Towards an Effective Attribute-based Access Control Model for Neo4j

Adil Achraf Bereksi Reguig[1], Houari Mahfoud[1], and Abdessamad Imine[2]

Abou-Bekr Belkaid University & LRIT Laboratory, Tlemcen, Algeria[1]
Lorraine University and Loria-Cnrs-Inria, Nancy, France[2]
`{adilachraf.bereksireguig, houari.mahfoud}@univ-tlemcen.dz`
`abdessamad.imine@loria.fr`

Abstract. The graph data model is increasingly used in practice due to its flexibility in modeling complex real-life data. However, some security features (e.g., access control) are not receiving sufficient attention from researchers since that graph databases are still in their infancy. Existing access control models do not rise to the finest granularity level of data, use expensive methods for data filtering or explicitly enforce access control rules in application code which may lead to several data security breaches. Based on the most popular graph database system Neo4j and its query language Cypher, this paper provides an Attribute-based Access Control (ABAC) support to Neo4j which makes the model more fine-grained and allows to specify more expressive access policies. Next, we provide a rewriting algorithm that transforms an arbitrary Cypher query into a safe one that enforces the underlying access control policy by returning only authorized data. Contrary to most existing solutions that use non-practical query languages, the proposed solution can be integrated easily within the Neo4j database system.

Keywords: Graph Database · Cypher · Neo4j · Access Control · ABAC · Query Rewriting

1 Introduction

Several works in the literature showed that the graph data model provides more flexibility, efficiency and scalability when it comes to deal with large and highly interconnected data. Graph database systems (e.g. Neo4j) are being used and studied by both researchers and practioners for many applications such as fraud detection, knowledge discovery, business analytics, recommendation systems, data integration and cleaning. Traditional access control models proposed for relational data cannot be adapted for graph data. Indeed, fulfilling the main requirements of access control (i.e. fine-grained, context management and efficiency for preventing non-authorized data access) is still a challenge in graph data type context due to it schema-less characteristic and the lack of standards neither for data model nor for query language, which makes the design of a graph access control model more intriguing.

We consider the most popular graph database system, Neo4j[1], and its query language Cypher [8]. Neo4j provides a Role-Based Access Control (RBAC) model which is not suitable enough for applications that require fine-grained control over data access and permissions for different resources in the graph.
One scenario that could highlight a limitation of this model is when dealing with dynamic authorization rules. For example, in an Electronic Health Record system, if the role *Doctor* allows reading patient *Health Record* (*HR*), so each doctor will be granted to access any patient HR even though this latter is not being treated by him. This limitation raises a need of an access control model that considers attributes of the context.

Therefore, we propose an extension of the Neo4j RBAC model by allowing specification of more expressive access rules that can be applied to the finest granularity level of data. Precisely, our extension incorporates attributes to access authorization of Neo4j which yields to an ABAC model. Our model takes all advantages of the Neo4j model, overcomes its limits, and allows access policies to be specified based on information of the user and the data being accessed. Based on a large class of Cypher queries, we propose a rewriting algorithm, we enforce our access policies by rewriting any Cypher query into a safe one that returns only accessible data.

Contributions and Road-map. The main contributions of this paper are as follows: *1)* We give a thorough study of the Neo4j access control model and we show its limits (Section 2); *2)* We propose an ABAC model for graph databases that aims to overcome limits of the Neo4j model (Section 3); *3)* By considering a practical fragment of the Cypher query language, we provide a rewriting algorithm that translates queries from this fragment into safe ones (Section 4); *4)* We conduct an experimental study to show effectiveness and efficiency of our approach (Section 5).

Related Work. Table 1 compares the most important access control models for graph databases. These models differ in:
Data type. Several access control models have been proposed for graph modeled data including especially works designed for RDF triplestores [2] and those for *Property Graph* (*PG*) [3, 9, 11, 12, 14]. Unlike models for RDF graphs which focus on semantic relationships [15], those for PGs require to handle fine-grained permissions to nodes, edges and properties level.
Access Control Model (ACM). Several graph database systems (e.g. Neo4j, TigerGraph) provide a RBAC model [10, 12] which assigns permissions to users based only on their roles. This model lacks granularity when it comes to specify permissions for a specific user or level of the data. To overcome this limit, several works have combined the RBAC with other ACMs: e.g. [2] with *Mandatory Access Control* (*MAC*); [15] with *Mandatory* and *Discretionary Access Control* (*DAC*); [9] with *Attribute-Based Access Control* (*ABAC*). Some studies [11, 14] focused on *Attribute* and *Relation-Based Access Control* (*ReBAC*) that naturally express the

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

Graph	**Work**	**ACM**	**(-)**	**Granularity**	**Reinforcement**
RDF	J. Chabin et al. [2]	RBAC+MAC		node,rel	Query Rewriting
PG	S. Rizvi et al. [14]	ABAC+ReBAC		node,rel	Query Rewriting
	A. Mohamed et al. [11]	ABAC+ReBAC	✓	node,rel	Brute-force
	C. Morgado et al. [12]	RBAC		node	Brute-force
	M. Valzelli et al. [15]	DAC+MAC +RBAC	✓	node	Views
	S.Clark et al. [3]	ReBAC	✓	node,rel	Not mentioned
	D.Hofer et al. [9]	RBAC+ABAC	✓	node,rel	Query Rewriting
	Neo4j ACM [1]	RBAC	✓	node,rel,attr	At Query time
	Our work	RBAC+ABAC	✓	node,rel,attr	Query Rewriting

Table 1: Comparative table of related works.

connections and associations between a subject and a resource, enabling intuitive access decisions based on their inherent relationships [3].

Fine-grained level. Nodes, relationships and attributes are entities that compose any PG. However, existing ACMs focus only on nodes and relationships [3,9,11,14], with some designed solely for nodes [12, 15]. Some works [12, 14] allow definition of only positive permissions. The Neo4j system allows definition of unconditional positive (+) and negative (-) access permissions to all entities of the PG.

Reinforcement mechanism. The commonly used mechanisms are: *i*) *Brute-force* which involves either executing a query as-is or rejecting it if it contains any unauthorized element; *ii*) *Two-steps* by fetching the data generated by the query, then filtering the outcomes before they are returned to the user; *iii*) *Views* allows the administrator to build personalized views that represents accessible data only; and *iv*) *Query rewriting* where queries are modified to include access control filters prior to their execution. The rewriting principle has been recently considered for graph databases: [14] transforms user's query into another query that returns authorized data only, while [9] introduces access control filters to the methods of the Neo4j Object Graph Mapper framework.

Our work differs from existing ones in: *i)* we allow controlling access to each entity of the PG; *ii)* we propose expressive access control policies with negation, conditions and several kinds of access privileges; *iii)* we use the query rewriting principle that has demonstrated high efficiency for relational data [4] as well as graph data [14]; and *iv)* we propose a practical solution by considering a large class of Cypher queries and by showing how to integrate it within Neo4j, the most popular graph database system.

2 Background

We next discuss the Cypher query language as well as the Neo4j RBAC model.

2.1 Cypher Queries

For the sake of clarity and along the same lines as [14], we consider a specific subset of the Cypher query language that includes the most commonly used features[2]. The syntax of a Cypher query Q is given as follows:

Q `::=` $\mathcal{M}\ \mathcal{W}\ \mathcal{R}$
$\mathcal{M}$ `::= Match` α `|` $\mathcal{M}\ \mathcal{M}$
$\mathcal{W}$ `::=` ϵ `| Where` β
$\mathcal{R}$ `::= Return` φ
α `::=` α' `|` α'`,`α
α' `::=` (m) `|` (m)`-[`m`]->`α' `|` (m)`<-[`m`]-`α'
m `::= v | v{`δ`} | :l | :l{`δ`} | v:l | v:l{`δ`}`
δ `::= a:val |` δ`,`δ
β `::=` β `AND` β `|` β `OR` β `| NOT` β `| Exists{ Match` $\alpha\ \mathcal{W}'$ `} | (v.a op val)`
$\mathcal{W}'$ `::=` ϵ `| Where` β'
β' `::=` β' `AND` β' `|` β' `OR` β' `| NOT` β' `| (v.a op val)`
φ `::= v | v.a |` φ`,`φ

A Cypher query consists of three types of statements: *Match* ($\mathcal{M}$), *Where* ($\mathcal{W}$) and *Return* ($\mathcal{R}$) statements. In essence, the *Match* statement defines the patterns to search for in the data graph. The *Where* statement adds conditions to these patterns. Upon finding matches for these patterns in the data graph, the *Return* statement specifies the data components to be presented to the user. Each Cypher query contains one or more *Match* statements, each declared using the *Match* clause. A single *Match* statement (Match α) is formed by one or more path patterns (α'). A path pattern is a sequence of nodes and/or relationships. A node (m) (or a relationship [m]) is defined by a label l, an optional variable v, and an optional list δ of attribute conditions. Notably, all nodes and relationships are labeled. If a variable v appears in a *Match* statement without an associated label, it is assumed that the label of v has been declared in a previous *Match* statement. Optional *Where* statements can follow *Match* statements to filter the patterns. The *Where* statement applies filtering using simple conditions (e.g., v.a op val) that operate on attributes within the *Match* statements. These conditions employ operators (=, <>, <,<>,<=,>=) or built-in functions (contains, starts with, ends with). Complex conditions, referred to as path conditions, are path patterns intersecting with the *Match* statement's patterns to filter nodes/relationships. These complex conditions are expressed using the Exists function, which validates the existence of specified path patterns. We adopt a simplified syntax of the Exists function for clarity. Finally, the *Return* statement is employed to present nodes, relationships, and/or attributes that are linked to the *Match* statements.

Example 1. A property graph is given in Figure 1 to represent a health information system. Nodes represent entities such as *Doctor*, *Health Records* (HR) and *Event*, while edges represent relationships between these entities. We consider the following Cypher queries:

[2] It is worth noting that our findings can be extended to cover other features of the language as well.

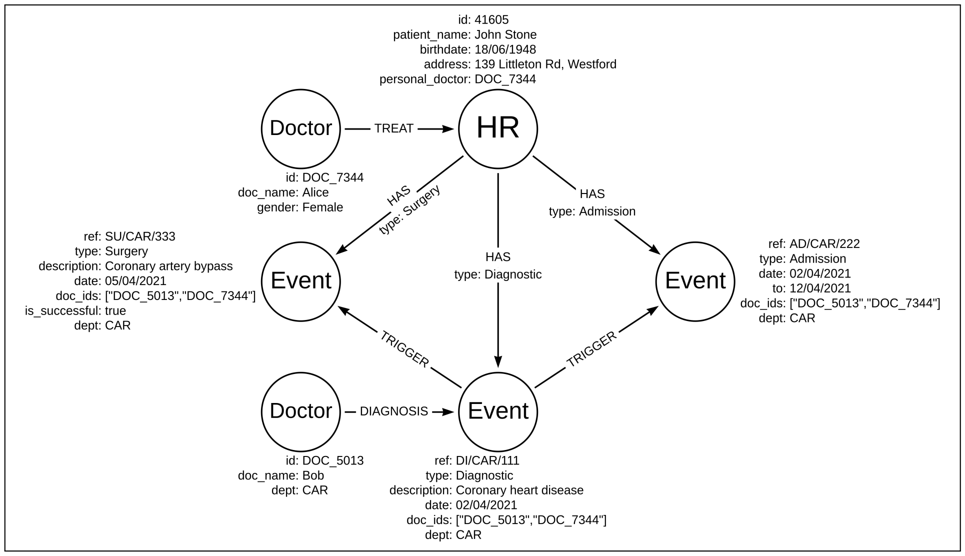


Fig. 1: Property Graph sample.

```
Q1 : Match (d:Doctor) Where d.gender='female' AND Exists{
        Match (e:Event{Description:'Coronary heart disease'})
        Where d.ID IN e.doc_ids
    } return d.doc_name

Q2 : Match (hr:HR{patient_name:'John Stone'})-[r:HAS]->(e:Event),
        (e)<-[r2:DIAGNOSIS]-(d:Doctor)
        return d
```

$Q1$ returns names of woman doctors which are involved in events related to coronary heart disease. $Q2$ provides all information of doctors who were involved in diagnosing the patient named *John Stone*. □

2.2 The Neo4j Access Control Model

We next discuss the Neo4j Role-Based Access Control model.

Syntax & Semantics. ACPs created within Neo4j are *closed*, which means that all entities of the underlying data graph (i.e. nodes, relationships and attributes) are by default unauthorized. For each user role, the administrator can grant/deny privileges to some entities. Three kinds of read privileges are supported:

1. Traverse: this privilege allows the user query to traverse nodes and/or relationships of the data graph without examining their attributes.
2. Read: this privilege allows the user query to access to attributes of some nodes and relationships of the graph.
3. Match: having this privilege on some node (resp. relationship) is semantically the same as having both Traverse and Read privileges on this node (resp. relationship). That is, this privilege allows the user query to traverse an entity as well as to read its attributes.

```
ACP ::= ACR | ACR, ACP
ACR ::= T | R | M
T := (Grant|Deny) Traverse on Graph G (Nodes|Relationships) E to R
E := * | E'
E' := l | l , E'
R := (Grant|Deny) Read{A} on Graph G (Nodes|Relationships) E to R
A := * | A'
A' := a | a , A'
R := (Grant|Deny) Match{A} on Graph G (Nodes|Relationships) E to R
```

Fig. 2: Grammar of the Neo4j Access Control Model.

Neo4j allows definition of positive and negative permissions, specified respectively with the clauses Grant and Deny. Given the above, an *Access Control Policy* (*ACP*) can be written, as a set of *Access Control Rules* (*ACRs*), following the grammar of Figure 2. Despite the privilege type, each *ACR* specifies the graph G, the user role R and the set E of entities (i.e. nodes and relationships) for them the privilege will be applied. The entities are specified either by *, i.e. all nodes/relationships are concerned; or a finite set of nodes/relationships labels. In addition, an *ACR* with Read or Match privilege specifies in addition a set A of attributes (or * for all attributes) that can(not) be read.

Notice that a denied Match privilege is a little bit confusing. Its semantics depends on whether a concrete attributes set is specified or it is just *. A denied Match with a concrete attributes set specifies that those attributes cannot be read while the corresponding entities can be traversed. However, if a denied Match is specified with * as attributes set, then both traversing the entities and reading their attributes are denied.

Example 2. Consider the following *ACRs* specified for *Doctor* role:

```
GRANT MATCH{patient_name,age} ON GRAPH * NODES HR TO Doctor
DENY MATCH{address} ON GRAPH * NODES HR to Doctor
DENY MATCH{*} ON GRAPH * NODES HR to Doctor
```

The first one allows traversing all nodes labeled HR as well as reading their attributes patient_name and age. This ACR can be written differently as follows:

```
GRANT TRAVERSE ON GRAPH * NODES HR TO Doctor
GRANT READ{patient_name,age} ON GRAPH * NODES HR TO Doctor
```

The second ACR denies access to attribute address of nodes labeled HR. The third ACR denies traversing nodes labeled HR as well as reading their attributes. □

As any access control model, conflicts may arise when opposite permissions are given for the same entity. In such a case, deny privileges take precedence over grant privileges. More details about this resolution are given in Section 3.2.

Reinforcement Mechanism. Neo4j reinforces its access control policies dynamically at query time. Precisely, when the user asks the execution of a Cypher query, the Neo4j's query planner establishes the most efficient way to execute this query by using index, caches as well as access control privileges of the system.

That is, entities requested by the query are found efficiently and are also filtered to keep only authorized ones. This process differs from *Brute-force* approaches in the sense that the query is not rejected if it attempts to find some unauthorized entities. Instead, the system purifies the output, delivering solely the authorized entities that were originally requested by the query.

Limits. As any RBAC model, an ACP defined within Neo4j for some role will be applied for all users who are assigned that role, while assigning a personalized policy for each user may lead to role explosion and complicated administration of those policies [6]. Moreover, many real-life scenarios often require to consider not only user information, but also specific contextual factors and properties of the resources being accessed. Such fine-grained access control is necessary to accurately reflect the intricacies of different situations and ensure appropriate security measures. Furthermore, access control policies may be relatively big in practice [16] and must be written within a simplified syntax so that the administrator could express exactly what she/he needs. However, as mentioned before, a denied Match privilege may lead to confusion, and its semantics will be more difficult to understand when considering update rights.

3 Our Access Control Model

We propose an ABAC model for Cypher queries that keeps the expressive power of the Neo4j's model and enhances it with conditional permissions.

3.1 Syntax & Semantics

We consider the grammar of the Neo4j's access control model (Figure 2) and we make two fundamental changes. Firstly, we add the following rules:

```
T := (Grant|Deny) Traverse on Graph G (Nodes|Relationships) E to R WHERE C
C := C AND C | C OR C | NOT C | (C) | @a op val | $a op val
```

Where `a` is an attribute, `val` is a value and `op` is an operator or a built-in function as described for the Cypher syntax. Intuitively, our additional rules allow defining granted/denied traverse permissions based on some conditions. Notice that `@a` refers to attribute of the data graph while `$a` refers to an attribute of the user/context (e.g. user degree, department number). Moreover, `val` is a value belonging either to the data graph or the user/context. Such conditions allow taking into account properties of the data graph, the user and/or the context where expressing permissions. It is worth noting that we do not assign conditions to read privileges in order to avoid circular permissions[3] which may lead to ambiguous, inconsistent and vulnerable policies. That is why we assign conditions only to Traverse privileges.

[3] An attribute A is accessible depending to the value of an attribute B and vice versa.

/////		**Grant Traverse**		**Deny Traverse**		**Deny Read**
		uncond.	**cond. c2**	**uncond.**	**cond. c2**	
Grant Traverse	**uncond.**	Yes	Yes	No	not (c2)	/////
	cond. c1	Yes	c1 or c2	No	c1 and not(c2)	
Deny Traverse	**uncond.**	No	No	No	No	
	cond. c1	not(c2)	c2 and not(c1)	No	not(c1 or c2)	
Grant Read						No

Table 2: Conflict Resolution

Secondly, we omit the use of the Match privilege as it may lead to confusion and obfuscated semantics as explained in Section 2. This will not hinder practicability of our approach since the semantics of Match can be expressed using Traverse and Read privileges.

Example 3. We consider the data graph of Example 1 and we define two user roles (*Doctor* and *Administrator*) with different permissions. A *doctor* has access to (a) all his information; (b) events in which he is involved and (c) health record of all patients that he treats. An *Administrator* is authorized to access (a) all health records without their attributes; (b) all events with their associated attributes except 'doc_ids' attribute; and (c) relationships 'HAS' having a property type equals to 'Surgery'. These permissions are expressed in the same order via the following ACRs:

```
GRANT TRAVERSE ON GRAPH * NODES Doctor TO Doctor WHERE @ID=$doctorID
GRANT TRAVERSE ON GRAPH * NODES Event TO Doctor WHERE $doctorID in @doc_ids
GRANT TRAVERSE ON GRAPH * NODES HR TO Doctor WHERE @personal_doc=$doctorID
GRANT TRAVERSE ON GRAPH * NODES HR TO Administrator
GRANT TRAVERSE ON GRAPH * NODES Event TO Administrator
GRANT READ{*} ON GRAPH * NODES Event TO Administrator
DENY READ{doc_ids} ON GRAPH * NODES Event TO Administrator
GRANT TRAVERSE ON GRAPH * RELATIONSHIPS HAS TO administrator WHERE @type='Surgery'
```

Where the use of @ (resp. $) refers to attribute of the data graph (resp. system). □

3.2 Conflict Resolution

As our approach is an extension of the model proposed by Neo4j, we adopt the same conflict resolution mechanism as Neo4j in which negated access take precedence over granted access. Details about our conflict resolution rules are given in Table 2 where *Yes* (resp. *No*) specifies that the entity is accessible (resp. inaccessible) after resolving conflicts. In addition, the accessibility of such entity may be defined by combining many conditions of the access control policy.

4 Policy Reinforcement Mechanism

The query rewriting mechanism has been largely considered for different kind of data [4,5,7,13]. Recently, some authors have adopted this mechanism to reinforce access control policies over graph databases [9,14]. Given an access control policy

Algorithm 1: SafeCypher(Q, P, ur)

```
   Input: A Cypher query $Q$ , an ACP $P$, and a user role $ur$.
   Output: A safe version of $Q$, $Q^s$ or $\emptyset$ otherwise.
1  $G^P$ := accessGraph($P$);
2  $Q^s := Q$;
3  Add a meta-variable to each free node (resp. relationship) in $Q^s$;
4  Extract a set $S$ of Match statements in $Q^s$;
5  Define a function $L$ that returns the label of each variable in $Q^s$;
6  Define a function $T$ that returns the type of each variable in $Q^s$;
7  Let $AC$ be a list of attribute conditions belonging to the Where statement of $Q^s$;
8  Let $R$ be a list of statements returned by $Q^s$;
9  $ACFilters$ :='true';
10 foreach $s \in S$ do
11     $sub$ := isSubQuery($s, Q^s$);
12     $s'$ := singleMatchRewriter($s, sub, G^P, ur$);
13     if ($sub = true$) then
14         if ($s' = \emptyset$) then Replace $s$ with $false$ in $Q^s$;
15         else if ($s' \neq s$) then Replace $s$ with $s'$ in $Q^s$;
16     else if ($s' = \emptyset$) then return $\emptyset$;
17 if ($ACFilters \neq$'true') then
18     if ($AC \neq \emptyset$) then Add $ACFilters$ to the Where statement of $Q^s$;
19     else Create a Where statement in $Q^s$ with $ACFilters$ as condition;
20 foreach $ac \in AC$ with the form "$v.attr$ $op$ $val$" do
21     $rp$ := attributeReadCheck($G^P, ur, L(v), attr, T(v)$) ;
22     if ($rp = false$) then Replace $ac$ with null in $Q^s$ ;
23 foreach $r \in R$ with $r = v$ or $r = v.attr$ do
24     $ga$ := getGrantedAttrs($G^P, ur, L(v), T(v)$);
25     $da$ := getDeniedAttrs($G^P, ur, L(v), T(v)$);
26     if ($r = v$ AND ($da.contains(*)$ OR $ga = \emptyset$)) then Replace $r$ with $v\{\}$ in $Q^s$ ;
27     else if ($r = v$ AND $ga.contains(*)$) then Replace $r$ with $v_{|da}$ in $Q^s$ ;
28     else if ($r = v$ AND $!ga.contains(*)$) then Replace $r$ with $v\{ga\}$ in $Q^s$ ;
29     else if ($r = v.attr$) then
30         $rp$ := attributeReadCheck($G^P, ur, L(v), attr, T(v)$);
31         if $rp = false$ then Replace $r$ with null in $Q^s$;
32 $Q^s$ := optimizeQuery($Q^s$)
33 return $Q^s$;
```

Fig. 3: An algorithm for safe rewriting of Cypher queries.

P and a user query Q, it is to rewrite Q into a safe one, Q^s, by considering rules of P, so that: for any data graph G, $Q^s(G)$ will traverse and return only accessible entities (nodes, relationships and attributes). We describe hereafter our query rewriting algorithm designed for Cypher queries and Neo4j system.

Our algorithm, referred to as SafeCypher, is given in Figure 3. It inputs a Cypher query Q, an access control policy P, and the role ur of the user requesting Q. The algorithm produces a safe version of Q, Q^s, or $\emptyset$ if no safe version of Q can be generated. First of all, it is clear that a textual ACP is hard to analysis. That is, we construct a graph representation of our ACP P, called *access graph* and denoted by G^P. Such access graph allows checking easily the accessibility of any entity requested by the query. Due to space limit, we report all details about this structure to appendix A. Once the access graph G^P is constructed (line 1), meta-variables are generated for each free element[4] of the query (line 3). All elements composing Q are extracted (lines 4-8) including: Match statements

[4] I.e. node or relationship with no variable attached to it.

belonging both to the body of the query and its Where statement; the Return statement; and all attribute conditions defined within the Where statement. Each match statement s is rewritten separately using the procedure singleMatchRewriter (line 12). If s is a subquery[5] of Q and its safe version s' is not empty then s is replaced by s' in the original query, if it is empty then it is simply replaced by *false* in the original query (lines 13–15). Otherwise, if s is not a subquery then the algorithm returns an empty query as a fundamental part of Q cannot be made safe (line 16). After checking all elements belonging to Match statements of Q, their corresponding accessibility conditions are grouped as a conjunctive formulas *ACFilters*. This latter is added to the Where statement of the query to ensure the accessibility of each entity parsed/returned by Q (line 18). If Q has no Where statement, then it is created to enforce the computed accessibility conditions (line 19).

Next, the algorithm examines each attribute condition belonging to the Where statement (lines 20–22) and checks whether the corresponding attribute is accessible. Precisely, for each attribute condition $v.attr\ op\ val$, the algorithm calls the procedure attributeReadCheck in order to check whether the attribute $attr$ is accessible via the element referred to by the variable v. If this is not the case then the attribute condition is replaced with NULL in the query.

On the other side, each return statement rt is rewritten (lines 23–31) in order to check the accessibility of each entity the query Q wants to return. Each return statement can be either a node/relationship (i.e. a variable v) or attributes of this later (i.e. $v.attr$). The procedure getGrantedAttrs is called to get a list ga of all granted attributes (line 24). Similarly, the procedure getDeniedAttrs is called to get a list da of all denied attributes (line 25). Using these lists, all attributes being returned by Q are checked. If an attribute is inaccessible w.r.t P and it is explicitly returned by Q (i.e. case of $v.attr$) then it is replaced by null at the original query (lines 29–31). Moreover, if Q returns implicitly a list of attributes (i.e. case of v) then this list is sanitized to keep only accessible attributes. We denote by $v_{|_{da}}$ a safe rewriting of v that returns the corresponding entity (node/relationship) without the unauthorized attributes. In addition, $v\{ga\}$ refers to granted attributes of that entity. Thus, $v\{\}$ returns the entity without any of its attributes (lines 26–29). However, Cypher language does not allow to remove a list of attributes from a node making part of the query result while returning the remaining accessible attributes. So, the syntax $v_{|_{da}}$ is implemented using APOC procedures provided by Neo4j. A use-case is given at Example 4.

Notice that adding accessibility conditions to the different elements of Q may lead to a safe but unoptimized version of Q. For instance, the condition null *and* null may be replaced by null. That is why the algorithm calls finally the procedure optimizeQuery to returns an optimized safe version of Q.

The procedure singleMatchRewriter (Figure 4) aims to check the accessibility of each entity composing some Match statement in input. It extracts vertex (resp. relationship) tokens composing M, as well as attribute conditions defined over

[5] According to our Cypher syntax, a subquery is any Match statement appearing inside an Exists call.

vertices and relationships of M. A token has the form $v : l$ while an attribute condition has the form $\{attr : val\}$. The procedure uses two other procedures: elementTraverseCheck checks accessibility of vertex/relationship (more details are provided in appendix B; while attributeReadCheck checks accessibility of attribute. If an element (i.e. vertex or relationship) is not accessible then the procedure returns $\emptyset$ (line 7). If such element is conditionally accessible, then its corresponding accessibility condition c will be added to a global filter *ACFilters* (line 8). If the element is unconditionally accessible then no change is done. Furthermore, if an attribute *attr* is not accessible then $\emptyset$ is returned (lines 14–18). The final safe version of M is finally returned (line 19). Given a subquery composed by an Exists call, and a Match statement s and a Where statement w inside it. Then, accessibility conditions related to s are added to w rather than the Where statement of the whole query (lines 10–13). That is why *subACFilters* is used to define accessibility conditions of subqueries only, while *ACFilters* adds accessibility conditions to the Where statement of the entire query.

Example 4. Consider a Cypher query Q made by an administrator and its safe version Q^s computed w.r.t the access policy defined in example 3:

```
Q: Match(hr:HR) where Exists{
      Match(hr)-(x:HAS)->(e:Event)
      Where date='15/08/2020'
   } return hr,e
```

```
Qs: Match(hr:HR) where Exists{
       Match(hr)-(x:HAS)->(e:Event})
       Where(date='15/08/2020'
       AND x.type='Surgery')
 } return hr{},apoc.map.removeKeys(e,["doc_ids"])
```

The safe version Q^s first adds an accessibility condition, i.e. x.type='surgery', as the administrator can access to only relationships of type *surgery*. The variable e refers to events and thus it is refined by Q^s in order to return only accessible attributes of events: i.e. all attributes excepting docs_ids. The function apoc.map.removeKeys(e,["doc_ids"]), from Neo4j APOC library, is used to remove the unauthorized attributes (doc_ids in this case) from the returned element e. In addition, administrator has a granted traverse for nodes hr but not a granted read, that is no attribute in node hr can be returned. Thus, Q^s replaces hr by hr{} at query result. □

5 Experimental study

Using real-life data, we next conduct several experiments in order to check efficiency and scalability of our model compared to that of Neo4j.

5.1 Experimental setting

Datasets. We use the *Stack Exchange*[6] dataset which is a vast collection of anonymized questions, answers and user interactions from various *Stack Exchange* websites. It contains 8 node labels; 7 relationship types representing interaction between nodes; and a vertex set varying from 85K to 1.3M.

[6] https://archive.org/details/stackexchange

Algorithm 2: singleMatchRewriter(M, sub, G^P, ur)

```
Input: A match statement $M$, a boolean $sub$, an access graph $G^P$, and a user role $ur$.
Output: A safe version of $M$, $M^s$, or $\emptyset$ otherwise.
1  Let $VT$ (resp. $RT$) be list of vertex (resp. relationship) tokens in $M$;
2  Let $AC$ be list of attribute conditions defined over elements of $M$;
3  $subACFilters$ := 'true';
4  $M^s := M$;
5  foreach $t \in (VT \cup RT)$ with $t = v : l$ do
6      $ac$ := elementTraverseCheck($G^P, ur, v, l, T(v)$);
7      if ($ac = false$) then return $\emptyset$ ;
8      else if ($ac = c$ and $sub \neq true$) then $ACFilters := ACFilters \wedge ac$;
9      else if ($ac = c$ and $sub = true$) then $subACFilters := subACFilters \wedge ac$;
10 if ($subACFilters \neq$'true') then
11     if (there exists $ac \in AC$ with the form "$v.attr$ op $val$") then
12         Add $subACFilters$ to the Where statement of $M^s$;
13     else Create a Where statement in $M^s$ with $subACFilters$ as condition;
14 foreach $ac \in AC$ do
15     $rp$ := attributeReadCheck($G^P, ur, L(v), attr, T(v)$) ;
16     if ($rp = false$) then
17         if $ac$ with the form "$v.attr$ op $val$" then Replace $ac$ with null in $M^s$;
18         else if $ac$ with the form "$\{attr : val\}$" then return $\emptyset$;
19 return $M^s$;
```

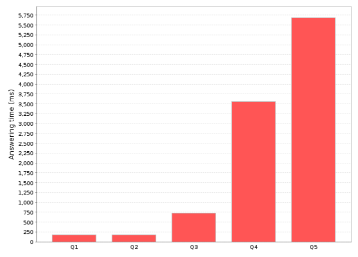
Fig. 4: Procedure to safely rewrite Match statements.

Implementation. To ensure the integration of our model within Neo4j, we first implemented a Java application that inputs an access control policy in textual form (i.e. via the syntax given in Section 3), and transforms it into an access graph that will be stored within the Neo4j database for rewriting purpose. Secondly, we implemented a Java program that randomly generates access control policies. Finally, we implemented our Cypher query rewriting algorithm in Java and we made its execution possible within the Neo4j browser via the Call clause. That is, one could simply call our algorithm by inputting any Cypher query. This latter will be first rewritten using our algorithm w.r.t the access graph previously computed, and next its rewritten version will be evaluated over the underlying data graph and the result will be returned to the user.
The experiments were carried out on a machine featuring an 8-core processor Intel i5-1135G7 2.4GHz, 16GB of RAM and a NVMe SSD, running Ubuntu 22.04.2. Each experiment were ran 10 times and the mean time is reported.

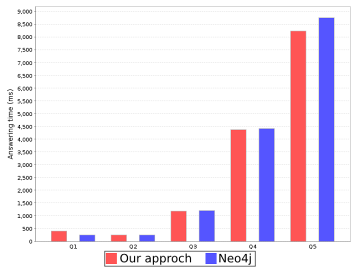
5.2 Experimental results

We report hereafter results of our different experiments. The size of a data graph is given by the number of its nodes and edges. The size of a Cypher query is given by the number of nodes and relationships composing all its statements (i.e. Match, Where and Return). Furthermore, the size of an *ACP* refers to the number of its rules.

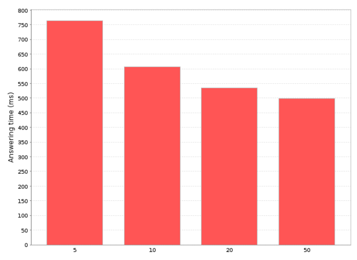
Experiment 1: Efficiency checking.
We experimented the efficiency of our approach by measuring the required times for rewriting Cypher queries as well as times for answering their rewritten versions. We fixed our data graph at 296K nodes and 307K relationships, and we varied sizes of Cypher queries and ACPs.



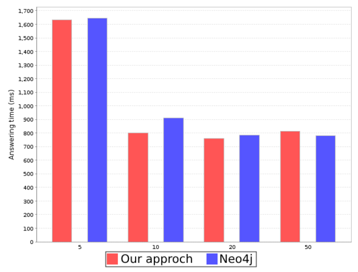
(a) Varying queries (our approach)



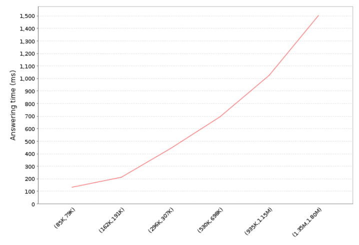
(b) Varying queries (our approach v.s Neo4j)



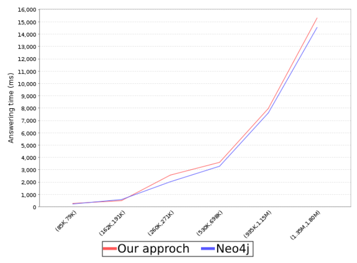
(c) Varying size of the ACP (our approach)



(d) Varying ACP (our approach v.s Neo4j)



(e) Our approach



(f) Our approach v.s Neo4j

Fig. 5: Experimental results

a) Varying size of the Cypher query: We fixed the size of the access control policy into 50 rules and we varied the size of the Cypher queries. We defined different queries ($Q_1, \cdots, Q_5$) by varying the number of nodes, relationships, attribute conditions, Match statements and path conditions. Details of these queries can be found at the appendix C. Then, we computed the whole times required to answer the five queries over the underlying data graph. The results reported in Figure 5-a show that the answering time increases in case of complex queries such as those containing multiple match statements and path conditions.

Next, we compared the efficiency of our model w.r.t that of Neo4j. We first updated our ACP by keeping only unconditional access control rules (as Neo4j does not allow conditional rules), and then we measured the whole time required to answer the five queries used both our approach and Neo4j. The results reported in Figure 5-b shows that the two models behave in a similar way by varying the complexity of the Cypher queries and demonstrate the same efficiency.

b) Varying size of the access control policy: We considered a Cypher query with 3 Match statements, 7 node and relationship labels and 5 conditions. We varied the size of the ACP by considering different kinds and number of access control rules (i.e. positive, negative and conditional rules). Then, we computed the whole time required to answer the Cypher query using each of the defined ACPs. Figure 5-c demonstrates that as the number of access control rules increases, the answering time decreases. This is due to the fact that a greater number of access control rules result in more filtering of the rewritten Cypher query's result.

Consequently, the result size decreases as more access control conditions are incorporated into the Cypher query. This explains why an access control policy with a smaller (resp. larger) size allows for the retrieval of a larger (resp. smaller) result, thereby influencing the increase (resp. decrease) in answering time.

We updated the ACP by removing all conditional access rules and we computed the answering time of both our approach and that of Neo4j. Results are given in Figure 5-d. Notice that the differences between both models are minimal.

Experiment 2: Scalability checking.
By fixing the size of the Cypher query (3 nodes, 2 relationships and 1 attribute condition) and that of the ACP (50 rules), we varied the size of the data graph from $(85K, 79K)$ to $(1.35M, 1.80M)$. We answered the Cypher query over each data graph and we reported the corresponding times in Figure 5-e. Our approach scales well as it takes only $1,500$ ms to answer a complex Cypher query over a big data graph composed of $1.35M$ nodes and $1.80M$ of edges, w.r.t a complex access control policy.

Similarly to previous experiments, we removed all conditional rules in order to favor Neo4j, and we compared our answering times with those required by Neo4j (for the same Cypher query and the same ACP). Results reported in Figure 5-f show that the two models have a high degree of similarity in terms of efficiency.

5.3 Summary of results

The experiments showed that: 1) our approach scales well with complex Cypher queries, complex access control policies as well as large data graphs. 2) The rewriting process adopted by approach does not make our live much harder since our approach has the same degree of efficiency as Neo4j when considering only unconditional rules. 3) Conditional rules do not decrease the efficiency of our approach as the more conditional rules are added the less is the answering time.

6 Conclusion and Future works

Our main motivation was to provide a fine-grained access control for the graph databases stored under Neo4j. Therefore, we proposed an access control language that overcomes limits of the Neo4j model and allows specifying attribute-based policies for each entity of the data graph. To enforce our policies, we adopted a rewriting mechanism that transforms arbitrary Cypher query into a safe query which returns only accessible parts of the data graph. Unlike other solutions, our approach seamlessly integrates into the Neo4j database system, offering a practical and efficient solution for enhancing data security in graph databases. We demonstrated the efficiency and scalability of our approach via an extensive experimental study based on real-life data graph. It is worth noting that current works consider only read privileges. The Neo4j access control model takes into account update privileges but several limits arise especially when it comes to understand and resolve conflicts between read and update privileges. We plan to extend our model by addressing these issues.

References

1. Neo4j access control. `https://neo4j.com/docs/cypher-manual/current/administration/access-control/`, accessed: 2023-06-10
2. Chabin, J., Ciferri, C.D., Halfeld-Ferrari, M., Hara, C.S., Penteado, R.R.: Role-based access control on graph databases. In: SOFSEM. pp. 519–534 (2021)
3. Clark, S., Yakovets, N., Fletcher, G., Zannone, N.: Relog: A unified framework for relationship-based access control over graph databases. In: IFIP. pp. 303–315 (2022)
4. Colombo, P., Ferrari, E.: Efficient enforcement of action-aware purpose-based access control within relational database management systems. IEEE Transactions on Knowledge and Data Engineering. pp. 2134–2147 (2015)
5. Colombo, P., Ferrari, E.: Towards virtual private nosql datastores. In: ICDE. pp. 193–204 (2016)
6. Elliott, A., Knight, S.: Role explosion: Acknowledging the problem. In: Software Engineering research and practice. pp. 349–355 (2010)
7. Fan, W., Chan, C.Y., Garofalakis, M.: Secure xml querying with security views. In: SIGMOD. pp. 587–598 (2004)
8. Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., Taylor, A.: Cypher: An evolving query language for property graphs. In: Proceedings of the 2018 international conference on management of data. pp. 1433–1445 (2018)
9. Hofer, D., Mohamed, A., Küng, J.: Modifying neo4j's object graph mapper queries for access control. In: iiWAS. pp. 421–426 (2022)
10. Jin, Y., Kaja, K.: XACML implementation based on graph databases. In: CATA. pp. 65–74 (2019)
11. Mohamed, A., Auer, D., Hofer, D., Küng, J.: Extended authorization policy for graph-structured data. SN Computer Science. pp. 351–369 (2021)
12. Morgado, C., Baioco, G.B., Basso, T., Moraes, R.: A security model for access control in graph-oriented databases. In: QRS. pp. 135–142 (2018)
13. Rizvi, S., Mendelzon, A., Sudarshan, S., Roy, P.: Extending query rewriting techniques for fine-grained access control. In: SIGMOD. pp. 551–562 (2004)
14. Rizvi, S.Z.R., Fong, P.W.: Efficient authorization of graph-database queries in an attribute-supporting rebac model. ACM Transactions on Privacy and Security (TOPS). pp. 1–33 (2020)
15. Valzelli, M., Maurino, A., Palmonari, M.: A fine-grained access control model for knowledge graphs. In: ICETE (2). pp. 595–601 (2020)
16. You, M., Yin, J., Wang, H., Cao, J., Wang, K., Miao, Y., Bertino, E.: A knowledge graph empowered online learning framework for access control decision-making. World Wide Web pp. 827–848 (2023)

APPENDIX

A Access Graph

In order to reinforce our access control policies correctly and efficiently, we use an intermediate structure, called *access graph*, which allows to convert any textual access control policy into a graph. The benefits are threefold: *1)* it allows to manage easily conflicts between different rules; *2)* it provides a view to the administrator so he/she has a clear idea about the policy; and *3)* it allows our reinforcement algorithm to check quickly the accessibility of any entity.
Each role, privilege (i.e. Traverse or Read) and entity (nodes, relationships or attributes) in the access control policy are represented as nodes in the access graph while edges of the access graph connect privileges to their associated entities. Access condition are defined in edge properties.

We define an access graph G as a property graph (V, E, L, A) where:

1. V is a finite set of vertices;
2. $E \subseteq V \times V$ is a finite set of directed edges where $(u, w) \in E$ specifies an edge starting from vertex u and ending at vertex w;
3. L is a function assigning a label to each vertex in V (resp. edge in E);
4. A is a partial function assigning a set of pairs of the form $(a_1 : v_1), ..., (a_n, v_n)$ to vertices (resp. edges) such that: a_i is a vertex (resp. edge) property and v_i is its value.

We formally define an access control policy P to be a pair (U, R) where: 1) U is a set of user roles involved by the policy; and 2) R is a set of access control rules.

For each rule $r \in R$, its details are given as follows:

a) *r.u*: user role involved by the rule
b) *r.category*: Traverse or Read;
c) *r.permission*: Grant or Deny;
d) *r.value*: true or a condition
c) *r.type*: Node or Relationship;
e) *r.entities*: node (resp. relationship) labels
f) *r.attributes*: node (resp. relationship) attributes if $r.category = Read$

Using our access graph structure, we describe next how to translate any access control policy into its equivalent access graph.

Transformation rules: Given an access control policy $P = (U, R)$, its corresponding access graph $G^P = (V, E, L, A)$ is generated as follows:
For each $r \in R$:

1. If $r.u \notin L(V)$, we create a vertex $v_u \in V$ with $L(v_u) = r.u$ and $A(v_u) = \emptyset$;
2. We create a vertex $v_c \in V$ (if not exists) with $L(v_c) = r.category$ and $A(v_c) = \emptyset$ where $e_u = (v_u, v_c) \in E$ is an edge between v_u and v_c;

3. We create a vertex $v_p \in V$ (if not exists) with $L(v_p) = r.permission$ and $A(v_p) = \emptyset$ where $e_c = (v_c, v_p) \in E$ is an edge between v_c and v_p;
4. For each $e \in r.entities$, we create a vertex $v_e \in V$ (if not exists) with:
   4.1 $L(v_e) = e$ and $(type : r.type) \in A(v_e)$ if $r.category = Traverse$;
   4.2 For each $a \in r.attributes$, we create a vertex with $L(v_e) = a$ and $(entity : r.entities), (type : r.type) \in A(v_e)$ if $r.category = Read$;
   4.3 There exists an edge $e_p = (v_p, v_e) \in E$ with $(condition : X) \in A(e_p)$: $X = r.value \mid (X'$ or $r.value)$ and X' (is the old condition);

B Entity Access Decision Algorithm

We define the procedure elementTraverseCheck given in Figure 6 as follows: *1)* It takes in input an Access Graph G^P, a user role ur, a variable v and it corresponding label l and $type$ (Node or Relationship). *2)* Denied elements list DTE is extracted from access graph G^P using a graph traversal algorithm (line 1). *3)* Check whether the label in input l is in the DTE list or not, if so, access conditions are retrieved from the policy graph G^P if exists, and combined using $AND\ NOT$. In case of deny access condition is true then the access decision to the element l is false (line 3-10). *4)* Similarly to *3)* and *4)*, the label l is checked against the granted element list GTE and use an OR operator to combine grant access conditions (line 11-19). *5)* Finally, grant and deny conditions are combined, optimized then attached to the variable v (line 20-23).

C Experiments Queries

We defined different queries $(Q_1, \cdots, Q_5)$ with different size and complexity to check the efficiency of our approach. The criteria that defines read queries size is: Match statements, number of nodes and relationships in a Match statement, conditions based on attributes, path conditions and Return statements. Hereafter, the description of each query:

– Q1 represents a simple query with a simple pattern.
– Q2 introduces the inclusion of simple attribute conditions.
– Q3 presents a more elaborate query pattern, consisting of a lengthy and intricate structure.
– Q4 represents a query that comprises multiple match statements.
– Q5 combines all the elements found in the aforementioned queries, encompassing both attribute and path conditions.

Algorithm 3: elementTraverseCheck($G^P, ur, v, l, type$)

```
procedure ELEMENTTRAVERSECHECK($G^P, ur, v, l, type$)
    Input: An access graph $G^P$, a user role ur, a variable v, the label l of v, type of the element referred by v (node/relationship)
    Output: The accessibility of v, i.e. true, false or a condition
1   $DTE := (G^P \to ur \to Traverse \to Deny).getRels()$ ;
2   $denyCond = true$;
3   if $\exists dte \in DTE$ with $dte.getEntity(type).label = l$ then
4       if $dte.getAttribute(condition)! = true$ then
5           $denyCond := NOT(dte.getAttribute(condition))$;
6       else return false ;
7   if $\exists dte \in DTE$ with $dte.getEntity(type).label = *$ then
8       if $dte.getAttribute(condition)! = true$ then
9           $denyCond := denyCond$ AND $NOT(dte.getAttribute(condition))$;
10      else return false ;
11  $GTE := (G^P \to ur \to Traverse \to Grant).getRels()$ ;
12  $grantCond = false$;
13  if $\exists gte \in GTE$ with $gte.getEntity(type).label = l$ then
14      if $gte.getAttribute(condition)! = true$ then
15          $grantCond := gte.getAttribute(condition)$;
16      else $grantCond := true$ ;
17  if $\exists gte \in GTE$ with $gte.getEntity(type).label = *$ then
18      if $gte.getAttribute(access)! = true$ then
19          $grantCond = grantCond$ OR $gte.getAttribute(access)$;
20  if $denyCond = true$ then $cond := grantCond$ ;
21  else $cond := grantCond$ AND $denyCond$ ;
22  OptimizeCond($cond$);
23  AttachVariable($cond, v$);
24  return $cond$;
```

Fig. 6: Procedure to check accessibility of an element (node/relationship).