Future Generation Computer Systems 114 (2021) 491–505

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Evaluating the effects of access control policies within NoSQL systems

Pietro Colombo *, Elena Ferrari

Department of Theoretical and Applied Science, University of Insubria, Via O. Rossi, 9 - 21100 Varese, Italy

ARTICLE INFO

Article history:
Received 15 November 2019
Received in revised form 10 March 2020
Accepted 16 August 2020
Available online 22 August 2020

Keywords:
Access control
NoSQL datastores
Big data
Authorized views

ABSTRACT

Access control is a key service of any data management system. It allows regulating the access to data resources at different granularity levels on the basis of access control models which vary on the protection options they offer. The more powerful is the access control model in terms of protection requirements, the more difficult is for security administrators to understand the effect of a set of access control policies on the protected resources. This is further complicated within schemaless systems, like NoSQL datastores, when fine grained access control policies are specified for data resources characterized by heterogeneous structures. The lack of a reference data model and related manipulation languages exacerbates this issue. To the best of our knowledge, a general approach to evaluate the impact of access control policies on the protected resources within NoSQL systems is still missing. In this paper, we start to fill this void, by proposing a data model agnostic approach, which, starting from schemaless datasets protected by different discretionary access control models, derives a view of the protected resources that points out authorized and unauthorized contents. Experimental results show the approach efficiency even with large datasets.

© 2020 Elsevier B.V. All rights reserved.

1. Introduction

Access control is among the major security services that are currently supported by RDBMSs and NoSQL datastores. For instance, in RDBMSs access control has been enforced according to a variety of models. These range from traditional ones, such as discretionary, mandatory, and role-based models [1], to more recent proposals aimed at enforcing customized forms of data protection. For instance, the copious family of Purpose Based Access Control (PBAC) models (e.g., [2,3]), and the Attribute Based Access Control (ABAC) models [4,5], are getting growing popularity. Discretionary (DAC) and role-based access control (RBAC) has also been used within NoSQL datastores (e.g., MongoDB[1] supports RBAC), whereas more recent work proposed the integration of PBAC [6], context-based access control [7], and ABAC [8,9]. Research proposals for RDBMSs and NoSQL datastores allow protecting the access to data resources up to the finest possible granularity (e.g., [9,10]). Although the majority of RDBMSs natively enforce DAC at table level, the use of views allow reaching finer granularity levels. Moreover, some commercial solutions targeting RDBMSs, such as Oracle Virtual Private Database [11], operate at fine grained level, achieving better scalability than views. As far as commercial NoSQL datastores are concerned, the majority operate at coarse grained level (e.g., MongoDB RBAC operates at document collection level), whereas few systems enforce access control at cell level (e.g., Accumulo).[2]

The protection options further increase when additional features are considered, such as negative/ positive policies, policy composition and conflicts resolution strategies, as well as policy propagation criteria [12]. Considering the variety of data models, access control models, and related configuration options, it can be really hard for security administrators to understand the effects of a set of access control policies on the data resources handled by their systems.

As an example, let us consider a relational database *db*, and a set *Ps* of access control policies (both positive and negative), which have been specified to regulate the access to *db* data at different granularity levels by a set of users *Us*. Any policy *p* of *Ps* protects the access to: (i) the whole database *db*, or (ii) to a table of *db*, or (iii) to a row, or (iv) to a cell of any *db* table. Let us now assume that a set of policy composition options, conflicts resolution strategies, and policy propagation criteria have been specified for the policies in *Ps*. For instance, suppose that denials take precedence has been selected as conflict resolution strategy. Let us also assume that two policies p_1 and p_2 apply to a cell *c* of *tb*, where p_1 is a positive policy (i.e., p_1 specifies a permit), whereas p_2 is a negative policy (i.e., p_2 expresses an explicit denial). Let us suppose that user *u* aims at accessing cell *c* at time *t*, and both p_1 and p_2 are satisfied at time *t* for *u*, causing a conflict.

* Corresponding author.
E-mail addresses: pietro.colombo@uninsubria.it (P. Colombo), elena.ferrari@uninsubria.it (E. Ferrari).

[1] https://www.mongodb.com/.

[2] https://accumulo.apache.org/.

https://doi.org/10.1016/j.future.2020.08.026
0167-739X/© 2020 Elsevier B.V. All rights reserved.

The conflict resolution strategy *denials take precedence* addresses this issue by favoring the negative policy p_2, which forbids the access to *c*.

A security administrator, on the basis of the specified policies and access control options, may wish to check which cells of a table *tb* of *db* can be accessed by a user *u* at time *t*. The same analysis could be repeated when different access control options are specified, or at a different time. Any of these evaluations requires checking all access control policies that regulate the access to any cell of *tb*. Policies need also to be combined on the basis of the specified access control options.[3] Overall this analysis results into a complex task.

This issue is even more relevant within NoSQL databases, as these systems allow the management of heterogeneous schemaless data, possibly characterized by complex hierarchical structures. Depending on the specified access control options, policies specified for a resource *r* may affect the permit to access any finer grained resource *r'* included in *r*, whose access, in turn, could be regulated by additional policies specified for *r'*. Deep hierarchical structures complicate policy analysis requiring the composition of access control policies specified for data resources at different granularity levels. It is worth noting that data resources with complex hierarchical structures are favored by different data modeling patterns adopted for NoSQL systems (e.g., see [13]). For instance, let us consider the data *denormalization* pattern [13], which is counted among the best practices for data modeling [13]. Any denormalized resource *dr* is defined in such a way to embed local copies of the data resources that are expected to be jointly accessed with *dr*. This modeling strategy favors very efficient data analysis, since there is no need to perform costly join operations as all data that need to be jointly accessed are included in a single resource, at the cost of data resources with quite complex structures.

Example 1. Let us consider a document oriented NoSQL database of an e-shop, which keeps track, in separate collections, of documents representing: the ordered items, the shipping of the ordered goods, and the customer profiles. Any order is stored in a document that keeps track of: details related to the ordered items, the customer who performed the purchase, and the possible use of a customer loyalty card. Fig. 1 shows two example documents, serialized in JSON format, each specifying data related to a purchase order. The considered documents have different structures, as the first document describes a purchase achieved using a customer loyalty card, whereas the second without any card.

Due to the composite nature of denormalized data resources, and the heterogeneity of the aggregated data, the access to any aggregated data item could be regulated by multiple sets of access control policies. For instance, referring to the JSON documents in Fig. 1, we could assume that the access to any field that refer to personal information (e.g., field *phone*) is regulated by a dedicated set of policies. Policies can be specified for data items at the finest supported granularity level (e.g., for *email*), or for a composite element (e.g., for *customer*). Depending on the specified set of access control options, the protection scope of the policies specified for a composite field, could be extended to the included sub-fields. For instance, policies specified for *customer* may also affect the access to *name* and *email*. The hierarchical structure of data resources favor scenarios where the access to any item of a composite resource is protected by multiple sets of policies specified at different granularity levels, which need to be properly composed.

[3] For instance, to determine whether a cell *c* can be accessed or not, it might be necessary to check policies that apply to *c*, to the row *rw* that includes *c*, to the table *tb* where *rw* is included, and to the database *db* that includes *tb*.

```
[
 {  "orderId": "kj7236545wx",
    "orderDate": "2019−08−04T15:32:44",
    "customerCard" : {
       "cardNumber": 415879,
       "emissionDate": "2019−07−02T14:45:19",
       "cardHolder":{
         "customerId":"1258FA43",
         "name":"John",
         "sex": "Male"
         "birthdate": "1990−08−12",
         ...
         "phone": "+123−1234567",
         "email": "john@email.com"
         ...
       },
   },
   "orderItems":[
     { "itemId":"15d6859k",
       "categoryId":"x73sw348"
       ....
       "price":15.58
     },
     ...
   ]
 },
 {  "orderId": "wkj3875xg",
    "orderDate": "2019−08−04T15:37:16",
    "customer":{
      "customerId":"1759AB3S",
      "name":"Mary",
      ...
      "email": "mary@email.com"
      ...
    },
    "orderItems":[
     { "itemId":"15lksdjf9k",
       "categoryId":"x172n4m48"
       ....
       "price":10.25
     },
     ...
   ]
 },
 ...
]
```

Fig. 1. Examples of purchase order documents.

Example 2. Let us consider again the scenario introduced in Example 1 and let us assume that a set of policies have been defined to allow the analysis of orders by analysts of a third party company who aim at identifying purchasing trends. More precisely, a set of collection level policies grant read access to collection *orders*, whereas a set of document level policies restrict the access authorization to a selection of *orders* documents which refer to a specific range of dates. A set of field level policies have also been specified, which map customer's preferences constraining the accessibility of personal data in case the purchase is achieved without a customer loyalty card. In this scenario, a security administrator may be interested to check the effectiveness of the specified policies and related access control options, by checking the accessibility of *orders* data by third party analysts. Due to the hierarchical structure of the considered documents, and the heterogeneous document structures, this analysis is significantly more complex than the one related to the RDBMSs scenario discussed at the beginning of this section, which presented a flat, tabular data organization. The access to any field of *orders* documents is regulated by multiple policies specified at different

granularity levels. For instance, the access to field *phone* in the first document of Fig. 1 depends on the policies specified for: (1) field *phone*, (ii) all fields preceding *phone* in the document structure (i.e., *cardHolder*, and *customerCard*), the document *d* where *phone* is enclosed, (iii) the document collection *orders* that comprises *d*, and)iv) the whole database *e-shop* where *orders* is included. Overall, the accessibility of any document field *f* can only be derived by composing, on the basis of the specified access control options, the policies specified for the elements preceding *f* in the hierarchical structure of the considered data resource.

Most of the previous research efforts dealing with the analysis of access control policies have been devoted to mechanisms finalized at verifying correctness, detecting inconsistencies, redundancies, and reasoning on completeness of policy sets (see Section 8 for more details). However, none of the existing approaches consider the effects of policy sets on data accessibility, which is the focus of current work, and none have been specifically designed to operate with NoSQL datastores. More precisely, for accessibility of a data resource for which a set of policies and access control options have been specified, we mean the indication, at the finest granularity level possible, of the data portions whose access is granted/denied.

In this paper, we start filling this void with an approach which, for schemaless datasets hosted by different NoSQL systems and protected by access control policies defined according to multiple DAC models and configuration options, allows assessing the impact of the considered policies on data accessibility. The proposed approach helps security administrators in configuring the set of access control policies for a NoSQL datastore, since it allows them to evaluate the effectiveness of access control policies before they are deployed into a target NoSQL datastores. The approach can ease the identification of a set of policies and access control options capable of granting an acceptable protection level for target datasets, in scenarios that potentially involve numerous subjects. For instance, security administrators may be interested to see which portions of a target dataset could be accessed by a new subject who joins their company/organization, if a specific set of policies and access control options were specified. It is worth noting that the great majority of NoSQL systems integrate quite basic access control features. For instance, MongoDB natively enforces RBAC at document collection level and only supports positive policies. Therefore, security administrators may decide to adopt third-party access control frameworks to grant a finer grained and customizable data protection. For example, Apache Ranger[4] allows the integration of advanced enforcement mechanisms in NoSQL databases of the Hadoop ecosystem, whereas several academic frameworks (e.g., see [7]) allow enforcing DAC according to a variety of models at fine grained level in target NoSQL databases. The proposed approach allows assessing the impact of policies to be supported by the se frameworks, also supporting, at the same time, those specified according to the native access control models of multiple NoSQL systems. Accessibility analysis can even be achieved before enabling any data protection mechanism in a NoSQL database. The analysis allows assessing the level of protection that would be granted by the adoption of a security framework characterized by: (i) a given access control model, (ii) a set of policies specified for such a model, and (iii) configuration options for the considered policies.

The flexibility that is required to operate with multiple NoSQL systems, and the intrinsic complexity of schemaless data resources, make the definition of this framework an ambitious goal. We approach the problem by first introducing a unifying data model capable of representing data resources of multiple NoSQL data models, and supporting the specification of policies according to the major DAC models. Along with the data and the specified policies, the unifying model allows tracing structural and security-related metadata characterizing a target resource, which are then used for policy analysis purposes. The unifying model is complemented with data-model agnostic services supporting the mapping of a target data resource referring to a native data model to a resource of the unifying model and back. This representation is then used to assess the accessibility of the target resource, by generating a view that shows its authorized and unauthorized contents.

In order to maximize portability, the proposed approach has been built on top of MapReduce [14]. Indeed, several NoSQL systems provide native support for this computational paradigm (e.g., MongoDB), and connectors exist which allow the execution of MapReduce tasks within systems not having a native support for it.

The approach is independent from a specific data model, thus, it can be potentially used with NoSQL systems operating with the document oriented, key–value, and wide column data models. It can be also easily extended to more traditional DBMSs (e.g., relational, object-oriented). Taking advantage of this flexibility, our framework can also be profitably used in federated database systems that involve multiple heterogeneous NoSQL datastores. The proposed unifying solution allows system administrators to evaluate data accessibility with policy sets that regulate the access to any database of the federation.

The approach supports access control policies expressed with all the major DAC models. However, in this paper, to simplify the presentation, we consider policies referring to the ABAC model [4]. Experimental evaluations of the analysis process efficiency performed on real datasets show good performance results.

To the best of our knowledge this is the first work aiming at assessing the effect of access control policies on the protected resources within NoSQL systems.

The remainder of the paper is organized as follows. Section 2 introduces background knowledge. Section 3 discusses requirements for the proposed framework. Section 4 introduces the unifying data model, whereas Section 5 details our approach. Section 6 presents the experiments. Finally, Section 9 concludes the paper.

2. Background

2.1. MapReduce program synthesis

The algorithms at the basis of our approach are presented using a notation for MapReduce tasks inspired by [15].

A MapReduce task *mr* can be seen as a function which, starting from a collection of elements of type *T*, by composition of parallel operations, derives a collection of elements of type *T'*, where *T'* models a key–value pair *(k, v)*. Aligned with the notations presented in [15], the operations of each MapReduce computation are parametrized as functions. Therefore, *mr* consists in the parallel execution of multiple instances of a *map* function *m* and of a *reduce* function *r*. A mapper forwards any element received as input by *mr* to an instance of *m*, which once analyzed the element, emits a key–value pair *(k, v)*. For each distinct key *k* emitted by *m* instances, the emitted pairs that specify *k* as key are forwarded to a reducer, which processes them by means of the reduce by key function *r*. Function *r* is invoked specifying as input the key *k*, and the collection of value components of the redirected pairs. *r* aggregates the collection of values received as input, returning a key–value pair *(k, v)*. If a finalization function *f* has been specified for *mr*, *f* is invoked once *r* completes the

[4] https://ranger.apache.org/.

execution. f receives as input the key–value pair generated by r, and returns a pair possibly specifying a new value component.

mr is thus specified by means of the notation (map m)* $\mapsto$ (reduce $r \triangleright$ finalize f)*, where, * denotes parallel executions of the functions between the round brackets, $\mapsto$ denotes the flow of key–value pairs emitted during the mapping phase of mr, which are provided as input to the reduce by key phase, whereas $\triangleright$ is used to denote the execution of f once r execution is ended.

Functions m, r, and f may in turn rely on auxiliary functions for their computation. The notation $f_1 \Rsh f_2$ is used to denote the invocation of a function f_2 within the execution of a function f_1. For instance, (map $m \Rsh f_1$)* $\mapsto$ (reduce $r \Rsh f_2 \triangleright$ finalize f)*, specifies that f_1 is executed during the execution of m, whereas f_2 is executed during the execution of r. Finally, in case multiple functions $f_1..f_n$ are sequentially executed during the computation of a function f_0, the ordered sequence of executions is specified by listing the considered functions between squared parenthesis. For instance, (map $m \Rsh [f_1, f_2, f_3]$)* $\mapsto$ (reduce $r \Rsh f_4$)* denotes the execution of f_1, f_2, and f_3 during the execution of m, and the execution of f_4 during the execution of r.

2.2. Attribute based access control (ABAC)

We now focus on the core features of the ABAC model [4,5], which, for its flexibility and growing diffusion (e.g., [8,9]), has been chosen to illustrate the proposed approach.

ABAC policies are built on top of the concepts of *subject*, *object*, and *environment*. A *subject* is a model of a user that sends access requests, and is characterized by attributes specifying properties of the modeled user, such as the covered roles. An *object* is a model of a data resource which is protected by a policy. Objects can model coarse-grained resources, such as databases, as well as resources at finer granularity levels, like documents or related fields. Objects are characterized by attributes, specifying properties of the modeled resources, for instance, metadata specifying the sensitivity level of the resource content. Finally, an *environment* is a model of the context within which an access request is issued, and it is composed of attributes modeling context properties. For instance, an environment attribute can specify the time at which an access request has been issued by a subject.

ABAC policies regulate the access to an object by a subject within an environment on the basis of the satisfaction of constraints specified on object, subject, and environment attributes. For instance, an ABAC policy specified for an object o may require that at least one of the roles of the subject s who issues the request to access o is authorized to access data with sensitivity level greater than or equal to the one specified for o, and the MAC address of the device which is used by s to request the access belongs to a list of authorized devices.

3. Requirements

Let us now focus on the key requirements that we have considered in defining an approach to evaluate the impact of access control policies on data handled by NoSQL systems. These requirements have been derived from the literature on access control (e.g., [12,16]), existing enforcement monitors (e.g., Apache Ranger[5]), and features of NoSQL datastores.

Flexibility is a key goal, as NoSQL systems operate with different data and AC models, handling schemaless data with heterogeneous structures. Resources can be protected by multiple policies, thus proper policy combination strategies are required. The approach should support both the minimum and maximum privilege strategies. As such, the *combining options* (co) *all* and *any* need to be supported (e.g., see Oracle Vault[6]). According to the option *all*, an access request *ar* targeting a resource *rs*, for which a set *Ps* of policies has been specified, is authorized if all policies in *Ps* are satisfied. In contrast, the option *any* requires that at least one policy in *Ps* is satisfied.

Although DAC models of the majority of DBMSs support positive policies, some proposals also enforce negative policies, which express explicit denials [1]. For instance, Apache Ranger allows enforcing positive and negative policies within HBase (https://hbase.apache.org) platforms. As such, the approach has to support *conflict resolution strategies* (crs) to handle possible conflicts among positive and negative policies protecting the same resource, such as the strategies *permissions take precedence* and *denials take precedence*, which respectively prioritize positive and negative policies (e.g., see [12,17]).[7]

Some data resources may not be covered by any policy. A system is denoted as *open*/*closed*, if it authorizes/prohibits the access to resources for which no access control policy has been specified. The proposed approach has therefore to operate in both open and closed systems.

Policy propagation is another feature affecting the impact of policies on data accessibility. For instance, in a document store, a policy specified for a document d can affect the decision to access a field of d. The approach has to support state of the art *policy propagation criteria* (ppc), like *no propagation*, *no overriding*, and *most specific overrides* (e.g., see [12]). When the option *no propagation* is used, the access decision related to a resource dr is not propagated to any included resources. The option *most specific overrides* propagates the access decisions from dr to any included resource dr' unless at least one policy has been specified for dr'. In this case, the decision derived from the local policies prevails. Finally, the option *no overriding* propagates the access decisions derived for dr to any finer grained resource included in dr, where it is combined with the local policies.

A summary of the required features is presented in Table 1.

Table 1
Access control options.

Access control option		Supported criteria
co	Combining options	{*any*, *all*}
crs	Conflict resolution strategies	{*denials take precedence*, *permissions take precedence*}
ppc	Policy propagation criteria	{*no propagation*, *no overriding*, *most specific overrides*}
st	System type	{*open*, *closed*}

4. Unifying data model

The proposed approach has been designed for NoSQL datastores operating with the key–value, wide column, and document-oriented models [18]. Since the considered data models refer to data resources using heterogeneous terms, hereafter we introduce a data model independent unifying terminology.

The term *data unit* denotes a data resource at the finest granularity level at which data insertion can be executed in a NoSQL system. Within key–value stores, data units map ⟨*key*, *value* ⟩ pairs, within wide column stores they map table rows, whereas within document stores, data units map documents. Data units can either represent data of simple type (e.g., numeric), denoted as *basic resources*, or data of complex type (e.g., an object),

5 https://ranger.apache.org.

6 https://docs.oracle.com/cd/B28359_01/server.111/b31222/toc.htm.

7 These strategies map the XACML policy combining algorithms *permit override* and *deny override*, respectively.

Table 2
Reference notation summary.

Notation	Meaning
`emit(k,v)`	Denotes the emission of a key–value pair ⟨k,v⟩
$\pi_f(o)$	Denotes the value of field f of object o
$\Gamma(o)$	Denotes the set of all fields composing the first layer of the hierarchical structure of object o
`setField(o,f,v)`	Denotes the initialization of field `f` of object `o` to the value `v`, where `f` is a field at the first layer of `o` structure. In case field `f` does not exist within o, `f` is added to `o` structure, and then initialized to `v`
`delField(o,f)`	Denotes the removing of field `f` from object o, where `f` is a field at the first layer of `o` structure
`push(S,e)`	Denotes the inclusion of element `e` within the set `S`

denoted as *composite resources*. Data units mapping composite resources are composed of more elementary entities, referred to as *data unit components*, each of which in turn maps a basic or a composite resource. Data unit components refer to the finest granularity level at which read and update operations can be executed. For instance, in wide column stores, a data unit component can either map a row cell, or groups of cells belonging to the same column family. Finally, in document stores, components of a data unit *du* representing a document *dc* map the fields of *dc*, which can either be basic, or composite resources.

Different criteria are used to represent coarse grained resources in different NoSQL data models. Key–value pairs are stored within databases denoted as key-spaces. Document and wide column stores introduce a resource hierarchy. Indeed, collections and tables group documents and table rows, and are included in databases. Data resources are seen as entities collecting either fine grained or coarse grained resources. Containment relations are exclusive in all data models, thus data resources can be represented as trees. More precisely, let *dr* be a data resource and let T_{dr} be the tree representing the structure of *dr*. Each tree node v of T_{dr} refers to finer grained data included in *dr*. Let us denote with *unifying resource property* (*urp*) a node of a resource tree. A data resource *dr* can thus be modeled as a set of *urp*s related among them to form a tree. Any *urp* contributing to the definition of *dr* specifies the inclusion within any *urp* that maps a (sub)resource of *dr*.

Definition 1 (*Unifying Resource Property*)**.** Let *dr* be a data resource which refers to a data model *dm*, and let *sdr* be a resource included in *dr*. The unifying resource property *urp* modeling *sdr* is a tuple ⟨*id*, *path*, *k*, *v*⟩, where *id* is the identifier of *urp*, *path* specifies a list of identifiers referring to the unifying resource properties that precede *sdr* within *dr*'s structure, while *k* and *v* specify the identifier of *sdr* within *dm*, and its optional value, respectively.

Component *path* of *urp* specifies a relative path which allows positioning *sdr* in *dr*, whereas component *v* may be left unspecified. The value is specified only when unifying resource properties are used to represent data resources at the finest granularity level.

Example 3. Suppose *dr* is a document oriented database of emails. *dr*'s emails are documents, whereas email properties, such as *body* and *subject*, are document fields. *dr* can be modeled as a set of urps. Any urp modeling an email field *f* refers to the urp representing the email document that includes *f*. The approach is recursive, thus, any urp that maps an email *e* refers to the urp that models the document collection where *e* is included. Finally, the urps of these collections refer to a urp that maps the whole database *dr*. Let us now consider the definition of a unifying resource property *u*, which maps an email document *e* of a document collection *c* included in *dr*. Let us suppose that the urps referring to *c* and *dr* specify u_c and u_{dr} as identifiers, respectively. Let us also assume that u_{id} is a unique identifier within the namespace of urps specified for *dr*. *u* is thus specified as $\langle u_{id}, [u_{dr}, u_c], e_{id}, \perp \rangle$, where u_{id} is the identifier of *u*, $[u_{dr}, u_c]$ specifies the urps preceding *u* within the tree structure of *dr*, e_{id} is the identifier of *e*, and $\perp$ indicates that no value is explicitly specified within *u* as *e* is a composite resource. The content of *u* is modeled by urps which map data resources included in *e* (e.g., *body* and *to*).

The urps specified for *dr* and its internal resources form the *unifying resource model* of *dr*, namely the representation of *dr* within the unifying data model.

Definition 2 (*Unifying Resource Model*)**.** Let *dr* be a data resource of a data model *dm*. The *unifying resource model* of *dr*, denoted as dr^*, is the set of all urps which map the resources included in *dr*. More precisely, $dr^* = \bigcup_{v \in V_{T_{dr}}} u_{r_{dr}^v}$, where $V_{T_{dr}}$ denotes the set of nodes in the tree representation of *dr* and $u_{r_{dr}^v}$ denotes the unifying resource property modeling the resource included in *dr* whose tree structure specifies *v* as root.

The proposed model is general enough to represent resources of different data models at any granularity level. For instance, considering the document oriented data model, *dr* can represent a database, a collection, a document, or a field.

5. The approach

Our aim is to define a general approach to evaluate the impact of a policy set on the accessibility of schemaless resources handled by NoSQL datastores operating with different data and DAC models. The generality is attained by leveraging the unifying data model (see Section 4), and by building the approach on top of *MapReduce* [14], which is extensively supported by NoSQL systems. The approach relies on bidirectional mappings between data resources referring to a native NoSQL data model and resources in the unifying model.

We define a process articulated into 3 phases, presented in Fig. 2. Starting from: (1) a target data resource *rs* of a native data model, (2) a set of access control policies *Ps*, and (3) of access control options *ao* (cfr. Table 1), the process derives a view *rs'* of *rs* showing the effects of policy enforcement. Phase 1 focuses on the derivation of rs^*, the unifying resource model of *rs*. A map-reduce job, denoted *unifier*, identifies *rs* components, and derives a urp for each component (cfr. Section 5.1). Phase 2 focuses on the specification of security metadata and access control policies regulating the access to *rs* (see Section 5.2). Policies and metadata are bound to the data by labeling the urps derived in phase 1. Finally, phase 3 handles the evaluation of policy impact on data accessibility. A map-reduce job, denoted *projector*, analyzes the considered policies and access control options, and derives a view *rs'* of *rs* that shows the accessibility of *rs* components, pointing out those authorized and unauthorized (see Section 5.3).

In the remainder of this section, we present in more detail each of the above mentioned phases.

5.1. Phase I: Derivation of a unifying resource model

Let us now consider how a data resource *dr* represented in its native data model can be mapped to the unifying data model. *dr* can represent data resources of any type, referring to any of the supported data models, and the mapping can be configured to operate at any granularity level. Hereafter, the approach is

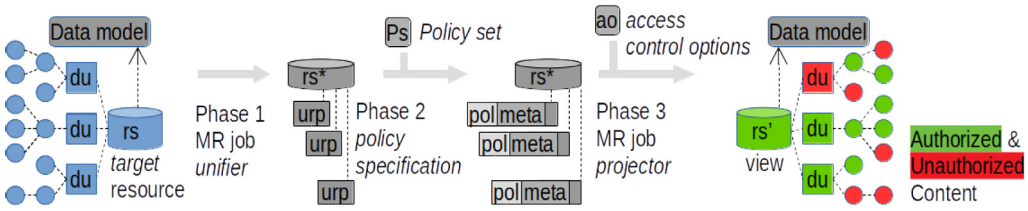

Fig. 2. Overview of the approach.

illustrated for coarse-grained composite resources which map sets of data units, such as key-spaces, collections of documents, or tables. Resources at coarser grained level (e.g., databases) can be handled by applying the approach to all sets of data units composing the resource. The mapping is achieved by a MapReduce task *unifier* (see Algorithm 1), which receives as input *dr*, visits its tree structure, (2) derives a urp for each node of the tree, and, by composition of these elements, (3) derives dr^*, the unifying resource model of *dr*. The approach relies on a few basic functions, summarized in Table 2.

Algorithm 1: The MapReduce task *unifier*

MapReduce task *unifier* **is**
  input : a data resource dr that refers to a data model *dm*, composed of data units *du*
  output: dr^*
  (map m ↱ *duMapper*)* ↦ (reduce r)*
end

The mapping function *m* of *unifier* (see Algorithm 2) is executed for any data unit *du* of *dr*. For any component of *du*, *m* emits a key–value pair that models a urp (see Definition 1). The key of the emitted pair is the identifier of the urp, whereas the value is a record structured according to the tuple in Definition 1.

Algorithm 2: The mapping function *m* of *unifier*

function *m* **is**
  input : du
  output: ⟨*k*, *v*⟩, where *v* models a unifying resource property *urp*, and *k* refers the value of field *id* of *urp*
(1) **var** urpS:=duMapper (∅, du);
(2) **for** urp ∈ urpS **do**
(3) emit (π_{id}(urp),urp);
  end
end

The analysis of data units and related components is handled by function *duMapper* (see Algorithm 3), which is executed on request of *m* for a data unit *du*. *duMapper* performs the recursive analysis of *du* components, deriving a set *urpS* of objects modeling urps, which, once returned to *m*, are emitted as key–value pairs. The analysis performed by *duMapper* starts by considering the data resource *obj*.

duMapper visits the tree structure of *obj*, and, for each field *f* of *obj*, generates an object *urp* that includes the fields *path* and *K*, which specify the position and identifier of *f*, and an optional field *V*, which keeps track of the value of *f*. If *f* maps a simple field of *du*, field *V* is initialized to the value of *f*. In contrast, if *f* maps a complex property (i.e., if field *f* of *obj* is an object), *V* is not set, and *duMapper* is recursively invoked on the object representing the value of *f* within *obj*. The recursive execution of *duMapper* results into a set of objects *sUrpS* which is merged with *urpS*.

```
{"_id" : "3e29",
 "body" : "I'm ready, are you?....",
 "mailbox" : "bass-e",
 "headers" : {
  "From" : "daphneco64@bigplanet.com",
  "To" : "eric.bass@enron.com",
  "Date" : "Tue, 14 Nov 2000 07:25:00 -0800 (PST)",
  "Message-ID" : "<1075854677459.JM.evans@thyme>",
  "Content-Type" : "text/plain; charset=us-ascii",
  "Subject" : "Re: check it out"}
}
```

Fig. 3. An example of data unit representing an email document.

Algorithm 3: The auxiliary function *duMapper* of *unifier*

function *duMapper* **is**
  input : cPath, obj
  output: a set urpS of objects modeling unifying resource properties
(1) **var** urpS:=[];
(2) **for** f ∈ Γ(obj) **do**
(3) **var** urp:=⊥;
(4) setField (urp, "path", cPath);
(5) setField (urp, "id", generateId ());
(6) setField (urp, "K", f);
(7) **if** |Γ(π_f(obj))|> 0) **then**
(8) **var** sPath:=valueOf (cPath);
(9) push (sPath,π_{id}(urp));
(10) **var** sUrpS:=duMapper (sPath,π_f(obj));
(11) urpS:=urpS ∪ sUrpS;
  else
(12) setField (urp,"V",π_f(obj));
  end
(13) push (urpS,urp);
  end
(14) **return** urpS;
end

The reduce function *r* keeps unvaried the pairs emitted by *m*. Thus, the urps resulting from the mapping compose dr^*.

Example 4. Let *du* be an email of *dr* (see Example 3), serialized in JSON format in Fig. 3. Let us now consider the execution of *unifier* specifying *dr* as input data resource. Function *m* is executed for any data unit included in *dr*. In particular, during the execution of *m* targeting *du*, *duMapper* is invoked by *m* to analyze *du*, and derives a urp for any field *f* of *du*. The set *urpS* of all generated urps is then returned to *m*, which emits one key–value pair for each included element. A sample of 3 emitted key–value pairs is shown in Fig. 4.

Field *path* of any generated urp specifies the position of the modeled data within the data resource. The first, second and third elements of *path* in Fig. 4 refer the database, collection and

```
{ "_id" : "53d6",
  "value":{
     "path":["53d3","53d4","3e29"],
     "id" : "53d6",
     "K" : "body",
     "V" : "I'm ready, are you?...."}   }
{ "_id" : "53da",
  "value" : {
     "path":["53d3","53d4","3e29"],
     "id" : "53da",
     "K" : "headers"}                    }
{ "_id" : "53dc",
  "value" : {
     "path":["53d3","53d4","3e29","53da"],
     "id" : "53dc",
     "K" : "From",
     "V" : "daphneco64@bigplanet.com"}   }
```

Fig. 4. A sample of the unifying resource properties generated by *unifier*.

document that include the modeled fields. The urps that model a field f at level 2 of du structure also keep track of the field that includes f. Thus, for field *From*, the last element of *path* refers to the identifier of the urp modeling field *headers*.

5.2. Phase II : Policy specification and binding

The proposed framework supports the analysis of access control policies specified according to different DAC models, protecting up to single data unit components. It supports the analysis of both negative and positive policies, possibly specified on top of security metadata, as several access control models use them (e.g., [3] and [19]).

The binding of policies and security metadata to the protected resources is achieved by integrating these data in urps. Two fields are added to the tuple specifying a urp: (1) *meta*, which keeps track of the security metadata; and (2) *pol*, which models the policies specified for the mapped resource.

An access control policy *acp* specified for a unifying resource property *urp*, is a pair ⟨*exp, tp*⟩, where *exp* specifies a boolean expression constraining the access to *urp*,[8] whereas *tp* specifies if *acp* is positive or negative. In contrast, security metadata are specified as sub-fields of component *meta* of *urp*.

In this paper, policy specification and binding is exemplified with ABAC policies. A policy p protecting data modeled by a unifying resource property *urp* is specified within component *pol* of *urp*, in such a way that field *exp* of p specifies a boolean expression defined by composition of the variables s, o, and e, which model subject, object, and environment attributes.

Example 5. Let us consider the specification of security metadata related to a data unit component *duc* that models field *body* of the email in Example 4. Let us suppose that security metadata are used to specify the purposes for which *duc* can and cannot be accessed. Let *urp* be the unifying resource property modeling *duc*. Field *meta* of *urp* is defined in such a way to include the fields *aip* and *pip*, which specify the purposes for which the message can/cannot be accessed.

The ABAC policies *p1* and *p2* regulate the access to *body* on the basis of purpose compliance [2]. *p1* is a positive policy specifying the predicate *"s.ap∈meta.aip"*, whereas *p2* is a negative policy specifying the predicate *"s.ap∈meta.pip"*. Both predicates refer: (1) the subject attribute *ap*, specifying the access purpose of the subject s, and (2) the properties *aip*/*pip* of the security metadata specified for *duc* within *urp*, which specify the purposes for which *duc* can/cannot be accessed. The binding of these policies is achieved as shown in Fig. 5.

[8] *Exp* is defined by composition of variables referring to conceptual elements of the considered access control model (e.g., subject, object), mathematical operators (>, <, =, +, −, ∗, /), logical operators (∧, ∨, ¬), set operators (∈, ⊂, ⊆, ∩, ∪, \), and logical quantifiers (∀, ∃).

```
{"_id" : "53d6",
 "value" : {
  "path" : [ "53d3","53d4","3e29"],
  "id" : "53d6",
  "K" : "body",
  "V" : "I'm ready, are you?...."
  "meta": [{"aip": ["research","administration"]},
   {"pip": ["marketing"]}],
  "pol":[{"exp":"s.ap in meta.aip","tp":"positive"},
   {"exp": "s.ap in meta.pip","tp":"negative"}]}
}
```

Fig. 5. Unifying resource properties integrating policies and security metadata.

5.3. Phase III: View generation

Phase III is the core task of the whole approach, which maps resources of the unifying model back to the original data model, deriving a view of the original resource, which on the basis of the specified policies and access control options, points out authorized and unauthorized contents. To ease the comprehension of this complex task, in Section 5.3.1, we introduce the rationale of the mapping task, which is instrumental to the definition of the view generation approach, later presented in Section 5.3.2. The mapping is instrumental to assess the effects of policies and related options on the accessibility of the protected resources. We believe that security administrators could better perceive these effects with a view derived for the original data model, rather than by means of an abstract representation based on the unified model.

5.3.1. Resource mapping from unifying to native data model

In this section we discuss the *reverse mapping* of a resource dr^* of the unifying data model, to the original resource dr from which dr^* has been derived, without considering the access control policies possibly specified for dr within dr^*.

The reverse mapping is achieved by the MapReduce task *remodeler* (see Algorithm 4), which receives as input a resource dr^* of the unifying data model, and derives the data resource dr from which dr^* has been generated. *remodeler* operates recomposing the data units of dr, by aggregation of the unifying resource properties in *urpS*, where *urpS* is the set of unifying resource properties characterizing dr^*.

Algorithm 4: The MapReduce task *remodeler*

MapReduce task *remodeler* **is**
  input : A unifying resource model dr* consisting of a set urpS of unifying resource properties derived from dr
  output: a collection of data units characterizing dr
  (map m)* ↦ (reduce r ▷ finalize f ↱ *updateDu*)*
end

Let *urp* be the unifying resource property analyzed by a single execution of m (i.e., *urp* ∈ *urpS*), the key of the emitted key–value pair is the identifier of the data unit referred to by *urp*, and the value is an object structured according to Definition 1.

On the basis of the specified keys, the emitted pairs are forwarded to distinct execution tracks of the reduce function *r*, in such a way that each execution handles the aggregation of urps contributing to the definition of the same data unit. Therefore, *r* is defined in such a way to receive as input a set of urps that map components of the same data unit, and thus specify the same key (see Algorithm 5 that shows the pseudocode of *r*). The incremental building of the data unit is achieved by defining an initially empty object *du*, and then adding a new field to *du*, for each urp received as input by *r*.

Due to the possible hierarchical structure of data units, and to the unpredictable order with which the urps received as input by *r* are analyzed, *r* may not be able to complete the derivation of the data units. For instance, at a given point of the execution, *r* could analyze a urp that maps a data unit component *duc* specifying as parent component *duc'*, a data unit component of *du* that has not been yet added to *du* structure. Therefore, *r* may not be able to properly position *duc* within *du*. This issue is handled by keeping track of all dangling components in temporary fields of the derived data units, postponing the restructuring of these resources during the finalization phase of the *remodeler* task. The details of this mechanism will be provided later in this section.

Algorithm 5: The reduce by key function *r* of *remodeler*

```
    function r is
        input : a set urpS of urps modeling components of
                the same data unit, the identifier key of the
                referred data unit
        output: a data unit du
(1)     var du:=⊥; tbs:=[]; tbp:=[];
(2)     for urp ∈ urpS do
(3)         if last (π_path(urp)) = key then
(4)             if V ∈ Γ(urp) then
(5)                 setField (du, π_K(urp), π_V(urp));
                else
(6)                 setField (du, π_K(urp), π_id(urp));
(7)                 push (tbs, π_id(urp));
                end
            else
(8)             if last (π_path(urp)) ∉ Γ(urp) then
(9)                 setField (du, last (π_path(urp)), ⊥);
(10)                push (tbp, last (π_path(urp)));
                end
(11)            if V ∈ Γ(urp) then
(12)                setField
                    (π_last(π_path(urp))(du),π_K(urp),π_V(urp));
                else
(13)                setField
                    (π_last(π_path(urp))(du),π_K(urp),π_id(urp));
(14)                push (tbs, π_id(urp));
                end
            end
        end
(15)    setField (du,"tbs", tbs);
(16)    setField (du,"tbp", tbp);
(17)    return du
    end
```

The input parameter *key* of *r* refers the identifier of the target data unit *du* to be derived, whereas *urpS* denotes the set of urps that map the components to be included into *du*. The derivation of *du* considers a single unifying resource property *urp* at a time, from those included in *urpS* (see line 2 of Algorithm 5).

If the last element included in field *path* of *urp* refers to the value of *key*, *urp* models a field at level one of *du* structure.[9] If *urp* includes a field *V* of simple type, a new field is added to *du*, which has the value of *K* as identifier and the value of *V* as value (i.e., $\pi_V(urp)$ - cfr Table 2, see line 5 of Algorithm 5). In contrast, if *urp* does not include a field *V*, as it models a field of complex type, the field added to *du* structure specifies the value of *K* as field name, and the value of *id* as field value (see line 6 of Algorithm 5). Due to the unpredictable processing order of urps operated by *r*, the corresponding component could have been already included in *du* or it could be added in a subsequent stage. The value of field *id* (i.e., $\pi_{id}(urp)$) is thus a placeholder which, during the finalization phase of *remodeler*, will be replaced with the value of a field of *du* having $\pi_{id}(urp)$ as identifier. *r* adds $\pi_{id}(urp)$ to *tbs*, a list of placeholders to be replaced with components at layer one of *du* structure.

In contrast, if *path* does not refer to *du*'s identifier, *urp* does not correspond to a component at the first level of *du* structure, but to one at a deeper level. In this case, the identifier referred to by *path* is related to the object which should include the component modeled by *urp* as field. If no component exists within *du* whose identifier corresponds to the one referred to by *path*, a field with such an identifier is added to *du* structure and it is initialized to an empty object (see lines 8–9 of Algorithm 5). This object is populated on the basis of the information extracted from the other urps analyzed by *r*. In this case, *path* does not refer to a component that will be included at level one of the data unit, but to one which needs to be moved to a deeper level of *du* structure, substituting the respective placeholder specified as value of another *du* field. Thus, *r* keeps track of such an identifier within *tbp*, a variable collecting fields which need to be pruned out from *du* during the finalization phase, once the required substitutions have been performed (see line 10 of Algorithm 5). The component modeled by *urp* is thus added as sub-field of the one referred to by *path*, following the same criteria that have been considered for fields at the first level of *du* structure. Therefore, similar to the previous scenario, *urp* can specify a field *V*, meaning that *urp* models a field with a value of simple type. In this case, a field with identifier *K* and value *V* is added to the structure of the component referred to by *path* (see lines 11–12 of Algorithm 5). In contrast, if *urp* does not specify any field *V*, *urp* models a subfield of *du* characterized by a complex type, and, therefore, a field with identifier *K* is added to the structure of the object referred to by *path*, and initialized to the identifier of *urp*. Like previous case, this value acts as a placeholder to be substituted with an object represented as direct field of *du* (see lines 13–16 of Algorithm 5).

Example 6. Let *urpS* be a set of urps derived from *dr*, the database of emails introduced in Example 4, and let *sUrpS* be a subset of *urpS* composed of urps mapping fields of the email message shown in Fig. 4. All fields, except *headers*, have values of simple types. Let us now suppose that field *path* of the urps modeling *Date*, *From*, *To*, and *Subject* refers to the urp that maps field *headers*. Let us also suppose that field *path* of the urps modeling *body*, *mailbox*, and *headers* refers the identifier of the email. Let us now consider the execution of *remodeler* on *urpS*, focusing on the pairs emitted by *m* for the urps in *sUrpS*. Function *r* incrementally builds an object *du*, representing the considered email. *r* initially defines *du* as an empty object, and then incrementally adds fields representing the above considered components. The analysis of urps whose field *path* refers to *d* causes the inclusion of the corresponding components at level one of *du* structure. The fields

[9] *path* models the list of urps mapping components which, within the tree structure of *dr*, precede the component mapped by *urp*.

body and *mailbox* are initialized with the value of component *V* of the related urps. In contrast, the urp mapping *headers* does not explicitly specify a value. Thus, field *headers* is added to *du* and it is temporarily initialized to *53da*, the identifier of the corresponding urp (see Fig. 4). The string *53da* is then appended to field *tbs* of *du*, which keeps track of the list of placeholders to be substituted. Since field *path* of the urps modeling *Date*, *From*, *To*, and *Subject* refers to *53da*, as soon as *r* analyzes the first of these elements, field *53da* is added to *du* structure. This field is initialized to an object which is then populated with information extracted from the other urps referring *53da* as parent component. Thus, *53da* is incrementally defined as characterized by the fields *Date*, *From*, *To*, and *Subject*, which are set to the simple values extracted from the corresponding urps.

Finally, the finalization function *f* of *remodeler* derives the data units modifying the objects returned by *r*. Algorithm 6 shows its pseudocode.

Algorithm 6: The *finalize* function *f* of *remodeler*

```
    function f is
        input  : the data unit du derived by the execution
                 trace of r preceding f execution
        output: a modified version of the data unit du
                 received as input
(1)     for oid ∈ π_tbs(du) do
(2)         updateDu (du, oid, π_oid(du));
(3)         delField (du, oid);
        end
(4)     for oid ∈ π_tbp(du) do
(5)         delField (du, oid);
        end
(6)     delField (du, "tbs");
(7)     delField (du, "tbp");
(8)     return du;
    end
```

f operates on a single object *du* at a time, executing the substitutions specified within field *tbs*. For each placeholder *oid* of *tbs*, $\pi_{oid}(du)$ refers the replacement value, namely the value of the field of *du* specifying as identifier the value of *oid* (see line 2 of Algorithm 6). The substitution is handled by function *updateDu*, which traverses *du* structure looking for a field whose value matches the value of *oid*. Once such a field is found, *updateDu* reinitializes the field to the value referred to by $\pi_{oid}(du)$. Finally, *finalize* prunes out from *du* the field with identifier *oid*, and once all substitutions have been executed, it deletes *tbs* and *tbp* (see lines 6–7 of Algorithm 6).

Example 7. Let us consider again Example 6, now focusing on the activities performed by the finalization function *f* on the object *du* returned by *r*. By definition of *r* (see Algorithm 5), field *tbs* of *du* collects all the placeholders for which a substitution is required. In the considered scenario, *tbs* only includes the string *53da*, which represents the identifier of a field of *du* specifying as value an object characterized by the fields *Date*, *From*, *Cc*, *To*, and *Subject*. Function *updateDu* visits *du* structure looking for a field which specifies *53da* as value, and, consequently, identifies the field *headers* of *du*. As a consequence, *updateDu* substitutes the placeholder *53da* of *headers* with the value of field *53da*. Finally, *finalize* removes from *du* the fields *53da*, *tbs*, and *tbp* and returns *du*.

5.3.2. The view generation approach

Let dr^* be a unifying resource model derived from a data resource *dr*, which embeds a set of access control policies specified for *dr*. We now discuss the MapReduce task *projector*, which, starting from dr^*, a combining option *co*, a conflict resolution strategy *crs*, a policy propagation criterion *ppc*, a system type *st* (cfr. Table 1), and a set *arc* of parameters specifying an access request context,[10] derives a view of *dr* that points out authorized and unauthorized contents. *projector*, introduced in Algorithm 7, is defined by extension of *remodeler* (see Section 5.3.1), integrating analysis mechanisms into the reverse mapping.

Algorithm 7: The MapReduce task *projector*

```
MapReduce task projector is
    input  : (1) A unifying resource model dr* representing
             a data resource dr, which embeds security
             metadata and policies, (2) a combining option
             co, (3) a conflict resolution strategy crs, (4) a
             policy propagation criterion ppc, (5) the
             considered system type st, and (6) a set arc of
             parameters specifying the access request
             context
    output: a view of dr that shows authorized and
             unauthorized data
    (map m ↱ [ evaluate, combinePs, conflictRes] )* ↦
    (reduce r ▷ finalize f ↱ [updateDu, evaluate, combinePs,
    conflictRes, propagateDCG ↱ handleSP, propagateFCG ↱
    handleSP])*
end
```

In explaining the approach, we rely on functions that allow evaluating policy predicates, handling policy composition and conflict resolution, implementing the options presented in Section 3. Function *evaluate* checks the satisfaction of a policy predicate *pp* of a policy *p* wrt an access request context *arc*. Function *combinePs* combines a set *Ps* of positive/ negative policies specified for a resource *obj*, on the basis of the combining option *co* (see Table 1), and derives a decision by conjunction/ disjunction of policy predicates satisfaction. Function *conflictRes* addresses possible conflicts among positive and negative policies protecting a resource *obj*, on the basis of a conflict resolution strategy *crs* (see Table 1).

Let us start to consider at which point of the reverse mapping the specified policies can be analyzed. Policies specified for urps that map fine grained resources can only be analyzed at the end of the finalization phase, once the resources referred by the policies have been completely recomposed. Indeed, policies specified for a data unit *du* may refer to *du* components, which are correctly positioned within *du* only once the job is completed. As such, *projector* extends *remodeler* in such a way that any derived data unit *du* keeps track of the policies specified for *du* and its data unit components.

On the basis of these considerations, let us now focus on the extensions, introduced in *projector*, to the map and reduce functions *m* and *r* of *remodeler* (see Section 5.3.1).

Function *m* has been enhanced to evaluate policies specified for coarse grained resources. Let *urp* be a unifying resource property analyzed by *m*. If *urp* represents a coarse grained resource, *m* analyzes the policies specified for *urp*, otherwise *m* emits the pair representing *urp*. Policy enforcement is handled by functions *evaluate*, *combinePs* and *conflictRes* which combine the policies specified for *urp* and address possible conflicts on the basis of the specified criteria. Depending on the data model, one or two layers of coarse grained resources may be used by a data management system. For instance, MongoDB adopts a document oriented

[10] The parameters refer concepts of the considered access control model (i.e., for ABAC the subject s and environment e characterizing an access request).

```
{ "_id" : "3e29",
  "value" : {
    "tbs" : ["53da"],
    "tbp" : [ ],
    "meta" :[ {"id" : "body",
     "path" :["3e29"],
     "psSet":[ {"aip":["research,administration"],
        "pip" : ["marketing"]}]}, ... ],
    "pol" : [ { "id" : "body",
     "path" : ["3e29"],
     "psa" : ["s.ap in meta.aip"],
     "psp" : ["s.ap in meta.pip"] }, ... ]
    "body" : "I'm ready, are you?....", ...
    "headers" : "53da",
    "53da" : {
      "From" : "daphneco64@bigplanet.com", ...
```

Fig. 6. Portion of data unit derived from *reduce* of *projector*.

model with 2 layers of coarse grained resources (i.e., database and collection), whereas Redis uses a key–value model with a single layer (i.e., the key-space). *m* keeps track of the derived decisions within global variables, in such a way that the decisions can be accessed during policy propagation. More precisely, variables *dcgl1* and *dcgl2* are used to keep track of decisions related to coarse grained resources at level 1 and 2, respectively, referred to by two additional global variables, denoted *cgl1* and *cgl2*.

The reduce by key function *r* has been extended in such a way to derive a data unit *du*, which, embeds the policies and security metadata that have been specified for the urps received as input. Policy and metadata referring to *du* and *du* components are specified within the fields *pol* and *meta*, which are added to *du* structure. The policies in *pol* are objects composed of the fields *id*, *path*, *psa*, and *psp*, where *id* identifies the protected object, *path* denotes the position of the protected object, whereas *psa* and *psp* refer the predicates of the positive and negative policies specified for *id*. Similarly, security metadata are defined by means of the fields *id*, *path*, and *psSet*, where *psSet* specifies security properties, whereas, *id* and *path* refer the resource to which the properties in *psSet* are bound. *pol* and *meta* are derived from the corresponding fields of the urps received as input by *reduce*.

Example 8. Let us consider again the scenario in Example 6. Fig. 6 shows a partial view of the data unit with identifier 3e29, resulting from the execution of *r* of *projector*. The temporary structure of the data unit shown here still has to be modified during the finalization phase.

Most of the extensions that have been introduced in *projector* are related to the finalization function *f*, which differs from the analogous function of *remodeler* for policy analysis tasks executed once the restructuring phase of the data unit *du* operated by *f* is complete. Three tasks are sequentially executed: (1) the *policy composition* task, which handles the composition of all access control policies specified for *du* and any component *duc* included in *du*, and derives a temporary access decision for any protected resource; (2) the *policy propagation* task, which handles the propagation of access decisions within *du* structure, and derives a definitive decision for any resource within *du* hierarchy; and finally, (3) the *view generation* task, which generates a view of *du* marking any unauthorized component on the basis of the derived decisions. Let us now consider in more details each of these tasks.

5.3.2.1. Policy composition. Let us denote with *du* the data unit generated by an execution trace of *r*, which is provided as input to *f*. For any element *p* within component *pol* of *du*, *f* derives the protection object *obj* of *p* wrt which *p* must be evaluated. *obj* is derived copying the resource referred to by field *path* of *p* and then integrating possible security metadata.

Example 9. Let us focus on the policies specified for the data unit *du* considered in Example 8 and for the related components. In particular, let us consider the policy *p* specified for field *body*. The protection object *obj* derived by *f* maps the object which includes *body* as field, namely the whole content of field *value* of the key–value pair in Fig. 6. Since an element referring the same *path* as *p* is included in field *meta* of *du*, the properties *aip* and *pip*, respectively initialized to *[research, administration]* and *[marketing]* are added to *obj*.

The composition is handled by function *combinePs* and *conflictRes* (see the initial part of Section 5.3.2), which are invoked specifying the protected object *obj*, the set of positive and negative policies included in the fields *psa* and *psp* of *p*, the combining options *co*, and conflict resolution strategy *crs* that have been specified for *projector*. The execution results in an authorization, a prohibition, or, if no positive and no negative policy has been specified within *p*, in an undefined decision. The derived decision is temporary as it does not consider the policies defined for the coarser grained resources that include *obj*. The final decisions will be derived in the latter phase of the analysis, on the basis of the selected propagation strategies. *f* keeps track of the derived temporary decisions within the fields *authS*, *prohS*, and *undefS*, which are added to *du* structure. Such collector fields are used to keep track of the path of the resources whose access, on the basis of the analyzed policies, has been authorized, prohibited or it has not been regulated by any decision. The composition phase terminates once all policies in *pol* have been checked.

Example 10. Let us consider again the scenario in Example 9, and let us suppose that: (i) *any* and *denials take precedence* have been specified as policy combining option and conflict resolution strategy, respectively, and (ii) the access request context *arc* of *projector* refers to a subject attribute *ap* specifying the access purpose *marketing* authorized for subject *s*. The fields *psa* and *psp* of *p* include a single policy, thus, *combinePs* verifies the satisfaction of the corresponding predicates, returning a prohibition for both the policies. No conflict occurs and the access to field *body* is prohibited by the applicable policies. As such *body* is added to *prohS*.

5.3.2.2. Policy propagation. This task handles the propagation of access decisions through the internal structure of *du*. Temporary decisions are reconsidered on the basis of resource hierarchy and the specified propagation criteria. Three criteria are supported: *most specific overrides*, *no overriding*, and *no propagation* (see Section 3). Let *r1* and *r2* be data resources such that *r1* includes *r2*, let *fdr1* be the access decision derived for *r1*, and let *tdr2* be the temporary decision derived for *r2*. Function *handleSP* (see Algorithm 8) derives the final access decision *fdr2* for *r2* from *tdr2* and *fdr1*, on the basis of the policy propagation criteria *ppc*, the conflict resolution strategy *crs*, and the system type *st*.

Let us start to focus on the criterion *most specific overrides*, which propagates the decision of *r1* to *r2* if no decision has been taken yet for *r2* (see Section 3). First, the temporary decision *tdr2* specified for *r2* is checked. If *tdr2* specifies an authorization or a prohibition, the final access decision *fdr2* is set to *tdr2*. Otherwise, if *tdr2* is undefined, *fdr2* inherits the decision *fdr1* specified for *r1*.

According to criterion *no overriding*, if a decision has already been taken for *r2*, the final decisions of *r2* is derived by combining the decisions of *r1* and *r2*, otherwise the access to *r2* is regulated on the basis of *r1* decision (see Section 3). Therefore, if no temporary decision has been taken for *r2*, the criterion *no overriding* is handled like *most specific overrides*. In contrast, if *tdr2*

Algorithm 8: Function *handleSP* of *projector*

```
    function handleSP is
        input : fdr1, tdr2, ppc, crs, st
        output: an access decision
(1)     var fdr2:=⊥;
(2)     switch ppc do
(3)         case most-specific-overrides
(4)             if tdr2=⊥ then return fdr1;
(5)             else return tdr2;
            end
(6)         case case no-overriding
(7)             if tdr2=⊥ then return fdr1;
(8)             else
                    switch crs do
(9)                     case denials-take-precedence return
                        fdr1∧tdr2;
(10)                    case permissions-take-precedence
                        return fdr1∨tdr2;
                    endsw
            end
(11)        case no-propagation
(12)            if tdr2!=⊥ then return tdr2;
(13)            else
                    switch st do
(14)                    case open return permit;
(15)                    case closed return deny;
                    endsw
            end
        endsw
(16)    return fdr2
    end
```

Algorithm 9: Function *propagateDCG* of *projector*

```
    function propagateDCG is
        input : (1) the identifier of the course grained
                resources referred to by cgl1 and cgl2, (2) the
                final access decisions dcgl1 and dcgl2 related to
                cgl1 and cgl2 respectively, (3) a conflict
                resolution strategy crs, (4) a policy propagation
                criterion ppc, and (5) the considered system
                type st
        output: an access decision dec
(1)     var dec:=⊥;
(2)     if dcgl1=permit ∨ dcgl1=deny then dec =dcgl1;
(3)     else
(4)         if st=open then dec =permit;
            else dec =deny;
        end
(5)     if dec =deny then push (unauthCGR, cgl1);
(6)     if cgl2≠⊥ then
(7)         dec =handleSP (dec, dcgl2, ppc, crs, st);
(8)         if dec =deny then push (unauthCGR, cgl2);
        end
(9)     return dec
    end
```

specifies a prohibition or an authorization, the final decision for *r2* is derived from the combination of *tdr2* with *fdr1*, on the basis of the conflict resolution strategy *denials take precedence/permissions take precedence*, specified by *crs*. Finally, the criterion *no propagation* represents the most straightforward case, as it does not require to propagate and combine decisions. In this case *fdr2* only depends on the temporary decision *tdr2* which has been possibly taken for *r2*. If the temporary decision *tdr2* specifies an authorization or a prohibition, *fdr2* is set to *tdr2*. Otherwise, if no temporary decision has been taken for *r2*, the final decision *fdr2* is derived on the basis of the open/close option specified by *st*.

Example 11. Let us suppose that the final access decision *fdr1* related to the data unit *du* in Example 10 is to grant access, whereas the temporary decision *tdr2* for field *body* specifies a prohibition. Let us suppose that *most specific override* has been specified as propagation criterion, whereas *denials take precedence* as conflict resolution strategy. The final decision derived for *body* corresponds to the previously derived temporary decision, and thus to a prohibition.

The proposed propagation approach, which targets a pair of resources, is applied to the resource hierarchy of a data unit *du*. The analysis starts considering the decisions related to the coarse grained resources that include *du*, which have been derived during the mapping phase (see the initial part of this section). Function *propagateDCG* (see Algorithm 9) derives the decision to be propagated to *du* from the access decisions taken during the mapping phase, for the coarse grained resources that include *du*.

Depending on the data model, one or two layers of coarse grained resources may be used. If only one layer is used (i.e., if *cgl2* is null), and *cgl1* refers to an authorization or a prohibition, this becomes the final decision. Otherwise, if the decision is undefined, the final decision is derived from the system type (i.e., open/closed) specified by *st*. If the resulting decision is a prohibition, the resource referred to by *cgl1* is added to the set *unauthCGR*, a global variable which keeps track of unauthorized coarse grained components of the target resource. In contrast, if two layers of coarse grained resources are used (i.e., if *cgl2* is not null), the decision *dec* for the resource at layer one, is provided as input to *handleSP*, which propagates *dec* to the coarse grained resource at layer 2, and returns the combined decision for the resource referred to by *cgl2*, which includes *du*. If the decision is a prohibition, the resource referred by *cgl2* is added to *unauthCGR*.

Example 12. Let us suppose that: (1) the email considered in Example 4 is a document of the collection *messages*, included in the database *emailDB*, (2) no policy has been specified for *messages*, and (3) the policies specified for *emailDB*, evaluated during the mapping phase of *projector* grant the access. Let us suppose that the system is *closed*, and the analysis is performed considering *denials take precedence* as conflict resolution strategy, and *most specific overrides* as propagation criterion. Function *f* handles the propagation invoking function *propagateDCG*. *cgl1* and *cgl2* refer *emailDB* and *messages*, respectively, and, on the basis of previous assumptions, *dcgl1* is set to *permit*, whereas *dcgl2* is *undefined*. Since a decision has been taken for *cgl1*, *dec* is initially set to permit (see line 2 of Algorithm 9), and the decision is propagated to *messages*. The propagation is handled by function *handleSP* (see line 7 of Algorithm 9), which, for the considered parameters, returns *permit*. As a consequence, *propagateDCG* authorizes the access to *emailDB* and *messages*.

The decision derived by *propagateDCG* is in turn propagated to *du* where it is combined with the previously derived decision, resulting in a final decision for *du*. The decision for *du* is then propagated to the data unit components of *du*, where the propagation approach is recursively applied. An additional auxiliary function, denoted *propagateDFG*, is used to propagate

the decisions within *du*, through the depth first visit of *du* tree structure. *propagateDFG* operates on a single resource *r* of *du* tree structure at a time, deriving the final access decision *d* for *r*, on the basis of: (1) the decision *d'* related to the resource *r'* that precedes *r*, and (2) the temporary decision *td* that has been derived for *r* during the execution of the policy composition task (see Section 5.3.2.1). The derived decision *d*, whose computation relies on function *handleSP* (see Algorithm 8), is then propagated from *r* to the resources included in *r*, recursively invoking *propagateDFG* for each resource. *propagateDFG* keeps track of the path of any resource whose final access decision specifies a prohibition within a collector field that is added to *du* structure.

Example 13. Let us consider the scenario analyzed in Example 12. The decision derived from the policies specified for *emailDB* and *messages* is propagated at data unit level, and then, at component level. Let us consider the case of *du*, the data unit representing the email introduced in Example 4, and let us assume that a temporary authorization has been derived starting from the policies specified for *du*. Such a temporary authorization is first combined with the permission propagated from collection *messages* (see Example 12), deriving the authorization to access *du*, and then the derived final decision is propagated to *du* fields, such as to field *body*, where it is combined with the temporary decision derived from the policies specified for this field (see Example 9).

Once this step has been completed, *du* specifies all information which is required to derive a view of the target resource which points out authorized and unauthorized content of the related original resource. The straightforward view generation is achieved by traversing *du* structure by means of a depth first visit, and marking the unauthorized components.

6. Performance analysis

In this section, we describe the experiments we have carried out to empirically assess how efficiently the proposed approach allows assessing the impact of a set of policies on data accessibility. The experiments have been done with MongoDB (ver. 3.4.5 Community Edition), which has been chosen as it natively supports MapReduce, and current surveys rank it as the most popular NoSQL datastore (see http://db-engines.com). However, any other datastore supporting MapReduce might have been used as well. The experiments have been executed on a server equipped with two 16-cores Xeon CPUs, and 128 GB of RAM, using a cluster-based MongoDB deployment with 16 nodes. The adopted MongoDB platform has been configured to use the default WiredTiger Storage Engine, which operates by accessing data stored on disk. We have not used the MongoDB's in-memory storage engine as the majority of NoSQL systems supporting MapReduce do not provide a similar feature, and we aimed at considering a representative scenario for our evaluation. Therefore, in our experiments data are loaded/stored from/on disk.

The integration of the proposed analysis approach in MongoDB has required to provide a Javascript implementation of all *mapping*, *reduce by key*, *finalize* functions, as well as the *auxiliary* functions used throughout the approach's tasks. Any MapReduce task has been implemented as a MongoDB MapReduce query whose *map*, *reduce* and *finalize* functions have been defined by providing a Javascript implementation of the corresponding task's functions. In contrast, auxiliary functions of the approach, once implemented as Javascript functions, have simply been added to the scope of the MapReduce queries. Overall the integration proved to be a straightforward programming activity that has mainly required to convert the pseudo code of the tasks into Javascript code.

Table 3
Analyzed datasets.

Dataset	#du	#duc	F/H	Ho/He
Enron	501 513	10 261 538	H	He
Restaurants	25 359	533 975	H	He
Media	1 769 759	12 388 313	F	He
People	1 000 000	5 000 000	H	Ho
Reddit	904 490	30 836 776	F	He
Stocks	4 308 303	43 083 030	F	He

Six datasets have been used for the experiments: (1) *enron*, a popular dataset of emails; (2) *restaurants*, which stores restaurants reviews; (3) *media*, a catalogue of videos published on youtube; (4) *people*, a dataset of players of an online game; (5) *reddit*, which stores metadata related to different forums; (6) *stocks*, a collection of data related to the stocks market. These datasets are available at: https://github.com/ozlerhakan/mongodb-json-files

In order to equally distribute the documents of these datasets across the 16 nodes of the cluster, a hash-based sharding strategy has been adopted, which, for any document collection, uses the document identifier (i.e., field "*_id*") as sharding key.

Table 3 shows selected features of the considered datasets: the number of data units (#du) and data unit components (#duc), the hierarchical (H) or flat (F) structure of the data units, and the homogeneity (Ho) or heterogeneity (He) of the data units.[11] Each dataset has been mapped to a unifying resource model, by executing the job *unifier* (see Section 5.1).

Due to the lack of policy benchmarks for NoSQL systems, the access to resources collected in the datasets has been regulated by randomly generated access control policies. The policies are purpose based [2], specified with the ABAC notation, and bound to the derived unifying resource models.

Let ds^* be the unifying resource model derived from a dataset *ds*. Policy specification has been achieved by randomly: (1) selecting from ds^* the urps mapping the resources to be protected, (2) deciding the number of positive and negative policies to be specified for each urp, and then (3) generating the policy predicates. The specification has been carried out in such a way that any urp has 0.5 probability to be covered by at least one policy, and, in such a case, it includes from 1 to 3 policies (pseudo uniformly distributed). In addition, any specified policy has probability 0.5 to be positive/negative.

Denoted with Ps_{ds} the set of policies specified for *ds*, the experiments aim at evaluating the time needed for the analysis, with different configurations of access control options, where any configuration specifies: a combining option (co), a conflict resolution strategy (crs), and a policy propagation criterion (ppc). The configurations are summarized in Table 4.

Fig. 7 shows, for any considered dataset *ds* and configuration option cf_i, the average time required for deriving from the unifying resource model ds^* of *ds*, a view of *ds* that points out authorized and unauthorized contents. The proposed measures for each configuration and dataset report the average duration related to 10 analysis processes. In our experiments the access control model has been configured as a closed system.

As visible in Fig. 7, for any dataset, the measured times show a pseudo constant trend for all configurations, with small variations of a few seconds. We therefore believe that analysis options have

[11] A set of data units is considered *homogeneous* when all the data units of the set have the same structure, whereas it is considered *heterogeneous*, when there exists at least a pair of data units with different structures.

Table 4
Configurations of access control options.

cf_{id}	co	crs	ppc
cf_1	Any	Permissions take precedence	Most specific overrides
cf_2	All	Permissions take precedence	Most specific overrides
cf_3	Any	Denials take precedence	Most specific overrides
cf_4	All	Denials take precedence	Most specific overrides
cf_5	Any	Permissions take precedence	No propagation
cf_6	All	Permissions take precedence	No propagation
cf_7	Any	Denials take precedence	No propagation
cf_8	All	Denials take precedence	No propagation
cf_9	Any	Permissions take precedence	No overriding
cf_{10}	All	Permissions take precedence	No overriding
cf_{11}	Any	Denials take precedence	No overriding
cf_{12}	All	Denials take precedence	No overriding

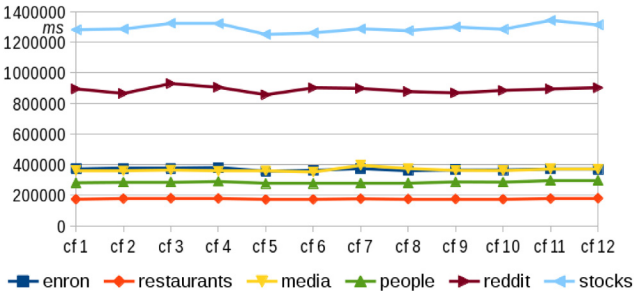

Fig. 7. Policy analysis execution time.

Table 5
The average execution time increment between pairs of configurations that only differ for the conflict resolution strategy option.

Dataset	Average time increment	
	(ms)	(%)
Enron	5501	1,52
Restaurants	4169	2,41
Media	12 978	3,63
People	6249	2,21
Reddit	22 814	2,63
Stocks	33 416	2,61

a small impact on the duration of the analysis process. Task *projector*, for its numerous accesses to data unit components during the recomposition phase, is the main responsible of the measured times. The recomposition has comparable complexity with different configurations. For any data unit, *projector* keeps track of the unauthorizedcomponents (see Section 5.3.2.2). The marking of any unauthorized component requires to traverse data unit structure until the searched element is found. The complexity of view generation is thus affected by the number of components: (i) in the data unit, and (ii) to be marked as unauthorized. Analysis options like *denials take precedence*, which during policy composition favor deny decisions, tend to raise the complexity of the view generation and the related execution time (e.g., see cf_3 vs cf_1 in Fig. 7). In order to better quantify this trend, Table 5 shows, for any dataset, the average increment of execution time which has been observed passing from configurations where *permits take precedence* has been set, to configurations specifying the same access control options but the conflict resolution strategy *denials take precedence* (e.g., cf_5 and cf_7). The growth has been observed with all datasets, with an average increase of 2,5% of the execution time.

The measured times primarily depend on the dataset size. The heterogeneity of the data unit structures has no direct implication on the analysis complexity, which in contrast is affected by the number of layers and components of each data unit. Data units with flat structure can be more easily recomposed than data units with hierarchical structure, as policy propagation is limited to a single layer of analysis. For instance, although dataset *media* has more data units components than *enron*, the observed times for these datasets essentially overlap for any configuration option. In this case the complexity due to the hierarchical structure of *enron* data units (up to 3 layers), has, as counterpart, a higher number of data units and components in *media*.

The measured times show that even with *stocks*, the biggest dataset with over 43 millions data unit components, the analysis is done in 1300 s.

7. Discussion

The proposed approach allows deriving views of the data handled by a NoSQL database, which show, on the basis of the specified access control policies and access control options, the resources accessible in a considered context by a user. The approach supports access control policies defined according to multiple DAC models and configuration options, and it is general enough to be used with all NoSQL systems providing support to MapReduce computation.

On the basis of the derived views, security administrators may decide to restrict or relax the specified policies or to modify the access control options with the aim to grant or deny the access, to given users, to specific portions of the managed data.

The proposed approach could be further extended, by measuring the effects of the specified policies and access control options on the protected data trough a set of metrics, such as for instance (i) the total number of data units that have been marked as unauthorized; (ii) the percentage of data units which, due to the specified policies, cannot be accessed; (iii) the total number of data unit components which have been analyzed; (vi) the number of unauthorized data unit components; (v) the percentage of unauthorized data unit components; and (vi) the average number of data unit components per data unit. The computation of these measures can be easily achieved by complementing the proposed analysis approach with an additional phase, which follows the view generation step. For example, the derivation of the exemplified metrics requires basic grouping operations, that can be straightforward encoded into a MapReduce task.

8. Related work

In the last years, numerous research efforts have been devoted to the study of policy analysis approaches. The research has been primarily oriented to approaches aiming at verifying correctness, supporting policy integration, detecting inconsistencies and redundancies, and reasoning on completeness of policy sets.

Several proposals use formal methods for verifying policy correctness. For instance, a formal specification language for access control policies has been proposed in [20], whereas in [21] a model checking algorithm has been defined on top of the notation proposed in [20] to assess permissions granted by access control policies.

Other contributions rely on Datalog based technologies for analysis purposes. For instance, in [22] a class of Datalog programs is used for the modeling and analysis of Relationship-based Access Control (ReBAC) policies. A Datalog-based approach is also discussed in [23], consisting of a policy specification language for decentralized composite access control systems, and a reasoning framework for the specified policies.

Other work used graph-based analysis strategies. For instance, in [24] an approach is proposed which targets the analysis of category based access control policies. The approach allows deriving properties of the modeled environments, thus easing the verification tasks for security administrators.

A relevant class of policy analysis approaches are those focused on Answer Set Programming (ASP), which translate XACML policies [25] to ASP programs, and use ASP solvers as reasoning tools (e.g., [26,27]). Other work used binary decision diagrams (e.g., [28]), and data mining techniques (e.g., [29]) to identify policy anomalies.

Other proposals (e.g. [30]) have focused on the analysis of policy similarity. For instance, a similarity metric has been proposed in [30], which takes into account categorical and numeric attributes. Lobo et al. [30] claim the practicality of their approach, which allows the efficient identification of similar policies in large policy sets. Policy similarity has also been studied in [31], where an approach to the integration of access control policies has been proposed. However, in this case, rather than deriving similarity measures, policies are classified with respect to the set of requests they authorize.

In the literature, policy integration appears as an extensively investigated analysis dimension. For instance, Rao et al. [32] propose an algebra supporting the specification of integration constraints, and a toolkit built on top of the algebra which targets the integration of XACML policies. A framework, denoted EXAM, has been proposed in [33], which combines different approaches with the aim to provide a comprehensive policy analysis solution. The analysis is achieved by means of SAT solvers and Multi-Terminal Binary Decision Diagrams based techniques. Finally, [34] proposed a tool, called VisABAC, which provides a visual interface to the evaluation of ABAC policies.

All above mentioned approaches aim at analyzing properties of policy sets (e.g., policy similarity) without considering the effects of policies and related access control options on resource accessibility, which is, in contrast, the focus of our paper. Our MapReduce-based approach integrates the implementation of well known conflict management, policy composition, and decision propagation mechanisms (e.g., see [12,17]) finalized at evaluating, for a considered set of policies and related access control options, the accessibility of data handled by NoSQL systems. A key feature of the approach is that it allows analyzing data accessibility at fine grained level operating with heterogeneous schemaless data resources of NoSQL systems that refer to the main data models. In addition, the supported access control policies can be specified according to the main DAC models. To the best of our knowledge, we are not aware of other policy analysis approaches that allow assessing resource accessibility within NoSQL datastores.

9. Conclusions

This paper proposed an approach to evaluate the effects of access control policies on schemaless data within NoSQL data management systems. The proposed approach supports the major existing DAC models, and it can be easily extended to resources modeled through traditional data models (e.g., relational). The proposed approach allows evaluating the effect of a set of policies on the protected resources, which is one of the core analysis services that should be provided to security administrators. Experimental results show the efficiency of the proposed solution, within a variety of scenarios, and even with datasets including millions of data units.

The work described in this paper is progressing in several directions. We are working at an extended version of the proposed analysis approach integrating different metrics which complement the derived views with measures quantifying the effects of the specified policies and access control options on the accessibility of considered data resources along different dimensions (see Section 7). With the aim to enhance the experimental evaluation proposed in this work, we are developing an implementation of the proposed approach for HBase (https://hbase.apache.org), and we plan to consider other popular datastores supporting MapReduce. We are also focusing on extending the framework to be used within federated systems, and we are investigating mechanisms for the XACML-based deployment of access control policies.

As a future work we also plan to investigate an enhanced version of the approach supporting incremental evaluation strategies. This would prevent repeating the whole analysis in case new policies are added to a considered policy set.

CRediT authorship contribution statement

Pietro Colombo: Conceptualization, Methodology, Software, Investigation, Writing - original draft. **Elena Ferrari:** Conceptualization, Methodology, Writing - original draft, Supervision, Project administration, Funding acquisition.

Acknowledgments

This work has received funding from CONCORDIA, the Cybersecurity Competence Network supported by the European Union's Horizon 2020 Research and Innovation program under grant agreement No 830927, and from RAIS (Real-time analytics for the Internet of Sports), Marie Skłodowska-Curie Innovative Training Networks (ITN), under grant agreement No 813162.

Declaration of competing interest

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

References

[1] E. Ferrari, Access Control in Data Management Systems, Morgan and Claypool Publishers, 2010.
[2] J. Byun, N. Li, Purpose based access control for privacy protection in relational database systems, VLDB J. 17 (4) (2008).
[3] P. Colombo, E. Ferrari, Efficient enforcement of action-aware purpose-based access control within relational database management systems, IEEE TKDE 27 (8) (2015).
[4] V.C. Hu, R. Kuhn, D. Ferraiolo, Attribute-based access control, IEEE Comput. 48 (2) (2015).
[5] V.C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone, Guide to Attribute Based Access Control (ABAC) Definition and Considerations, NIST Special Publication 800.162, 2014.
[6] P. Colombo, E. Ferrari, Enhancing mongodb with purpose based access control, IEEE TDSC 14 (6) (2017).
[7] P. Colombo, E. Ferrari, Towards virtual private NoSQL datastores, in: IEEE ICDE, 2016.
[8] J. Longstaff, J. Noble, Attribute based access control for big data applications by query modification, in: IEEE BigDataService, 2016.
[9] P. Colombo, E. Ferrari, Towards a unifying attribute based access control approach for NoSQL datastores, in: IEEE ICDE, 2017.
[10] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, D. DeWitt, Limiting disclosure in hippocratic databases, in: VLDB, 2004.
[11] K. Browder, M.A. Davidson, The Virtual Private Database in Oracle9ir2, Tech. Rep., Oracle Corporation, 2002.
[12] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, P. Samarati, Access control policies and languages in open environments, in: Security in Decentralized Data Management, Springer, 2006.
[13] Dan Sullivan, NoSQL for Mere Mortals, first ed., Addison-Wesley Professional, 2015.
[14] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008).
[15] C. Smith, A. Albarghouthi, Mapreduce program synthesis, in: ACM Conf. on Programming Language Design and Implementation, 2016.
[16] J. Bogaerts, B. Lagaisse, W. Joosen, Sequoia: scalable policy-based access control for search operations in data-driven applications, in: ESSoS, in: LNCS, vol. 10379, Springer, 2017.

[17] P. Bonatti, S. De Capitani di Vimercati, P. Samarati, An algebra for composing access control policies, ACM TISSEC 5 (1) (2002).
[18] R. Cattell, Scalable SQL and NoSQL datastores. SIGMOD Rec., 39(4), 11, 2011.
[19] R. Nehme, H. Lim, E. Bertino, FENCE: Continuous access control enforcement in dynamic data stream environments, in: IEEE ICDE, 2010.
[20] D.P. Guelev, M. Ryan, P.Y. Schobbens, Model-checking access control policies, in: ISC, in: LNCS, vol. 3225, Springer, 2004.
[21] N. Zhang, M. Ryan, D.P. Guelev, Evaluating access control policies through model checking, in: ISC 2005, in: LNCS, vol. 3650, Springer, 2005.
[22] E. Pasarella, J. Lobo, A datalog framework for modeling relationship-based access control policies, in: ACM SACMAT, 2017.
[23] P. Tsankov, S. Marinovic, M.T. Dashti, D. Basin, Decentralized composite access control, in: ETAPS POST, in: LNCS, vol. 8414, Springer, 2014.
[24] Sandra Alves, Maribel Fernández, A framework for the analysis of access control policies with emergency management, Electron. Notes Theor. Comput. Sci. 312 (2015).
[25] eXtensible Access Control Markup Language (XACML) Version 3.0 http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.
[26] G.J. Ahn, H. Hu, J. Lee, Y. Meng, Representing and reasoning about web access control policies, in: IEEE COMPSAC, 2010.
[27] C.D.P.K. Ramli, H.R. Nielson, F. Nielson, XACML 3.0 in answer set programming, in: LOPSTR, in: LNCS, vol. 7844, Springer, 2012.
[28] H. Hu, G.J. Ahn, K. Kulkarni, Discovery and resolution of anomalies in web access control policies, IEEE TDSC 10 (6) (2013).
[29] L. Bauer, S. Garriss, M.K. Reiter, Detecting and resolving policy misconfigurations in access-control systems, ACM TISSEC 14 (1) (2011).
[30] J. Lobo, E. Bertino, P. Rao, R. Ferrini, D. Lin, A similarity measure for comparing XACML policies, IEEE TKDE 25 (9) (2013).
[31] P. Mazzoleni, B. Crispo, S. Sivasubramanian, E. Bertino, XACML policy integration algorithms, ACM TISSEC 11 (1) (2008).
[32] P. Rao, D. Lin, E. Bertino, N. Li, J. Lobo, Fine-grained integration of access control policies, Comput. Secur. 30 (2–3) (2011).
[33] D. Lin, P. Rao, E. Bertino, EXAM: a comprehensive environment for the analysis of access control policies, Int. J. Inf. Secur. 9 (2010) 253.
[34] C. Morisset, D. Sanchez, Visabac: A tool for visualising ABAC policies, in: ICSSP, 2018.

Pietro Colombo is an assistant professor of Computer Science at the University of Insubria (Italy), where he works within the STRICT SociaLab of the Department of Theoretical and Applied Sciences. Dr. Colombo holds a BS, MS and Ph.D. degrees in Computer Science from the University of Insubria, and a 2nd Level Master in Information Technology from CEFRIEL-Politecnico di Milano (Italy). Dr. Colombo's most recent research activities are in the field of access control within NoSQL datastores, privacy aware data management, and data privacy within IoT ecosystems. He has been author of more than 40 scientific papers which have been published in international journals and conference proceedings. Dr. Colombo is also coinventor of 2 US patents.

Elena Ferrari is a full professor of Computer Science at the University of Insubria, Italy where she leads the STRICT SociaLab and is the scientific director of the K&SM Research Center. She holds a MS and Ph.D. degree in Computer Science from the University of Milano (Italy). She received the IEEE Computer Society's prestigious 2009 Technical Achievement Award for "outstanding and innovative contributions to secure data management". Since 2012 she has been an IEEE fellow (for contributions to security and privacy for data and applications), and in 2019 she has been named ACM Fellow. Her research activities are related to various aspects of data management, including data security, privacy and trust, social networks, cloud computing and emergency management. On these topics she has published more than 170 scientific publications in international journals and conference proceedings.