

## **COPYRIGHT WARNING**

This paper is protected by copyright. You are advised to print or download **ONE COPY** of this paper for your own private reference, study and research purposes. You are prohibited having acts infringing upon copyright as stipulated in Laws and Regulations of Intellectual Property, including, but not limited to, appropriating, impersonating, publishing, distributing, modifying, altering, mutilating, distorting, reproducing, duplicating, displaying, communicating, disseminating, making derivative work, commercializing and converting to other forms the paper and/or any part of the paper. The acts could be done in actual life and/or via communication networks and by digital means without permission of copyright holders.

The users shall acknowledge and strictly respect to the copyright. The recitation must be reasonable and properly. If the users do not agree to all of these terms, do not use this paper. The users shall be responsible for legal issues if they make any copyright infringements. Failure to comply with this warning may expose you to:

- Disciplinary action by the Vietnamese-German University.
- Legal action for copyright infringement.
- Heavy legal penalties and consequences shall be applied by the competent authorities.

The Vietnamese-German University and the authors reserve all their intellectual property rights.



Vietnamese-German University  
Department of Computer Science

Frankfurt University of Applied Science  
Faculty 2: Computer Science and Engineering

# Fine-grained access control in NoSQL

Le Minh Thu

Student ID: 13299

Supervisor: Prof. Manuel Clavel

Co-supervisor: Dr. Huong Tran Thi Thu



Vietnamese-German University

BACHELOR THESIS

Submitted in partial fulfillment of the requirements for the degree of  
bachelor of engineering in study program computer science,  
Vietnamese-German University, 2022

January 06, 2022

Binh Duong, Vietnam

## **Disclaimer**

I hereby declare that the information reported in the paper is the result of my own, original, individual work, except where references are made. I also certify that this undergraduate dissertation has not been previously or concurrently submitted for other degrees or other universities, institutions.

Le Minh Thu



## Abstract

Fine-grained access control has always been a security problem when working with databases. Traditionally, in SQL databases, this is done using techniques such as role-based access control using privileges and re-writing queries using views. We introduce one definition of fine-grained access control and show that the current native security support in databases cannot effectively enforce this definition. We show that there is a need for a new combination of policy specification and enforcement. NoSQL-exclusive security problems are also discussed.

## 1 Introduction

Fine-grained access control (FGAC) is a crucial part of databases and information handling. It defines which information the users are able to access and therefore protects the information from unauthorized operations. There are multiple scenarios where FGAC is needed:

- In a university, a student can only view his/her information and grades. A lecturer can view all grades of students whom he/she teaches.
- In a shared cloud storage, a user can view his/her files. He/She can view other people's files only if they access them through links.
- In a company, the intranet only allows access from 9 AM to 5 PM.

We take the definition of FGAC from [1], which states "fine-grained access control policies depend not only on static information, namely the assignments of users and permissions to roles, but also on dynamic information: namely, the satisfaction of authorization constraint on the current state of the system." In our own words, FGAC policies support the granularity up to field-level and respond differently to queries based on the current database state.

NoSQL is a database type that had experienced a significant rise in popularity in recent years. When the term NoSQL is coined in 1998, Carlo Strozzi originally meant databases that do not bear the properties of a traditional SQL database, such as stores data in relations, querying using SQL, etc [2]. Although there is not really a specification of NoSQL, it is commonly accepted that they are schemaless with simple queries and higher performance and scalability. NoSQL has several architectural patterns such as key-value database, document database, graph database, column-oriented database, etc, but we are only concerned document database in this document.

Table 1: NoSQL and SQL comparison

NoSQL	SQL
stores records as key-value pairs	stores records as tables rows
schemaless, data integrity is not always achieved	support a schema and strict data integrity
simpler queries: does not support the SQL-equivalent of JOIN, TRANSACTION, LIMIT and non-index WHERE	complex queries
required known access pattern	
higher scalability and performance	lower scalability and performance

Some well-known NoSQL databases are MongoDB, Amazons DynamoDB, Googles Firestore, Elasticsearch, just to name a few. In this document, we discuss Google Firestore and MongoDB as they are NoSQL databases with different approaches to security.

The outline of this document is as follows:

- Section 2. Apart from a query language, a database needs a policy language to enforce FGAC. We also introduce the attribute-based access control model. Although it is not enough to enforce FGAC, it is a step forward of development from the popular role-based access control model.
- Section 3. We discuss the two models that handle unauthorized access: the Truman model and the Non-Truman model. For each model, the general idea, mechanism, and drawbacks are discussed. The Non-Truman model, although seems to be very tempting on paper, is still under discussion due to the unknown decidability of the validity test.
- Section 5 The two NoSQL databases Google Firestore and MongoDB are discussed and compared based on categories such as granularity, policy language expressiveness, enforcement and performance. Although both databases use attribute-based access control, their approaches to FGAC are fairly different.
- Section 4. For queries that return multiple documents, it is shown that implementing enforcement on the Truman model is easier than the Non-Truman model. Due to the fact that views are a sub-collection of an actual collection, the Truman model can be thought of as multiple filters and these filters can be merged or performed operations on. It is not the same for Non-Truman databases.

- Section 6. We present an existing solution for SQL and identify the changes that need to be made to make it work for NoSQL. We show how the Non-Truman model can be implemented on MongoDB. In particular, due to the schemaless nature of NoSQL manual enforcement, authorization checks are done in a document-by-document and field-by-field fashion.
- Section 7. NoSQL has proven to be more effective in certain scenarios compared to SQL. However, their properties, which put performance and scalability as the highest priorities cause them to suffer from certain security problems that SQL did not have. This sections discuss NoSQL-exclusive problems and mention other interesting problems that are not in the scope of this document.

## 2 Policy language

### 2.1 Overview

Policy languages (often called security languages) are languages that can express the security policies of the applications [3]. Some examples of policy language include access control list (ACL), X.509, Simple Distributed Security Infrastructure (SDSI) [4], etc. Each security language has different strengths and is particularly designed for specific scenarios. For example, X.509 is used for X.500 database access control, ACL is mainly used in file systems, and so on.

One example of ACL in Linux systems is shown below. The policy in addition to the normal user-group-other permissions, allow user `steve` to read and execute.

```
# file: test/declarations.h
# owner: mandeep
# group: mandeep
user::rw-
user:steve:r-x
group::rw-
other::r--
```

Every language supports a model which defines how protected objects, users, and policies interact with each other. In most cases, the model that the language supports contains protected resources, identities, and some other objects that can evaluate the permissions of the caller. The definitions of these objects depend on the access control model and the language being used. They can be roles, relationships, a certain set of properties, etc. Policy

languages provide a convenient, high-level method to achieve security, but they also raise a number of problems such as enforcement and expressiveness, as we will see in section 7.1.

Some examples of access control models are: Discretionary Access Control (DAC), Mandatory Access Control (MAC), X-based Access Control (such as role-based, attribute-based, history-based, identity-based, lattice-based, etc).

## 2.2 Usage

Policy languages are used to limit what a user can do with the database information. However, in practice, they are rarely used directly. In a typical 3-tier application, the middle-tier is usually responsible for authentication and authorization. Part of the reason is that databases do not have a sufficiently expressive policy language that enables the database to protect itself without relying on another component.

Database policy language is becoming more and more important these days as cloud technology to some extent revives the 2-tier architecture: the client and the cloud. In this architecture, the client cannot be trusted as it can be modified, and it runs on hardware that the application owner/developer has no control of.

## 2.3 Attribute-base access control

Since 2018, there is a trend in databases to start implementing attribute-based access control (ABAC) due to the increasing need for fine-grained access control. Databases like Google Firestore, MongoDB, Amazon Dynamo, etc are starting to support ABAC and design a policy language for it in their latest version [5–7]. They do not completely switch to ABAC. Some databases build an ABAC system on top of their old RBAC system result in a hybrid system. For instance, MongoDB still uses RBAC but the process of granting roles is attribute-based.

### 2.3.1 Definition

ABAC is a model that grants or denies a request based on specific attributes of entities and on environmental conditions that could be globally recognized. Policies implemented by this model are limited by the language and

the richness of supported attributes/conditions [8]. An attribute can be user credentials, data value, metadata, etc. Ideally, these attributes do not need to be input by the system admin. In this document, 'ABAC languages' is used to refer to languages that support the ABAC model.

Some example policies:

- Allow access from 9 AM to 5 PM
- Allow access if the user is older than 18 or is a staff
- Allow access if the document state is public

### 2.3.2 Comparison to Role-based access control (RBAC)

A brief comparison to role-based access control (RBAC) is made because RBAC is currently a very common access control model for databases. RBAC is the intuitive solution to the access control problem. In a way, ABAC can be thought of as a natural development of RBAC and RBAC can also be considered as an ABAC model where the system evaluates user permission based on a limited set of, namely, pre-defined attributes roles and groups.



Vietnamese-German University

In both models, the handling of the new user is very flexible. When a new user/resource is created, there is no need to adjust the policies. The system only needs to inject the attributes (roles in the case of RBAC) into that new user/resource.

#### **Role-based access control**

- Easier to implement, defining roles is simple and fast, easy to manage
- Role explosion in the case everyone's permissions is different from each other.

#### **Attribute-base access control**

- Harder to implement, needs pre-define attributes and specified policies
- More fine-grained, more flexible

ABAC languages have a very big potential in this trend of development since they are fine-grained, flexible, and easy to manage. It is also very convenient to write field-level policies with ABAC.



### 2.3.3 Sample policy

Following are some samples of ABAC policy languages in MongoDB and Google Firestore. Different databases vary in granularity. For the sake of examples, document-level policies will be used so all databases can support it.


Policy 1: allow to view post if post's state is public or user is author of the post

Firestore policy language

```
service cloud.firestore {
  match /databases/{database}/PATH_TO_POSTS/{postId} {
    allow read: if resource.data.state == 'public' || resource.data.
      author == request.auth.uid
  }
}
```

MongoDB policy language.

```
{
  "%or": [
    { "state": "public" },
    { "author": "%user.id" }
  ]
}
```



Policy 2: allow a lecturer to view a record of a student if that lecturer teaches that student (assuming that each student object has an array keeping the ID of their lecturers)

Firestore policy language

```
service cloud.firestore {
  match /databases/{database}/PATH_TO_STUDENTS/{studentId} {
    allow read: if role == 'lecturer' && request.auth.uid in
      resource.data.lecturers
  }
}
```

MongoDB policy language

```
{
  "%user.id" : {
    "%in" : "lecturers"
  }
}
```

It can be seen that the developer is limited by what the languages supports, such as operators (`%or`, `||`, `%in`, etc) and objects (`request.auth.uid`, `%%user.id`, etc). Firestore policy language is similar to Javascript while MongoDB policy language follows JSON format. The current scenario of ABAC is each database comes up with their own format. The policies are not really reusable and must be re-written if a developer moves from a database to another.

Among the policy languages, eXtensible Access Control Markup Language (XACML) was proposed as the standard for ABAC policy languages and some applications did implement it. However, the language is XML-based and is low-level, which makes it hard for a human to write/manage the policy. Also, it does not support collection queries (queries which are expected to return multiple records). In general, XACML does not seem to be a fit for today's application.

For now, there is still no standardized policy language for ABAC but the general approach is clear, namely databases need to define attributes/operations and design a language where the policies can be expressed using those attributes and operations.

## 3 Enforcement Vietnamese-German University

Currently, there are two models, namely the Truman model and the Non-Truman model, representing different ways for databases to respond to unauthorized access.

### 3.1 Truman model

For every query, the Truman model transparently filters out the unauthorized data from the result of the query and returns the result to the user [9]. The name came from the movie "The Truman Show" where Truman is a person who lives in a fake reality created by the film crew. Similarly, the user does not know his/her query has been modified and believes that what the database returns is the truthful result of the query, which it is not.

Example 1. Given a NoSQL database instance with the collection "Movies" contains 3 records:

```
{name: "Frozen", rating: "General", review: 1.6},  
{name: "Ice Age", rating: "General", review: 2.6},
```

{name: "13 reasons why", rating: "Restricted", review: 3.6}

and the policy: Under-13-year-old online users can only view movies with "General" rating.

Suppose that user Abe is 12 years old. Abe makes a query <get all the movies in the database>. Truman databases would return

{name: "Frozen", rating: "General", review: 1.6},  
{name: "Ice Age", rating: "General", review: 2.6}

Example 2. This example shows field-level policy. We add the policy to the above example: Movies review is public if the value is greater than 2.5

In this case, with the same query <get all the movies in the database>, Truman model would returns

{name: "Frozen", rating: "General"},  
{name: "Ice Age", rating: "General", review: 2.6}

### 3.1.1 Mechanism of Truman model

The Truman model enforces security using a view called authorization view. An authorization view is a view created using user credentials as parameters, specifies the data that the user is authorized to read [9].

Although the authorization view can be created using a query (see [9] and section 7.1), it should be thought of as an abstract idea of a set/multiset of records that the user is allowed to view rather than an actual concrete database view. In practice, it is expensive to create a view for every incoming query so that databases can have a different implementation to achieve this same idea.

Instead of querying against the database, the Truman model querying against the authorization view.

In the case of Example 2, Abe's authorization view for the "Movies" collection contains

{name: "Frozen", rating: "General"},  
{name: "Ice Age", rating: "General", review: 2.6}

and his query <get all the movies in the datase> will be modified to <get all the movies in the authorization view> and therefore will return the result

in Example 2.

Example 3. If Abe is querying <get all movies with "review" equals to 1.6 from database>, Truman databases would return an empty set of results because the query has been transparently modified to <get all movies with "review" equals to 1.6 from Abe's authorization view>. Although there is actually a record that has review value equals 1.6 in the database, Abe cannot query/see it because he is not authorized.

### 3.1.2 Problem of Truman model

We have seen how the Truman model protects its data using the authorization view. Since the authorization view only contains what the user can see, querying from an authorization view can guarantee the user only see their authorized data.

However, the Truman model re-writes the query without the users knowing it and that makes the users believe what they see is the truthful result of the query.

Example 4. In this particular example, if Abe wants to count the total number of movies, he can query <count all movies in the database> and get 2. There are 3 movies, but only 2 of them are available to Abe, so he gets the result of 2. Abe does not know his query has been modified, so he believes the database has 2 movies, which is the 'fake reality' that has been mentioned before. If Abe compares his result with another adult user who made the same query, he will notice the inconsistent results.

This shows that the result returned by the Truman model can be misleading.

## 3.2 Non-Truman model

The Non-Truman model solves the problem of misleading results by either executing the query with no modification or rejecting/throwing errors if the database detects any unauthorized access.

In the case of example 1, databases implementing the Non-Truman model would throw an error saying the user is not authorized to make such a query. The reason is Abe is trying to access an unauthorized document {name: "13 reasons why", rating: "Restricted", review: 3.6}. It can be seen that different from the Truman model, the Non-Truman model does not automatically filter out unauthorized data.

In the case of example 3 and 4, the Non-Truman model would reject the query because the user is not authorized to see/count all movies.

Example 5. If the query is authorized, both models behave the same. If Abe queries <get names of movies with "general" rating>, both models would return {name: "Frozen"}, {name: "Ice Age"}

### 3.2.1 Mechanism of Non-Truman model

The Non-Truman model enforces access control using a validity test, a test to check if the query is valid. A valid query is defined as a query that can be answered only using authorized data. Failing the test, the query is rejected. Passing the test, the query is executed with no modification.

The Non-Truman model also uses an authorization view, but it is for the validity test rather than re-writing queries. For each query  $q$ , if the Non-Truman model can construct an equivalent query  $q'$  when querying against the authorization view, then the query  $q$  is valid. (It is phrased differently across multiple papers: the query is valid if it can be answered using only the authorization view/if it can be rewritten against the authorization view/if an equivalent query  $q'$  against the authorization view can be constructed.) The definition of "equivalent" is taken from [9], which means  $q$  and  $q'$  produce the same results for all database instances.

Example 6. If Abe makes the query  $q$  <get movies with "general" rating and review greater than 2.5 from database>. The equivalent query  $q'$  can be constructed as <get movies from authorization view>. Therefore,  $q$  passes the validity test and is executed with no modification.

[9] also mentioned the terminology unconditionally valid and conditionally valid. In short, unconditionally valid means the query is valid for all database instances, and conditionally valid queries are only valid for a number of database instances.

For example, Abe's query  $q_1$  <get name of movies with "general" rating> will always be valid no matter the content of the current database state, because Abe is allowed to view such data. Therefore, the query  $q_1$  is unconditionally valid.

However, suppose the current database "Movies" collection contains 2 records:

```
{name: "Frozen", rating: "General", review: 2.6},  
{name: "Ice Age", rating: "General", review: 2.6},
```

The query `q2` <get all movies in the database> is valid for this state, but it will be unauthorized in example 1 of section 3.1. Therefore, the query `q2` is conditionally valid, which means it only valid for some of the database instances.

### 3.2.2 Problem of Non-Truman model

**Restricted queries** The Non-Truman model solves the problem of misleading results in the Truman model. The result, in the case the query is accepted, is the same for everyone because the query is executed as it is. However, this leads to another problem, namely the queries in the Non-Truman model are more restrictive.

The Non-Truman model does not filter out unauthorized data. For the query to be accepted, the user must add a filter, which somehow, expresses his/her permission.

Example. Suppose there is a collection `Post` with the policy that allows access if the user is the author of the post. Since the query result is the same for every caller (in valid cases), the user must inform the database he/she is the author of the posts by adding the condition `author = $userId`, where `$userId` is replaced by the real `userId` value of the user. This requires the users to be aware of the policies every time they query.

**The validity test** Determining the validity of a query is not simple. If an equivalent is not found, it is unknown whether there is no such query. The decidability of the problem, and defining an algorithm to construct the equivalent query, is still under discussion [9–12].

As has been said before, unconditionally valid means the query is valid for all database instances, and conditionally valid queries are only valid for a number of database instances. For unconditional validity, the problem can be reduced to the old problem of rewriting queries using views and has already been proven to be decidable for conjunctive queries [9, 10, 13]. For the general case, it is undecidable [13].

For conditional validity, it is still unknown. Also, there is no algorithm yet that is complete for bag semantics.

The Non-Truman has been discussed a lot throughout the years, but still very few databases decided to implement it. Although it can handle most general queries, the inference rules and validity test has yet to cover all pos-

sible cases which lead to unpredicted behavior in different implementations. This raises the question of its practicality in real-life applications.

### 3.2.3 Comparison

In computer science, convenience and security have always been a trade-off.

The Truman model has higher data availability, higher independence between the query and the policy but the result is misleading. In the example, the issue is not very serious, but in automated processes where information is handled in a pipeline fashion (the output of one calculation is the input of another calculation), the errors could be accumulated over time and the consequences could be severe.

On the other hand, the Non-Truman model result is consistent but it is secure in the sense that it will lock out most queries and the queries are highly dependent on the policy. One small change in the policies might lead to a lot of queries getting rejected. On the bright side, the result can be trusted as truthful.



## 4 Challenges of FGAC-Non-Truman database

For the Truman model, some databases do not actually generate a view for authorization. It is because views are very costly for performance and most of the time databases would find another solution. For ABAC models, when a specific document is requested, it is easy to evaluate the policy and check for the attributes. But in the case where multiple documents are requested, it is not efficient to evaluate the permission for every single document. One solution that Axiomatics Reverse Query (ARQ) came up with is they will work out a "filter" from the policy and use that filter to directly query against the database [14].

Here is how it works. For every document in collection C, the policy says allow access if condition X is satisfied. (X might be a conjunctive or disjunctive combination of multiple sub-conditions). From that policy, ARQ can resolve that user A has the permission to view documents having property X'. After that, they combine the original query with this newly made filter, e.g. get documents having X' property from C and passed it directly to the database. For Truman databases, this is a great convenience. They achieve the same behavior (filtering out unauthorized data) without using views.

It is not the same for Non-Truman. The first problem is that if the database returns the result using a filter, it will never throw errors. In particular, if nothing matches the filter, an empty set is returned. A Non-Truman database needs to be able to detect unauthorized access and throw an error.

Another thing is that ABAC policies cannot take into account the database state. As stated above, ABAC evaluates permissions from attributes of user/request/resource (whose value is known when the query is made) or from global variables (such as current time, environment, partition, etc). If the information that needs to be checked is not inside the targeting documents, ABAC does not have an attribute to refer to it. This leads to the problem of not being able to verify the validity of conditionally valid queries when their validity depends on the current content of the database. For example, a policy such as a user can read the average star rating of all reviews if there are at least 100 reviews cannot be done using ABAC policy language.

The problem could be solved using NoSQL denormalization. Normally, NoSQL is encouraged to use duplicate data as their queries are not as complex as SQL queries (no cross-collection queries). It also sometimes makes sense to store duplicate data when the information is expensive to calculate. However, for this problem being mentioned here, the database structure is changed in order to make the policy works and that should not be the case. Non-Truman already has queries that are very attached to their policies. If the database structure is also dependent on the policy, changing the policy, which happens quite often, would result in adjusting too many things.

Although ABAC languages cannot achieve FGAC, its syntax combination (operator - attribute/variable - value) is very flexible and promising.

## 5 Comparison study: Google Firestore and MongoDB

### 5.1 Granularity

Firestore offers document-level policies while MongoDB offers up to field-level policies.



## 5.2 Expressiveness

### 5.2.1 Policy language

Firestore implements an ABAC system. MongoDB uses an RBAC system, but the process of granting roles is attribute-based. For an attribute-based system, a list of pre-defined attributes and operators are needed. Both databases have a language (or text format) to express those attributes and operators. For sample policies, see section 2.3.3

Because the attributes are pre-defined, these databases cannot take into account the database state. Both have workarounds, but in general, security rules alone cannot support permissions that take into consideration the database state.

Table 2: List of attributes and global variables

Google Firestore	MongoDB
request (path, method, data, auth)	%%request
request.auth (id, email, etc)	%%user (id, types, data, custom data, etc)
resource (id, data, path)	fields: call by names document: using 'all fields' flag
	%%values
	%%environment
	%%partition

Table 3: List of operators

Google Firestore	MongoDB
function()	%function
&&	%and
—	%or
exist()	%exist
in	%in
== / != / > / >= / < / <=	%eq / %neq / %gt / %gte / %lt / %lte

### 5.2.2 Query language

In general, MongoDB query language support is greater than Firestore's.

Table 4: Query tools supported

	Google Firestore	MongoDB
Querying with indexed filter	✓	✓
AND in 'WHERE-clause'	✓	✓
OR in 'WHERE-clause'		✓
Nested subqueries		✓
Aggregate operator	count	✓
SQL-Join		

### 5.2.3 Enforcement

Firestore's enforcement falls into the category of Non-Truman as stated in their documentations: 'Rules are not filters' [5]. Users cannot write a query for all the documents in a collection and expect Firestore to return only the documents that the current client has permission to access.

MongoDB, on the other hand, implements the Truman model. Although it is not mentioned in their documentation, this can be determined based on the way MongoDB reacts to unauthorized access.

Vietnamese-German University

### 5.2.4 Performance

Consider a collection with documents containing 2 fields, namely name and age. All documents have their age-value greater than 18. The query consisting counting the number of records whose age value is greater than 25. The policies are: names are public and ages are public if their values are greater than 18. The scenario is set so: (1) all access are authorized and both databases can return the result and a comparison can be made, (2) while we know all queries are valid, both databases still have to evaluate the permission since there are policies.

Table 5: Database response time test

	Firestore	MongoDB
1000 documents	about 500ms	about 800ms
10000 documents	about 500ms	about 2500ms

Although the number of documents is relatively small for a database, it shows that Firestore outperforms MongoDB in response time. This is not

totally surprising: Firestore sacrifices some functionalities (in particular, the query language support is simpler) that hindrance its speed in order to achieve high performance.

### 5.2.5 Firestore Non-Truman

Firestore does not completely implement the Non-Truman model. To avoid the unusual issue regarding conditional validity, the Firestore team has decided that the database only accepts unconditionally valid queries [15]. Firestore also bears the disadvantage of the Non-Truman model's restricted queries.

## 6 Workaround

Since there are still no algorithms that can correctly and completely handle the validity test, a possible solution is to perform the authorization check manually. For each incoming query, the result is not returned immediately but goes through an 'authorization test'. This test takes each record as an input and evaluated its content against the policies. If the content of any record does not match, the test returns an error. If no test triggers an error, the result is returned.

The general workflow of [1] is as follows:

1. The schema, policies, and SQL endpoints are defined in models.
2. A code generating tool takes the models as input and generates authorization functions and stored procedures.
3. The stored procedure corresponding to the required query (defined in models) is called using endpoints.

Although in the paper, FGAC is implemented in an SQL database, the same can also be done on a NoSQL database. It is more simple to handle NoSQL queries due to their lack of cross-collection (SQL-Join) queries. Stored functions, although is not as powerful as store procedures, but are enough to implement FGAC.

Another significant difference is the fact that NoSQL schema may not have a schema. In SQL, the method can go to the schema and get the list of attributes to be checked. For NoSQL, the method would need to generate a list

of protected fields for a collection. For each field in that list, check whether the targeting document has that field. If yes, the system will evaluate the permission using the corresponding policy.

## 6.1 Workflow

### 6.1.1 Define a security model

In the first step, a security model is defined. This security model defines which and how a resource is protected by the action upon that resource and the condition to be authorized. Figure 1 shows a security model for the following policies:

- A lecturer can read their own name and age.
- A lecture can read any student's name
- A lecture can read the age of a student if he/she teaches that student

  
Figure 1: Sample security model  
Vietnamese-German University

```
[
  {
    "roles": [
      "lecturer"
    ],
    "actions": [
      "read"
    ],
    "resources": [
      {
        "collection": "lecturers",
        "field": "name"
      },
      {
        "collection": "lecturers",
        "field": "age"
      }
    ],
    "auth": "doc._id == callerId"
  },
  {
    "roles": [
      "lecturer"
    ],
  },
]
```

```

    "actions": [
      "read"
    ],
    "resources": [
      {
        "collection": "students",
        "field": "name"
      }
    ],
    "auth": "true"
  },
  {
    "roles": [
      "lecturer"
    ],
    "actions": [
      "read"
    ],
    "resources": [
      {
        "collection": "students",
        "field": "age"
      }
    ],
    "auth": "callerId in doc.lecturers"
  }
]

```



Vietnamese-German University

A mapping between the queries and the endpoints is also defined.

Figure 2: Query and endpoint mapping

```

[
  {
    "name": "getAllLecturers",
    "query": "lecturers.find()"
  },
  {
    "name": "getAllLecturesAge",
    "query": "lecturers.find({}, {age:1})"
  },
  {
    "name": "getAllLecturersAgeLessThan30",
    "query": "lecturers.find({age:{$lt:30}})"
  }
]

```

### 6.1.2 Generate artifacts

In the next step, the program will parse through the security rules to construct a list L1 of protected fields for each collection. Similarly, the program will parse the query and construct a list L2 that contains the projecting fields and the fields in the filter of the query. These are the fields that gives the user the information and therefore need to be checked.

For each fields in L2, an authorization function is generated. It is a boolean function taking in the user credentials and the requested document. This function returns true if the user is authorized to perform the action on the corresponding field.

For example, suppose the query is <get names of lecturers whose age is 25>. Then two authorization functions will be generated. The user needs to be authorized to access both 'name' field and 'age' field for the query to be accepted. Although 'name' is the only field that got returned, the user also needs to be authorized to access 'age'. Because for every document that is returned, the user can work out from the query that the value of 'age' is 25, therefore it also needs to be protected.

In the case the query asks for all fields, L1 and fields in the filter are used instead. Since NoSQL databases do not have a schema, for each document, we check if it contains a protected field, which means an authorization function is responsible for it. If yes, we call the authorization and check if the user is authorized.

Figure 3 shows how stored functions can safely execute a query in a Non-Truman way, in pseudo-code for readability. For the actual functions generated in javascript, see A.

Figure 3: Pseudo code of generated stored function

```
function auth_check_att1 (callerId, role, doc) {
//return true if user is authorized
  switch (role) {
    case 'role1': return auth_condition_11; break;
    case 'role2': return auth_condition_12; break;
    ...
    default: return false;
  }
}

function auth_check_att2 (callerId, role, doc) {
//return true if user is authorized
  switch (role) {
    case 'role1': return auth_condition_21; break;
```

```

        case 'role2': return auth_condition_22; break;
        ...
        default: return false;
    }
}

function endpoint_name (callerId, role) {
    result = collection.find(filter).toArray()
    result.forEach( function(doc) {
        if ( att1 in doc && !auth_check_att1(callerId, role, doc) )
            throw new Error('Not authorized!')
        if ( att2 in doc && !auth_check_att2(callerId, role, doc) )
            throw new Error('Not authorized!')
        ...
    } )
    return result;
}

```

### 6.1.3 Execute safely the query

Finally, the stored functions are deployed on to the MongoDB clusters. The functions can be called directly by the client, or the client can use the mapping from step 1 to call the appropriate stored functions.

## 7 Related problems

### 7.1 Policy language and querying language mapping

The enforcement of the policy depends crucially on the query language, as it is the language that constructs the view.

Example. For the authorization view to get the authorized data of Abe, the querying language needs to support two policies: (1) values of "rating" is "General", (2) only show "review" if the value is greater than 2.5.

For SQL, the view can be built with

```

CREATE VIEW AuthorizationView AS
SELECT name, rating, CASE review>2.5 WHEN TRUE THEN review ELSE
    mask_value() END AS rating
FROM Movies
WHERE rating="General"

```

Because SQL is restricted with a schema, all records in the result set must have the same number of attributes. To suppress an attribute, a mask value must be used. Mask value is a placeholder that informs the existence of an attribute but hides its value.

For NoSQL, since there is no schema, to suppress a field, it only needs to be removed entirely. Following is a MongoDB example in javascript.

```
var pipeline = [
  {$project: {
    name: 1,
    rating: 1,
    review: {
      $cond: {
        if : { $gt : [ "$review", 2.5 ] },
        then: "$review",
        else: "$$REMOVE"
      }
    }
  }}
]
collection.aggregate(pipeline)
```

In this particular case, to enforce field-level policy, the query language must support an if-then-else-like syntax in the projection clause of the query. The problem, however, is not limited to enforcing field-level policy. Generally, if the query language is not as powerful or as expressive as the policy language, the policy cannot be enforced entirely. However, this problem is for future discussions and in this document, it is assumed that the policy is translated into the query language entirely.

## 7.2 NoSQL security and SQL security

NoSQL scalability and performance makes it more suitable for application in certain scenarios. However, they suffer from lack of support for security compared to the traditional approach of SQL.

- **Granularity and schemalessness:** As has been said in 6, policy specification and enforcement in NoSQL does not have a prior assumption based on the schema like in SQL. Documents in NoSQL are not bounded with a pre-defined structure, therefore the fields and field types are unknown. At the point of enforcement, it has to be manually checked whether a record falls under the responsibility of a policy.
- **Language:** For now, NoSQL still have yet to have a standardized universal querying language. This makes development and studying



NoSQL more difficult. It is also inconvenient in switching between databases as the user has to learn new semantics, syntax, etc.

- **Performance and security:** SQL has very strict constraints but it is not the same for NoSQL. With proper policy enforcement, a lot of work needs to be done. Some databases will make a reasonable security sacrifice as performance is the reason NoSQL is popular in the first place.

[16] discusses some other problems but above are the main ones we are concerned with. NoSQL is convenient in the case of scalability and when read happens more often than write. It is also faster to setup a prototype since there is no need of defining data models and constraints. Nonetheless, security in NoSQL is still a fairly new topic.

## 8 Conclusion

We have addressed the problem of FGAC. We show that the ABAC model which databases are moving to cannot enforce FGAC due to its limitation on pre-defined attributes and global variables. Two models for authorizing are described - the Truman and Non-Truman model. The study of the Non-Truman model's set of conference rules and validity test is still not complete which is why database providers are still hesitant to implement them. We present a workaround to show it can be done using existing tools. Related problems including mapping from policy language to querying language and security problems in NoSQL are also mentioned.

## References

- [1] H. N. P. Bao and C. Manuel, "A model-driven approach for enforcing fine-grained access control for sql queries," *Springer Nature Computer Science*, 2021.
- [2] C. Strozzi, "Nosql: A relational database management system," *Lainattu*, vol. 5, p. 2014, 1998.
- [3] J. DeTreville, "Binder, a logic-based security language," in *Proceedings 2002 IEEE Symposium on Security and Privacy*. IEEE, 2002, pp. 105–113.
- [4] R. L. Rivest and B. Lampson, "Sdsi-a simple distributed security infrastructure." *Crypto*, 1996.

- [5] “Writing conditions for Cloud Firestore Security Rules — Firebase documentation,” 2019, <https://firebase.google.com/docs/firestore/security/rules-conditions>.
- [6] “Rule Expressions MongoDB Realm,” n. d., <https://docs.mongodb.com/realm/rules/expressions/>.
- [7] “IAM JSON policy elements: Condition - AWS Identity and Access Management,” 2013, [https://docs.aws.amazon.com/IAM/latest/UserGuide/reference\\_policies\\_elements\\_condition.html](https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements_condition.html).
- [8] V. C. Hu, D. Ferraiolo, R. Kuhn, A. R. Friedman, A. J. Lang, M. M. Cogdell, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone *et al.*, “Guide to attribute based access control (abac) definition and considerations (draft),” *NIST special publication*, vol. 800, no. 162, pp. 1–54, 2013.
- [9] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy, “Extending query rewriting techniques for fine-grained access control,” in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 2004, pp. 551–562.
- [10] Z. Zhang and A. O. Mendelzon, “Authorization views and conditional query containment,” in *International Conference on Database Theory*. Springer, 2005, pp. 259–273.
- [11] B. van Velden, J. Voorbij, and L. Breure, “Authorized access to dynamic spatial-temporal data using the truman model,” *Department of Information and Computing Sciences. Utrecht University, Utrecht*, 2007.
- [12] K. Salem, “Fine-grained database access control,” 2007.
- [13] A. Y. Halevy, “Answering queries using views: A survey,” *The VLDB Journal*, vol. 10, no. 4, pp. 270–294, 2001.
- [14] “Blimey! Whats Axiomatics Reverse Query? - Axiomatics,” 2013, <https://www.axiomatics.com/blimey-what-s-axiomatics-reverse-query/>.
- [15] “Securely query data — firebase documentation,” n. d., [https://firebase.google.com/docs/firestore/security/rules-query#queries\\_and\\_security\\_rules](https://firebase.google.com/docs/firestore/security/rules-query#queries_and_security_rules).
- [16] P. Colombo and E. Ferrari, “Fine-grained access control within nosql document-oriented datastores,” *Data Science and Engineering*, vol. 1, no. 3, pp. 127–138, 2016.

## A Source code for section 6

Figure 4: Generated Javascript stored functions

```
// read_lecturers_name
exports = function(callerId, role, doc){
  switch (role) {
    case "lecturer":
      return doc._id == callerId;
  }
  return false;
};

// read_lecturers_name
exports = function(callerId, role, doc){
  switch (role) {
    case "lecturer":
      return doc._id == callerId;
  }
  return false;
};

// getAllLecturers
exports = async function(callerId, role){
  const lecturers = context.services.get("mongodb-atlas").db("test
  ").collection("lecturers");
  let res = await lecturers.find().toArray()
  .then(function(result) { result.forEach(function(doc) {
    if (!context.functions.execute("read_lecturers_name",
    callerId, role, doc)) throw "Not authorized!"
    if (!context.functions.execute("read_lecturers_age",
    callerId, role, doc)) throw "Not authorized!"
  })
  return result;
})
  return JSON.stringify(res);
};
```

Figure 5: Generating code using Javascript EJS

```
var policy = require("./policy.json")
var endpoint = require("./query-endpoint.json")
var parser = require("./queryParser.js")
var ejs = require("ejs")

function readPolicy() {
  var arr = []
  var roles = []
  var protected = []
```

```

policy.forEach(function(p) {
  p.roles.forEach(function(ro) {
    roles.push(ro)
    p.actions.forEach(function (a) {
      p.resources.forEach(function (re) {
        if (re.collection in protected) {
          protected[re.collection].push(re.field)
        } else {
          protected[re.collection] = [];
          protected[re.collection].push(re.field)
        }
      }
      arr.push({
        role: ro,
        action: a,
        collection: re.collection,
        field: re.field,
        auth: p.auth
      })
    })
  })
})
roles = Array.from(new Set(roles)) // remove duplicate
return {arr, roles, protected};
}

function readEndpoint() {
  return endpoint
  .map(function(q) {
    return {
      ...q,
      ...parser.parse(q.query)
    }
  })
  .map(function(r) {
    r.fields = r.projectionField
    if (!(r.fields && r.fields.length)) { // if not an array or
      empty array, do not process
      r.fields = readPolicy().protected[r.collection]
    }
    return r
  })
}

let roles = readPolicy().roles

readPolicy().arr.forEach(async function(p) {
  let temp1 = await ejs.renderFile("template_helper1.ejs", {...p,
    roles})
}

```

```

    console.log(temp1)
  })

readEndpoint().forEach(function (q) {
  ejs.renderFile("main_procedure.ejs", q, function(err, str) {
    if (err) throw new Error(`\nCODE: ${err.code}\nMESSAGE: ${err
      .message}`)
    console.log(str)
  })
})
})

```

Figure 6: EJS templates

```

// authorization function template
function <%= action %>_<%= collection %>_<%= field %> (callerId,
  role, doc) {
  switch (role) { <% for (var i=0; i<roles.length; i++) { %>
    case "<%= roles[i] %>":
      return <%= auth %>;
    <% } %>
  }
  return false;
}

// main stored function template
async function <%= name %>(callerId, role) {
  const <%= collection %> = context.services.get("mongodb-atlas").
    db("test").collection("<%= collection %>")
  let res = await <%= query %>.toArray()
  .then(function(result) { result.forEach(function(doc) { <%
    for (var i=0; i<fields.length; i++) { %>
    if (<%= fields[i] %> in doc) {
      if (!context.functions.execute("<%= action %>_<%=
        collection %>_<%= fields[i] %>", callerId, role,
        doc)) throw "Not authorized!"
    }<% } %>
  }}})
  return JSON.stringify(res);
}

```