" and
 compliesWith(cd,p)) av
```

According to Listing 1, the outer query accesses the view generated by the subquery, whereas the subquery projects email fields, provided that the access control policies specified for those fields are satisfied. As shown in Listing 1, any field of emails is explicitly referred to by name in the subquery.

Let us now consider a possible approach for the NoSQL scenario. Listing 2 shows the SQL-like pseudocode of the rewritten query.

Listing 2 Query rewriting for FGAC at field level within document stores

```
select av.cc, av.to, av.subject, av.body
from (select
  object <k,v> for <k,v> in object_pairs(e)
   when compliesWith(cd,p) end
 from emails e
 where obj like "party"
  and compliesWith(cd,p)) av
```

Similar to Listing 1, the derivation of the authorized view is achieved by a subquery, whereas the outer query simply projects fields of the derived view. In this case, the subquery generates an authorized image of each email using JSON manipulation operators. In the pseudo code, we have used the N1QL operator *object*,[15] which allows building an object by composition of key-value pairs of another object, which satisfies a condition. The authorized email images are thus defined in such a way to include any field of the original email e for which the specified access control policies are satisfied. In this case, we do not need to explicitly refer to the fields to be projected by name, and thus, it is not required to know in advance the fields composing the documents.

[15] For the sake of simplicity, Listing 2 reports a simplified syntax.

4.3 Monitor Implementation and Integration

The heterogeneity and variety of existing NoSQL datastores represent one of the main obstacles to the definition of a general development and integration approach. NoSQL document-oriented datastores typically provide APIs, often available for different programming languages, which support the programmatic interaction with the analysis and management features. Each set of APIs is a different interface to the provided services, and the enforcement monitor should be defined in such a way to regulate the fruition of these services, regardless they are invoked trough the APIs or by means of queries expressed with the supported data analysis languages. Although a possible integration strategy consists in the programmatic modification of the provided services, we believe that this cannot be a general and portable solution, as it strictly depends on the technological and architectural characteristics of each datastore, as well as on the availability of the source code, while the definition of a general solution requires to abstract from these aspects.

Due to the client–server nature of NoSQL systems, a possible way to handle the interaction with the datastore services is to define the enforcement monitor as a proxy. More precisely, the monitor should be responsible of the interaction of the datastore clients with the target server, exposing services to the clients and executing additional checks whenever a service is invoked.

Within several NoSQL datastores, the client–server interaction is achieved by means of dedicated, ad hoc defined, communication protocols, which regulate the information exchanged by the counterparts and the related format. Each message either encodes a client request or a server response to a client request. Client requests encode the invocation of analysis or management services provided by the server, whereas server responses encode the data that are returned by the server. Whenever a user invokes the execution of a service, either through an application or by means of a graphical interface, the request is first encoded and then sent to the server. Similarly, whenever the server completes the computation of a command, it encodes the response as a message sending it to the connected clients. In order to specify a client for a specific programming language, it is thus sufficient to implement an interpreter of the considered communication protocol, which is capable of encoding service invocations as messages. This suggests that, in order to enforce FGAC, one could focus on the messages exchanged by the clients and the server, rather than focusing on syntactical aspects of the query languages.

The proxy should thus be defined as an interpreter of the communication protocol supported by the considered datastore. This solution has the advantage of ensuring independence from APIs and programming languages, which are typically subject of continuous changes. In contrast, communication protocols are expected to be enhanced, to guarantee the interoperability of multiple client–server versions.

5 An Application Scenario: Enhancing MongoDB with FGAC

In this section, we discuss a possible implementation of the strategies proposed in Sect. 4 for MongoDB. A thorough presentation of technical aspects related to design, implementation, and integration of the proposed enforcement monitor can be found in [10, 12], which also provides a thorough discussion related to the efficiency and scalability of the proposed framework. In this section, we summarize relevant aspects related to policy specification and the developed enforcement mechanism.

5.1 Policy Specification

On the basis of the experience illustrated in [10], hereafter we describe a possible implementation of purpose-based access control policies, which are one of the most relevant type of FGAC policies. In this scenario, a policy p is a pair ⟨r, e⟩, where r specifies the list of resources for which p has been specified and e is the list of purposes for which the access to the protected resources is authorized, which are selected from a purpose set Ps. Access to a resource referred to within r is granted, if the access purpose ap associated with the access request complies with the purposes specified within component e of p. Coarse-grained policies regulating the access to collections of a database db have been specified within a dedicated collection cgp. In contrast, FGAC policies at document and field level have been directly specified within the protected documents.

Example 4 Let us consider the email dataset introduced in Example 1, and let us suppose that the considered dataset collects all the emails in collection cl. Let us now suppose that three purposed-based access control policies have been specified, which, respectively, grant: (1) the access to the collection of emails cl for marketing purposes; (2) the access to email em of cl for analysis purposes; and (3) the access to the list of receivers of em for research purposes. The first policy is specified as ⟨{cl}, {marketing}⟩ and encoded as a document of collection *cgp*, whereas the second and third policies are directly specified within a dedicated field of em.

5.2 Enforcement Strategies

The enforcement mechanisms for the considered scenario have been defined on the basis of the strategies introduced

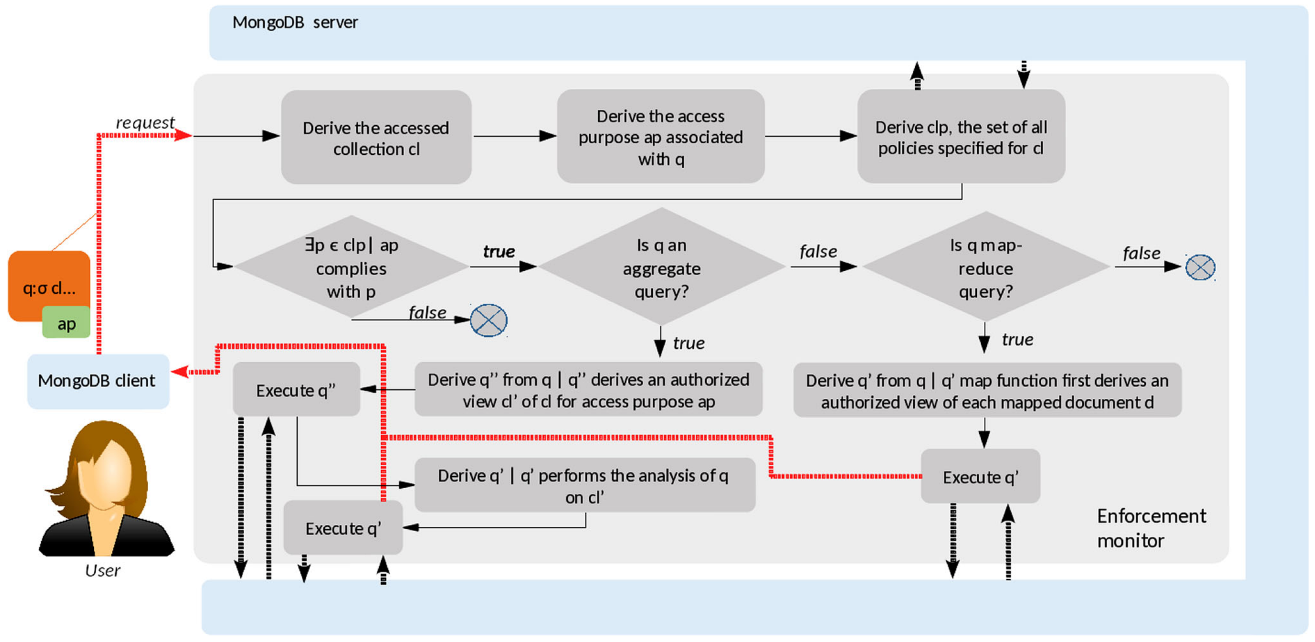
MongoDB server
request
Derive the accessed collection cl
Derive the access purpose ap associated with q
Derive clp, the set of all policies specified for cl
∃p ϵ clp | ap complies with p
true
false
Is q an aggregate query?
false
Is q map-reduce query?
false
true
true
q:σ cl...
ap
MongoDB client
Execute q''
Derive q'' from q | q'' derives an authorized view cl' of cl for access purpose ap
Derive q' from q | q' map function first derives an authorized view of each mapped document d
Derive q' | q' performs the analysis of q on cl'
Execute q'
Execute q'
Enforcement monitor
User

Fig. 1 Enforcement of FGAC within MongoDB

in Sect. 4. For space limitation, in this paper, we shortly discuss selected aspects related to the enforcement mechanism abstracting from model specific aspects as well as aspects related to authorization and compliance analysis within the purpose-based access control model.

The process, illustrated in Fig. 1, starts considering the query q which is submitted for execution and deriving the collection cl that is accessed by q. Afterwords, the process extracts from cgp all the access control policies that have been specified for cl. Then, the compliance of the access purpose of q with the purposes specified within the policies specified for cl is checked. If purpose compliance is satisfied for at least one policy, the process continues, otherwise it is immediately blocked due to the missing authorizations.

The process goes on considering FGAC policies specified for the documents to be accessed by q. Due to limitations of the supported query language, the mechanism is differentiated on the basis of the type of operation encoded by q. In particular, we focus here on aggregate and map-reduce operations, which represent the most advanced analysis operations of MongoDB. As introduced in Sect. 4, the enforcement mechanisms should derive an authorized view cl' of the collection accessed by q, and rewrite q as q' in such a way that q' executes the analysis specified by q on cl'. Two different rewriting approaches have been used for the considered operations, as MongoDB adopts a proprietary notation for aggregate queries, and Javascript for map-reduce queries. Abstracting from query language specific aspects which require a technical background on MongoDB, the main differences between the proposed approaches are related to the generation of the authorized view. In the case of map-reduce queries, the view is generated using the approach introduced in Sect. 4 and exemplified in Listing 2. However, different from Listing 2, where SQL-pseudocode is used, map-reduce queries are defined using Javascript functions. Listing 3 presents an example of a map-reduce query mrq, which counts the emails sent to addresses which have received at least one email.

Listing 3 An example of MongoDB map-reduce query

```
db.runCommand({
 mapReduce: clName,
 map: function() {
   var toF = this["headers"]["To"];
   for(i in toF){emit(toF[i], 1);}
 },
 reduce: function(key, values) {
   return Array.sum(values);
 },
 out: { inline: 1 }
})
```

The authorized view is derived by means of Javascript instructions which precede those performing the analysis and aggregation. Referring to the example in Listing 3, the query is rewritten in such a way that the first instructions of function map derive the authorized view of the mapped document. This is achieved by analyzing all field level policies regulating the access to the document, which according to the strategy proposed in Sect. 4, are referred to within a dedicated field. Given a document d, for each analyzed policy p specified for d fields, if the policy predicate of p is not satisfied, the fields referred by p are pruned out from d.

In contrast, limitations of MongoDB query language do not allow defining aggregate queries performing the derivation of the authorized view at the head of the operations pipeline. In this case, it is not possible to apply the approach introduced in Sect. 4, and the naive view generation approach comprising the serialization appears as the only viable solution. As such, this case is handled with the generation and sequential execution of two queries, which generate and serialize the view to a temporary collection and execute q processing activities on such a temporary collection.

Example 5 Let us consider an aggregate query agq, which counts the emails sent to given addresses, whose source code is shown in Listing 4.

Listing 4 An example of MongoDB aggregate query

```
var aliases = ["kenneth.lay@enron.com",
 "kenneth_lay@enron.net", "klay@enron.com"]
db.emails.aggregate([
 {$match: {"headers.To" :{"$in":aliases}}},
 {$group: {"_id":"sum", "num":{$sum:1}}}
]);
```

In this case, the rewriting is operated through the sequential execution of two queries: 1) a map-reduce query which derives an authorized view of the accessed collection on the basis of the policy compliance, as introduced for the example related to Listing 3; and 2) an aggregate query almost equivalent to agq.

5.3 Implementation

Let us briefly consider selected aspects related to the development of the enforcement monitor. According to the guidelines proposed in Sect. 4, the monitor has been defined as a proxy which analyzes the interaction of MongoDB clients and server. MongoDB client/server interaction is achieved by means of the Wire[16] communication protocol, which supports a request–response interaction initiated by the clients. The proposed monitor, which has been designed as an interpreter of the Wire protocol, analyzes all the requests that are issued by the clients to the server, rewrites the query execution requests in accordance with the previously discussed enforcement criteria, and executes complementary functionalities in support of access control enforcement, such as deriving access control policies applied to a resource, evaluating purpose compliance, handling the profiling of the users that issue query execution requests. The monitor has been developed as a Java multi-thread application deployed on a node which is at the interface of the network that hosts the MongoDB cluster.

We have extensively tested the developed monitor using both synthetic datasets and a real dataset, that is, the Enron corpus, a dataset of email messages comprising over 500K emails (1.5 GB of data) exchanged by around 150 employees of the Enron corporation [15]. The experiments targeting the synthetic datasets have considered a benchmark of 20 find, aggregate and map-reduce queries, which have been defined in such a way to ensure different complexity and selectivity levels.[17] Similarly, the benchmark targeting the Enron dataset includes 9 queries of type find, aggregate and map-reduce which have been inspired by the analysis functions presented in [21].

For our experiments, we have synthetically defined FGAC policies in such a way that these provide given selectivity levels. The experiments have assessed the enforcement overhead for the considered queries varying the selectivity of the considered policies. The results have shown that overall the overhead decreases with the increase in policy selectivity, due to the reduction of data that are accessed and analyzed by the queries. The experiments have shown that the overhead is negligible for find and map-reduce queries, but it is significant for aggregate queries. This confirms that the naive implementation of the view-based approach (cfr Sect. 3), which appears as the only viable solution for aggregate queries, suffers from low efficiency. The interested readers can refer to [10, 12] for a detailed presentation of the empirical results.

The experiences described in [10, 12] show significant differences between the measured overhead for document and field level access control policies. The overhead introduced with document level access control policies is low. Due to the effect of policy selectivity[18] with several queries of the considered benchmark, the execution time of the rewritten version of the queries is even lower than the execution time of the original queries. In contrast, the experience with field level access control policies reveals that the enforcement overhead varies with the type of the considered query. More precisely, aggregate queries suffer from a significant time overhead, whereas the overhead that has been measured with map-reduce queries is reasonably low.

[16] https://docs.mongodb.com/manual/reference/mongodb-wire-protocol/.

[17] By policy selectivity we mean the percentage of fields which are pruned out from the documents due to policy specification.

[18] We mean the effect of policy enforcement which brings to reduce the number of documents that are actually accessed.

6 Conclusions and Future Work

NoSQL datastores are getting increasing interest by users for their outstanding levels of flexibility, scalability, and performance, and their ability to manage huge data volumes. Despite this popularity, NoSQL datastores suffer from inappropriate data protection features. We believe that these shortcoming can be significantly addressed with the integration of FGAC features into the datastores. The integration of FGAC into a NoSQL datastore is a novel research topic that has been only recently addressed and that can open new research areas and applications. In this paper, we have discussed issues and challanges arising from the integration of FGAC features into NoSQL datastores. We have also described our experience with the MongoDB NoSQL datastore.

Our research is still progressing. In particular, we are currently focusing on recent standardization effort for NoSQL datastores, such as for instance SQL++ [19], a query language defined with the goal to become the reference query language for NoSQL datastores. SQL++ has been designed to keep compliance with SQL syntax, so that data analysts with a background on relational databases can easily migrate to a NoSQL datastore with a small initial effort. SQL++ is currently supported by AsterixDB[19] [2] and a few other datastores. A proprietary partial implementation of SQL++, denoted N1QL, is currently supported by the last version of Couchbase.[20] We believe that the involvement of industrial partners like Couchbase shows a concrete commercial interest in such a unifying solution. The availability of a general query language is an interesting basis for the specification of platform independent FGAC enforcement mechanisms, as well as for the development of multi-platform enforcement monitors.

Compliance with Ethical Standards

Competing of interest The authors declares that they have no competing of interests.

References

1. Agrawal R, Bird P, Grandison T, Kiernan J, Logan S, Rjaibi W (2005) Extending relational database systems to automatically enforce privacy policies. In: Proceedings of the 21st IEEE international conference on data engineering (IEEE ICDE)
2. Alsubaiee S, Altowim Y, Altwaijry H, Behm A, Borkar VR, Bu Y (2014) Asterixdb: a scalable, open source BDMS, PVLDB '14, pp 841–852
3. Bahri L, Carminati B, Ferrari E, Lucia W (2016) LAMP Label-based access control for more privacy in online social networks. In: Proceedings of the 10th WISTP international conference on information security theory and practice (WISTP 2016)
4. Browder K, Davidson MA (2002) The virtual private database in oracle9ir2. Oracle corporation, technical report 2002, oracle technical white paper
5. Buccafurri F, Lax G, Nicolazzo S, Nocera A (2016) A middleware to allow fine-grained access control of twitter applications. In: Proceedings of the international conference on mobile, secure and programmable networking (MSPN'2016)
6. Byun J, Li N (2008) Purpose based access control for privacy protection in relational database systems. VLDB J 17(4):603–619
7. Cattell R (2011) Scalable SQL and NoSQL data stores. SIGMOD Rec 39(4):12–27
8. Colombo P, Ferrari E (2014) Enforcement of purpose based access control within relational database management systems. IEEE Trans Knowl Data Eng (TKDE) 26(11):2703–2716
9. Colombo P, Ferrari E (2015) Efficient enforcement of action-aware purpose-based access control within relational database management systems. IEEE Trans Knowl Data Eng 27(8):2134–2147
10. Colombo P, Ferrari E (2015) Enhancing MongoDB with purpose based access control. In: IEEE transactions on dependable and secure computing (in press)
11. Colombo P, Ferrari E (2015) Privacy aware access control for big data: a research roadmap. Big Data Res 2(4):145–154. ISSN 2214-5796, Elsevier
12. Colombo P, Ferrari E (2016) Towards virtual private NoSQL datastores. In: 2016 IEEE 32nd international conference on data engineering (ICDE), Helsinki, Finland, pp 193–204
13. Jahid S, Mittal P, Borisov N (2011) EASiER: encryption-based access control in social networks with efficient revocation. In: Proceedings of the 6th ACM symposium on information, computer and communications security (ACM ASIACCS 2011)
14. Jin X, Wang L, Luo T, Du W (2013) Fine-grained access control for HTML5-based mobile applications in android. In: Proceedings of the 16th information security conference (ISC)
15. Klimt B, Yang Y (2004) The enron corpus: a new dataset for email classification research. In: Machine learning: ECML 2004. Springer, pp. 217–226
16. Kulkarni D (2013) A fine-grained access control model for key-value systems. In: Proceedings of the third ACM conference on data and application security and privacy, pp 161–164. ACM
17. LeFevre K, Agrawal R, Ercegovac V, Ramakrishnan R, Xu Y, DeWitt D (2004) Limiting disclosure in hippocratic databases. In: Mario A, Nascimento M, Tamer Z, Donald K, Rene JM, Jos A, Blakeley B, Schiefer K (eds) Proceedings of the thirtieth international conference on very large data bases (VLDB '04), vol 30. VLDB Endowment, pp 108–119
18. Okman L, Gal-Oz N, Gonen Y, Gudes E, Abramov J (2011) Security issues in NoSQL databases. In IEEE TrustCom
19. Ong KW, Papakonstantinou Y, Vernoux R (2014) The SQL++ unifying semi-structured query language, and an expressiveness benchmark of SQL-on-Hadoop, NoSQL and NewSQL databases. CoRR, abs/1405.3631
20. Rizvi S, Mendelzon A, Sudarshan S, Roy P (2004) Extending query rewriting techniques for fine-grained access control. In: Proceedings of the 2004 ACM SIGMOD international conference on management of data (SIGMOD '04). ACM, New York, NY, USA, pp 551–562

[19] https://asterixdb.apache.org.

[20] http://www.couchbase.com.

21. Russell MA (2013) Mining the social web: data mining Facebook, Twitter, LinkedIn, Google+, GitHub, and More. OReilly Media, Inc

22. Ulusoy H, Colombo P, Ferrari E, Kantarcioglu M, Pattuk E (2015) GuardMR: fine-grained security policy enforcement for MapRe- duce systems. In: ACM ASIACCS