Efficient Authorization of Graph-database Queries in an Attribute-supporting ReBAC Model

SYED ZAIN RAZA RIZVI and PHILIP W. L. FONG, University of Calgary

Neo4j is a popular graph database that offers two versions: an *enterprise edition* and a *community edition*. The enterprise edition offers customizable Role-based Access Control features through custom developed *procedures*, while the community edition does not offer any access control support. Being a graph database, Neo4j appears to be a natural application for Relationship-Based Access Control (ReBAC), an access control paradigm where authorization decisions are based on relationships between subjects and resources in the system (i.e., an authorization graph). In this article, we present AReBAC, an attribute-supporting ReBAC model for Neo4j that provides finer-grained access control by operating over resources instead of procedures. AReBAC employs Nano-Cypher, a declarative policy language based on Neo4j's Cypher query language, the result of which allows us to weave database queries with access control policies and evaluate both simultaneously. Evaluating the combined query and policy produces a result that (i) matches the search criteria, and (ii) the requesting subject is authorized to access. AReBAC is accompanied by the algorithms and their implementation required for the realization of the presented ideas, including *GP-Eval,* a query evaluation algorithm. We also introduce Live-End Backjumping (LBJ), a backtracking scheme that provides a significant performance boost over conflict-directed backjumping for evaluating queries. As demonstrated in our previous work, the original version of *GP-Eval* already performs significantly faster than the Neo4j's Cypher evaluation engine. The optimized version of *GP-Eval*, which employs LBJ, further improves the performance significantly, thereby demonstrating the capabilities of the technique.

CCS Concepts: • **Security and privacy** → **Access control**; **Authorization**; • **Information systems** → *Graph-based database models*;

Additional Key Words and Phrases: Relationship-based access control, attributes, graph patterns, graph database, nano-cypher, Neo4j, live-end backjumping

ACM Reference format:
Syed Zain Raza Rizvi and Philip W. L. Fong. 2020. Efficient Authorization of Graph-database Queries in an Attribute-supporting ReBAC Model. *ACM Trans. Priv. Secur.* 23, 4, Article 18 (July 2020), 33 pages.
https://doi.org/10.1145/3401027

1 INTRODUCTION

Access control is a cornerstone of application security. Databases such as MySQL offer built-in access control features so applications do not have to develop their own mechanisms for controlling what user has access to which resources [20]. Following this example, Neo4j, a popular graph

This work is supported in part by an NSERC Discovery Grant (RGPIN-2014-06611) and a Canada Research Chair (950-229712).
Authors' addresses: S. Z. R. Rizvi and P. W. L. Fong, University of Calgary, 2500 University Drive NW, Calgary Alberta T2N 1N4, Canada; emails: {szrrizvi, pwlfong}@ucalgary.ca.

2471-2566/2020/07-ART18 $15.00
https://doi.org/10.1145/3401027

database, recently introduced Role-Based Access Control (RBAC) features for its paid *enterprise edition* while the free-to-use *community edition* still suffers from a lack of access control features. The enterprise edition ships with five roles: *reader*, *editor*, *publisher*, *architect*, and *admin*. A limitation of this approach is that the capabilities provided by each role span the whole database. For example, a *reader* can read all of the data stored in the database. To remedy this limitation, Neo4j allows the implementer to define their own roles and *procedures* and assign role requirements to procedures. A procedure is a mechanism that allows Neo4j to be extended by writing custom code that can operate on the database and can be invoked directly from the query language. Thus, instead of being able to query the whole database, a user may only be able to invoke a limited number of procedures.

We argue that Neo4j is capable of even finer-grained access control by restricting access to resources rather than procedures. For example, if a procedure allows a requestor to access a certain type of resource, then she can access all resources of that type, regardless of her connection, or lack thereof, with the resources. A similar limitation was encountered by Rizvi et al. [36] for their work with OpenMRS, an open source medical records system. OpenMRS restricts access to its *Application Programming Interface* (API) methods, which are used to communicate with the database. Therefore, if a user has the capability to invoke the method that returns patient records, then she has access to all patient records in the system. They implemented Relationship-Based Access Control (ReBAC) into OpenMRS, thereby providing finer-grained access control so the user only had access to records of the patients she was treating. However, for methods that returned a collection of resources (e.g., searching for patients by name), authorization checking was performed one resource at a time. The reported average time for each authorization check took between 0.016–0.037 second under a specific authorization profile [36, §10]. If a method invocation returns a large collection, we immediately see a draw-back to their approach. Given the performance reported in Reference [36] with only 100 members in the query result set, it would take between 1.6–3.7 seconds for the requestor to get the result of her method invocation. Such a performance is unacceptable where users expect results within a second [29].

The above challenge serves as the primary motivation for this work. One way to remedy the limitation is to have a declarative access control policy language, which can then be translated into database queries. We can then merge the original search query with the applicable access control policy and obtain a single query for which the result set is already filtered to the resources to which the requestor has access. Rewriting queries to enforce access control policies is not new, even for NoSQL databases [16, 24]; this allows the system to evaluate the access authorization along with the database query. What is novel about the model presented in this article is the integration of an attribute-supporting ReBAC model into the paradigm of graph model databases.

In this article, we present AReBAC, an attribute-supporting ReBAC model for controlling access in Neo4j [3], a graph database. Along with the model, we present **Nano-Cypher** and **graph patterns** as two tools for specifying and evaluating database queries and access control policies. The former is a fragment of the original Cypher query language supported by Neo4j, and the latter is a data structure that can express Nano-Cypher queries and access control policies, and it is also used by our query evaluation algorithm. Being able to specify policies in more than one way demonstrates that ReBAC models are not necessarily restricted to their specific policy language. An important feature of these languages is that they are relatively easier to use than logic formulas [10] and variations of regular expressions [12, 18]. This article is accompanied by tools that translate queries and policies between both languages. This serves as the secondary motivation for our work. We need to design models and languages that are accessible by end-users, and we also need to provide the tools required for the realization of the presented ideas.

An appropriate application for the proposed model is where the database queries are defined at the application level and users are able to invoke those queries through an API. This is similar

to the architecture employed by OpenMRS. This approach offers two significant advantages. First, we can pull the access control administration to the application layer, which allows us to treat the database as a black box and not restrict the application to a specific database technology. Second, although we can ask the developers to write the database queries such that they already incorporate the corresponding access control policies, separating queries and policies allows us to modularize the two and manage them independently. We are then able to introduce new, and modify existing, database queries without the concern of the corresponding access control policies. Similarly, we can evolve access control policies without having to modify the database queries that will be affected by the update.

The contributions of this article are as follows:

- We present **Nano-Cypher**, a fragment of the Cypher query language supported by Neo4j. We limit our focus to Nano-Cypher instead of the original Cypher language so we can provide compatibility with graph patterns for specifying queries and policies (Section 2).
- We present AReBAC, an attribute-supporting ReBAC model for controlling access in Neo4j [3], a graph database (Section 3). What distinguishes AReBAC from existing ReBAC models (with attribute support) is that (1) it is rooted in existing technology used in industry; and (2) instead of using the traditional definition of an authorization request (u, r, a), where user u requests to execute action a on resource r, authorization requests in AReBAC are a pair (m, s), where user s requests to execute method m (a database query). This alternative definition allows us to combine database queries with access control policies so we can simultaneously perform query execution and authorization checking.
- We define **graph patterns** (Section 3.1), access control policies (Section 3.2), and the process of weaving queries with policies (Section 3.3). Weaving queries with policies allows us to overcome the two-step process (query execution followed by authorization checking) by evaluating a single graph pattern. The combined graph pattern specifies the requirements of the query as well as the policy. We choose to work with Nano-Cypher queries and evaluate graph patterns through our proposed Graph Pattern evaluation engine rather than Cypher queries using the Neo4j Cypher evaluation engine, because our proposed graph pattern evaluation algorithm is able to evaluate graph patterns significantly faster than the Neo4j Cypher evaluation engine can evaluate equivalent Cypher queries (as demonstrated in Reference [35]). Additionally, while the evaluation of a graph pattern does not depend on the syntactic sequencing of its components, it is not the case for Cypher queries. For example, given a Cypher query, we may be able to reorder its components while preserving its semantics. As a result, evaluating both versions of the query would provide the same results; however, one version may perform significantly faster than the other. This feature allows users to manually optimize Cypher queries for optimal performance. Since evaluating a graph pattern does not depend on the syntactic sequencing of its components, users do not need to manually optimize Nano-Cypher queries and/or their equivalent graph patterns.
- We define (i) how to translate a Nano-Cypher query to a Graph Pattern and vice versa (Section 4.1), (ii) how multiple graph patterns can be combined into a single graph pattern that maintains requirements of each of the original graph patterns (Section 4.2), and (iii) we introduce Live-End Backjumping (LBJ) and present *GP-Eval*, a Forward Checking algorithm with Live-End Backjumping (FC-LBJ) for evaluating a graph pattern against a database (Section 4.3). *GP-Eval* employs a novel approach to forward checking that focuses on minimizing the number of database accesses (Section 4.3.4), while the Live-End Backjumping component is a novel backtracking method based on the principles of Conflict-Directed Backjumping (CBJ). Forward Checking (FC) and CBJ are well-established intelligent backtracking algorithms [31].

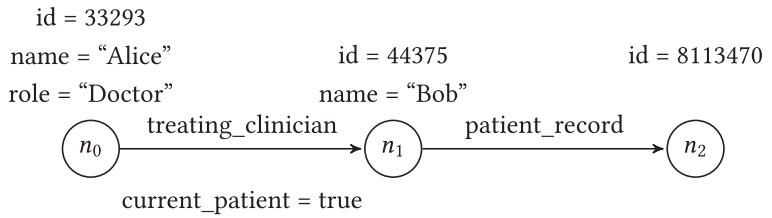


Fig. 1. Sample database.

- We evaluate (i) the performance of *GP-Eval* (FC-LBJ) and demonstrate that it performs faster than *GP-Eval* (FC-CBJ) and *GP-Eval* (FC) when evaluating queries that are expressible using Nano-Cypher (Section 5.1); and (ii) how often we encounter a high-degree node (i.e., a hub) when evaluating graph pattern queries (Section 5.2). The original version of *GP-Eval* employed FC and offered a significant performance boost over Neo4j's Cypher evaluation engine [35]. For the largest set of queries (in terms of number of nodes), *GP-Eval* was able to complete 616 evaluations out of 1K, where each evaluation was limited to 6 seconds and had an average completion time of 0.630 second. For the same set of queries, Neo4j's Cypher evaluation engine was able to complete only 71 evaluations and had an average completion time of 3.063 seconds. The optimized version of *GP-Eval* employs FC-LBJ, which offers a significant performance boost over the original version, thereby demonstrating that FC-LBJ is even more competitive.
- We provide the tools required for the realization of the ideas proposed in this article [33].

2 GRAPH DATABASE

Graph databases present a database model in which the data are structured as a possibly edge-labeled and directed graph [8]. Data retrieval and manipulation are expressed as graph-oriented operations, i.e., operations that operate on graph features such as paths, neighborhoods, subgraphs, and graph patterns. The graph model allows for a topological-based approach of data representation that can be exploited by the system for assessing connectivity of entities rather than depending on costly join operations, such as in relational databases. One popular example of graph databases is Neo4j [3]. Neo4j structures data as a directed and edge-labeled graph that supports properties for nodes and relationships. We use the term *property* in the context of Neo4j; otherwise, we use the term *attribute* (e.g., in the context of database queries). Figure 1 presents a small database example consisting of three nodes and two edges. The properties for each node are listed above the node, while the properties for relationship are listed below the relationship. Each relationship also consists of a label, which is specified above the relationship. For example, node n_1 has properties id and name, with values 44,375 and "Bob," respectively. Neo4j exposes two interfaces, an embedded *Java API* and an SQL-inspired query language *Cypher*.

"Cypher is a declarative, SQL-inspired language for describing graph patterns using ascii-art syntax" [1]. A Cypher query describes a graph pattern that is then matched against the data store for data retrieval and manipulation. The language uses `Match` statements for specifying nodes and relationships to match against the data store. `Where` statements with one or more clauses (conjuncted together) are used to specify further restrictions for matching the already-specified nodes and relationships, such as property requirements. `Return` statements describe the result set of executing a query, which can contain nodes, relationships, properties for both nodes and relationships, paths in the subgraph, and aggregation functions. Cypher also supports statements for manipulating

the data store; `Create` statements for creating nodes and relationships, `Set` statements for setting properties for nodes and relationships, and `Delete` statements for deleting nodes and relationships.

2.1 Attribute Support Schema

As mentioned above, Neo4j supports properties (attributes) for both nodes and relationships. Thus, to facilitate discussion, we will describe an attribute support schema. The purpose of the schema is to provide a uniform structure for us to discuss edge labels and attribute requirements for Nano-Cypher queries (Section 2.2) and graph patterns (Section 3.1), as well as attribute-supporting edge-labeled and directed graphs.

Before jumping into the schema, we will first describe the set of data types, DT. Suppose $\mathcal{T}$ is the universe of discourse. A data type dt is a pair $(T, \mathcal{R})$, where $T \subseteq \mathcal{T}$ and $\mathcal{R} \subseteq 2^{T \times T}$ is a set of binary relations over T. In other words, $R \subseteq T \times T$ for each $R \in \mathcal{R}$. We assume that a set DT of data types is given.

The attribute support schema, χ, is described as follows:

$$\chi = (DT, ID, VA, EA, VT, ET),$$

where DT and ID are sets of data types and edge labels, respectively. VA and EA are countably infinite sets of attribute names for vertices and edges, respectively. $VT : VA \to DT$ is a function that maps a vertex attribute name to its corresponding data type. Similarly, $ET : EA \to DT$ is a function that maps an edge attribute name to its corresponding data type. Example 1 extends the sample database presented in Figure 1 and presents the corresponding attribute support schema.

Example 1. The sample database presented in Figure 1 is based on an attribute support schema that consists of three data types, three vertex attribute names, and one edge attribute name. Before specifying the attribute support schema, χ, we define individual data types as follows:

$$\begin{aligned} \text{int} &= (\text{Integer}, \{=_{\text{int}}, \neq_{\text{int}}, <_{\text{int}}, \leq_{\text{int}}, >_{\text{int}}, \geq_{\text{int}}\}), \\ \text{str} &= (\text{String}, \{=_{\text{str}}, \neq_{\text{str}}\}), \\ \text{bool} &= (\text{Boolean}, \{=_{\text{bool}}, \neq_{\text{bool}}\}). \end{aligned}$$

Each data type consists of a set of values and a set of operations defined over those values. For example, the *int* data type is composed of Integer values along with the set of operations $\{=_{\text{int}}, \neq_{\text{int}}, <_{\text{int}}, \leq_{\text{int}}, >_{\text{int}}, \geq_{\text{int}}\}$. We are now able to define $\chi = (DT, ID, VA, EA, VT, ET)$, where each component is defined as follows:

$$\begin{aligned} DT &= \{\text{int}, \text{str}, \text{bool}\}, \\ ID &= \{\text{treating\_clinician}, \text{patient\_record}\}, \\ VA &= \{\text{id}, \text{name}, \text{role}\}, \\ EA &= \{\text{current\_patient}\}, \\ VT &= \{(\text{id} \to \text{int}), (\text{name} \to \text{str}), (\text{role} \to \text{str})\}, \\ ET &= \{(\text{current\_patient} \to \text{bool})\}. \end{aligned}$$

We introduce two functions to extract the components of data types λ and θ. Given a data type, $dt = (T, \mathcal{R})$, $\lambda(dt)$ returns its value-domain T, while $\theta(dt)$ returns its set of binary relations $\mathcal{R}$.

We define an attribute-supporting edge-labeled directed graph, where the attribute names and values correspond to χ, as $G = (V, E, A_V, A_E)$ where V and E represent the nodes and relationships. A relationship $id(u, v) \in E$, with $u, v \in V$ and $id \in ID$ specifies an edge from u to v with label id. The pair (A_V, A_E) provides the attribute support. $A_V : V \times VA \to \mathcal{T} \cup \{\perp\}$, is a function that assigns attribute values to nodes. It is further required that $A_V(v, va)$ returns a value

in $\lambda(VT(va)) \cup \{\perp\}$. If the node v has the attribute va, then $A_V(v, va) \in \lambda(VT(va))$, otherwise, $A_V(v, va) = \perp$. $A_E : E \times EA \to \mathcal{T} \cup \{\perp\}$ is similar to A_V, except it represents attributes associated with relationships instead of nodes. Again, we require that $A_E(e, ea)$ returns a value in $\lambda(ET(ea)) \cup \{\perp\}$.

Example 2. We are now able to formally define sample graph database presented in Figure 1, as $G = (V, E, A_V, A_E)$ where the components are defined as follows. Note, we use the names e_0 and e_1 to represent the relationships treating_clinician(n_1, n_0) and patient_record(n_1, n_2), respectively.

$$\begin{aligned} V &= \{n_0, n_1, n_2\}, \\ E &= \{e_0, e_1\}, \\ A_V &= \{((n_0, \text{id}) \to 33293), ((n_0, \text{name}) \to \text{“Alice”}), ((n_0, \text{role}) \to \text{“Doctor”}), \\ &\quad ((n_1, \text{id}) \to 44375), ((n_1, \text{name}) \to \text{“Bob”}), \\ &\quad ((n_2, \text{id}) \to 8113470)\}, \\ A_E &= \{(e_0, \text{current\_patient}) \to \text{true}\}. \end{aligned}$$

2.2 Nano-cypher

In this work, we focus our attention on a fragment of the Cypher query language that we dubbed **Nano-Cypher**. The limited scope allows us to work with a language that is compatible, in terms of expressiveness, with Graph Patterns (Section 3.1), so database queries can be merged with access control policies (Section 3.2) to simultaneously perform query execution and authorization checking. In the rest of this section, we describe the syntax and semantics for Nano-Cypher.

Nano-Cypher is a fragment of the Cypher query language supported by Neo4j [3] and is parametrized by countably infinite sets of vertex and edge variable names (VX, EX). Much like Cypher queries, Nano-Cypher queries are executed against edge-labeled and directed graphs that support attributes for node and relationships. Additionally, the edge labels and attribute support correspond to the schema χ. Note, to avoid confusion, we use the terms **nodes** and **relationships** when referring to the elements of the graph database, and the terms **vertices** and **edges** when referring to the elements of a query. A query execution is satisfied by a set of models where each model maps the vertices and edges from the query to nodes and relationships in the graph, respectively, such that each instance of mapping preserves the constraints specified in the query. We formally define such model below, along with the semantics of Nano-Cypher.

Abstract Syntax. Fixing a schema $\chi = (DT, ID, VA, EA, VT, ET)$, and sets VX and EX, a Nano-Cypher query Q is composed of three components: multiple `Match` statements, each followed by an optional `Where` statement and a single `Return` statement. We first describe the internal structures of each of these components and then describe how they fit together.

A Nano-Cypher query starts off with a `Match` statement with the body ϕ:

$$\phi ::= (v) \mid (v)\text{-}[e : id]\text{->}\phi \mid (v)\text{<-}[e : id]\text{-}\phi,$$

where $v \in VX$ is a vertex variable, $e \in EX$ is an edge variable, and $id \in ID$ is an edge label. (v)-$[e : id]$->(u) specifies an outgoing relationship from v to u with label id[1], and (v)<-$[e : id]$-(u) specifies an incoming relationship for v from u with label id.

We can specify *mutual exclusion constraints* and *attribute requirements* using the `Where` statements with a condition ψ:

$$\psi ::= v\text{<>}u \mid v.va\, R_{va}\, val_{va} \mid e.ea\, R_{ea}\, val_{ea} \mid \psi\ AND\ \psi,$$

[1] e is used to refer to the edge (v)-$[e : id]$->(u).

where $v, u \in VX$ are vertex variables that have appeared before in a ϕ statement, $va \in VA$ is a vertex attribute name, $R_{va} \in \theta(VT(va))$ is a relation defined over va, and $val_{va} \in \lambda(VT(va))$ is a valid value for va. Similarly, $e \in EX$ is an edge variable that has appeared before in a ϕ statement, $ea \in EA$ is an edge attribute name, $R_{ea} \in \theta(ET(ea))$ is a relation defined over ea, and $val_{ea} \in \lambda(ET(ea))$ is a valid value for ea. $v<>u$ specifies a **mutual exclusion constraint**, i.e., the same node cannot be assigned to both v and u, $v.va\ R_{va}\ val_{va}$ specifies an attribute requirement for v, and $e.ea\ R_{ea}\ val_{ea}$ specifies an attribute requirement for e. Note that the attribute requirement clauses follow an infix notation. For example, the vertex attribute requirement $v_1.age > 18$, where $v_1 \in VX$, $age \in VA$, $> \in \theta(VT(age))$, and $18 \in \lambda(VT(age))$, requires that the node assigned to v_1 must have the age attribute with value greater than 18.

Once we have specified all requirement, we can specify the `Return` statement of the query using the following syntax:

$$\mu ::= v \mid v, \mu,$$

where $v \in VX$ is a vertex variable that has appeared before in a ϕ statement. Intuitively the `Return` statement composes the result set of the query execution.

Combining all three of these components together to form a complete Nano-Cypher query, we get the following syntax:

$$\begin{aligned} Q &::= \alpha\ \texttt{Return}\ \mu \\ \alpha &::= \beta \mid \beta\ \alpha \\ \beta &= \texttt{Match}\ \phi\ \delta \\ \delta &::= \epsilon \mid \texttt{Where}\ \psi. \end{aligned}$$

Given a `Match` statement with body ϕ, the function $hd : \phi \rightarrow VX$ returns the left-most vertex variable (i.e., the head) in ϕ, which allows us to extract specific vertices from the query to evaluate it.

Example 3. $hd((v)\text{-}[a : friend]\text{->}(u)\text{<-}[b : friend]\text{-}(w)) = v$.

Semantics. When a Nano-Cypher query Q is executed against an attribute-supporting edge-labeled directed graph $G = (V, E, A_V, A_E)$, it is satisfied by a set of models where each model maps the vertex and edge variables from Q to nodes and relationships in G, respectively, such that each instance of mapping preserves the constraints specified in Q. We define a model as $M : VX \cup EX \rightarrow V \cup E$:

Example 4. $M = \{v_1 \mapsto 123, v_2 \mapsto 124, \ldots\ e_1 \mapsto treating\_clinician(125, 126),\ e_2 \mapsto parent(127, 128), \ldots\}$,

where $v_1, v_2 \in VX, e_1, e_2 \in EX, 123, \ldots, 128 \in V$, and $treating\_clinician(125, 126), parent(127, 128) \in E$. The return statement is then used to form a restriction on each model, $M\mid_{\mu}$, which forms one of the entries in the return set of the query. For example:

Example 5. Continuing from Example 4, if the return statement of the query only consists of the vertex v_2, i.e., $\mu = \{v_2\}$, then $M\mid_{\mu} = \{v_2 \mapsto 124\}$.

Let $\mathcal{M}$ be the set of all models that satisfy the query Q executed against the graph G. We can now define the product of query evaluation as $result\text{-}set(G, Q)$:

$$result\text{-}set(G, Q) = \bigcup_{M \in \mathcal{M}} M\mid_{\mu}.$$

$M \in \mathcal{M}$ iff M can satisfy the sequence of `Match` and `Where` statements. A `Match` statement is satisfied iff its body ϕ is satisfied, and a `Where` statement is satisfied iff its condition ψ is satisfied. The satisfaction relationship between M and ϕ and ψ is defined as follows:

$M \models v$ iff $M(v)$ is defined.
$M \models v$ -[$e : i$]->ϕ iff $M \models v$ and $M(e) = i(M(v), M(hd(\phi)))$ and $M \models \phi$.
$M \models v$ <-[$e : i$]-ϕ iff $M \models v$ and $M(e) = i(M(hd(\phi)), M(v))$ and $M \models \phi$.
$M \models v_1 <> v_2$ iff $M \models v_1$ and $M \models v_2$ and $M(v_1) \neq M(v_2)$.
$M \models v.va\ R_{va}\ val_{va}$ iff $M \models v, A_V(M(v), va) \neq \perp$, and $((A_V(M(v), va)), val_{va}) \in R_{va}$.
$M \models e.ea\ R_{ea}\ val_{ea}$ iff $M(e)$ is defined, $A_E(M(e), ea) \neq \perp$, and $((A_E(M(e), ea)), val_{ea}) \in R_{ea}$.
$M \models \psi_1$ AND ψ_2 iff $M \models \psi_1$ and $M \models \psi_2$.

The intuition behind the satisfaction relationship is that each vertex variable is mapped to a node and each edge variable is mapped to a relationship such that the connectivity structure, mutual exclusion constraints, and attribute requirements are satisfied. Once we have all the models that satisfy the query, we filter the domain of each model to only consist of vertex variables specified in the `Return` statement. The set of filtered models is the result set of the query execution.

3 AReBAC

Traditional authorization requests are a triple (u, r, a), where user u requests to perform action a on resource r. A limitation of this paradigm is that if one wishes to perform authorization checking on a collection of resources, then the authorization checking would need to be performed one resource at a time. For example, a query is first invoked that returns the result set, and then authorization checking is performed on each entry in the result set. An approach to address this limitation is to weave the query evaluation with the authorization checking, thereby performing both tasks simultaneously and obtaining the results that (1) match the search parameters and (2) the requester has access to. In this section, we present AReBAC, an attribute-supporting ReBAC model that is able to perform query evaluation and authorization checking simultaneously due to its declarative policy language. We will describe the ontology of the model in two steps: First, we define the components required for supporting the data store (i.e., the resources, users, authorization graph, and attributes); next, we define the components required for accessing resources, (i.e., the methods and access control policies), along with the utility components to help enforce access control policies. We follow the definition of the protection state with descriptions and examples for the components to demonstrate their purpose.

Definition 1. The data store support for AReBAC protection state, corresponding to the attribute support schema $\chi = (DT, ID, VA, EA, VT, ET)$, has the following components:

- O: The set of objects (resources).
- $S \subseteq O$: The set of subjects (users).
- $AG \subseteq ID \times O \times O$: The directed and edge-labeled graph, in which $(id, u, v) \in AG$ refers to a relationship from object u to object v that is labeled with the relation identifier id.
- $Attr_O : O \times VA \rightarrow \mathcal{T} \cup \{\perp\}$: The attribute assignment function for objects. For example, $Attr_O(o, va) = val$, for $o \in O$, $va \in VA$, and $val \in (\lambda(VT(va)) \cup \{\perp\})$. If the object o has the attribute va, then $val \in \lambda(VT(va))$, otherwise, $val = \perp$.
- $Attr_{AG} : AG \times EA \rightarrow \mathcal{T} \cup \{\perp\}$: Similar to $Attr_O$, except $Attr_{AG}$ assigns attributes to relationships.

Definition 2. The resource access support for AReBAC protection state, corresponding to the attribute support schema $\chi = (DT, ID, VA, EA, VT, ET)$, has the following components:

- MD: The set of methods (actions) that a subject can invoke. Each method represents a database query specified using Nano-Cypher.
- C: The set of categories. Categories are used to describe the types of methods (discussed below).
- $CH \subseteq C \times C$: A partial ordering on C called the category hierarchy. We use the shorthand $c \geq c'$ if $(c, c') \in CH$.
- $MC : MD \rightarrow C$: A mapping from a method to a category.
- ACT: The finite set of actors. The actors represent special vertices specified in queries (Section 3.1).
- $CA : C \rightarrow 2^{ACT}$: A mapping from a category to a set of actors.
- P: The set of all graph patterns recognized by the system (Section 3.1).
- $CP : C \nrightarrow \mathsf{P}$: A partial function that assigns some categories to system recognized graph patterns.

An appropriate application for AReBAC is where the database queries are defined at the application level and users are able to invoke those queries through API methods. This is similar to the architecture employed by OpenMRS. *Note that each API method is a wrapper for a database query that may access one or more resources in the database.* When a subject s invokes a method m, where m may require specific parameters, an **authorization request** is generated.

Definition 3. An **authorization request** (or simply a **request**) is a pair, $(m, s) \in MD \times S$, whereby subject s requests to invoke method m.

Recall that traditional authorization requests are a triple, (u, r, a), where user u requests to perform action a on resource r, and based on the protection state the request may be *allowed* and *denied* [12, 18, 20, 36]. Unlike traditional access control models, AReBAC always invokes the requested method m; however, the database query contained in m is modified based on the access control policies. As a result, the rewritten query only returns resources to which the user has legitimate access. In Section 3.2 and Section 3.3, we discuss how to obtain the corresponding access control policy for a given method and how to modify the contained query to enforce the access control policy, respectively. Note that authorization requests framed as *subjects requesting to invoke methods* seems similar to the model presented for Object Oriented Databases and OSQL [6] where a user requests to invoke a function that is defined as part of Object type description. We argue that the authorization requests presented for OSQL differ from the AReBAC requests, since the OSQL authorization model specifies the target resource and function, and the request is either allowed or denied (thus, it follows traditional authorization requests), whereas the AReBAC model always invokes the requested method and executes a modified version of the underlying database query.

Example 6. Alice, a doctor, wishes to read the health record of Bob, her patient. To do this, she invokes the method read_hr(bob_hr), where bob_hr is the identifier for Bob's health record, and provided to the method read_hr as a parameter. This method invocation gets translated to the authorization request (read_hr(bob_hr), Alice).

Each method has an associated **category** that describes the type of the method. Additionally, the category specifies the **actors** involved in the method. Actors represent the important objects involved in the method; however, not all involved objects are associated with actors. During the request evaluation objects are assigned to appropriate actors.

Example 7. The protection state consists of three categories; $c_{medical\_data}$, c_{read_hr}, and c_{read_admin}, with $c_{read\_hr} \geq c_{medical\_data}$ and $c_{read\_admin} \geq c_{medical\_data}$. The category $c_{medical\_data}$ is used to describe the common components between health record and administrative data, while c_{read_hr} and c_{read_admin} describe the specific details (e.g., the actors) associated with health records and administrative data, respectively.

Example 8. Once the authorization request of Example 6 is issued, the category $c_{read\_hr} = MC(\text{read\_hr})$ is looked up. Based on the category, the system finds the important actors involved in the method, $CA(c_{read\_hr}) = \{\text{requestor, health\_record, patient}\}$. In this case, the important actors for c_{read_hr} are the requestor, health_record, and the patient.

The underlying database query for the method is designed in such a way that it is intuitive to observe how the actors are involved in the query.[2] At this stage the query contains variables that are replaced by parameters to complete the query. Additionally, when the developer creates the method she also specifies the category for the method by updating *MC*.

Example 9. The Nano-Cypher query for the read_hr method has the form:

```
Match (requestor)
Match (patient) − [: patient_record]− > (health_record)
Where requestor.id = REQ_ID AND health_record.id = HR_ID
Return health_record,
```

where REQ_ID and HR_ID are variables to be replaced by Alice's unique identifier and the parameter *bob_hr*, respectively. The query also specifies the actor to vertex mapping through the variable names. Notice that the vertex requestor is not associated with any edges. However, the developer still specifies that requestor vertex, because it is a specified actor for the category, and its requirements will be specified in the corresponding access control policy.

The category is not only used for specifying the types of the methods, but also the types of the access control policies. The system uses the categories and the category hierarchy to find the appropriate access control policy for the authorization request. The category hierarchy, *CH*, is a partially ordered set that represents a refinement relationship between the categories. For $c_1, c_2 \in C$, if $c_1 \geq c_2$, then c_1 is more refined than c_2 and thus, all of the actors associated with c_2 are also associated with c_1 (i.e., $CA(c_1) \supseteq CA(c_2)$). The relation also asserts the requirement that it must be the case that c_1 is at least as restrictive as c_2. The process for finding the access control policy for an authorization request is defined in Section 3.2.

Finally, the system weaves the policy with the method and invokes the combined database query. The result of the query is the set of resources that (1) match the initial search query and (2) the requestor has access to. The process of weaving access control policies with methods is described in Section 3.3.

3.1 Graph Patterns

Since AReBAC uses graph databases as its back-end data store, we use **graph patterns** to specify database queries internally. Intuitively, graph patterns represent Nano-Cypher queries, with additional information regarding the type of the query (categories) and specific vertices (actors). Recall that a Nano-Cypher query Q consists of a set of vertex and edge variables, edge labels, mutual exclusion constraints, attribute requirements for vertices and edges, and a return statement.

[2]For the remainder of this article, we assume that the "important actor" variables in Nano-Cypher queries are named the same as their actor name.

Given an AReBAC protection state and the attribute support schema $\chi = (DT, ID, VA, EA, VT, ET)$, a graph pattern models these requirements through a set of vertices V, edges E, mutual exclusion constraints Σ, vertex attribute requirements Γ_V, edge attribute requirements Γ_E, and the return statement $Ret \subseteq V$. A graph pattern also contains a category c for describing the type of the graph pattern and a mapping G_{ACT} from actors in $CA(c)$ to V.

Definition 4. Given an AReBAC protection state and the attribute support schema $\chi = (DT, ID, VA, EA, VT, ET)$, a graph pattern is an eight tuple, $GP = (V, E, \Sigma, \Gamma_V, \Gamma_E, c, G_{ACT}, Ret)$, which consists of a set of vertices V, a set of directed and labeled edges $E \subseteq ID \times V \times V$, a set of mutual exclusion constraints $\Sigma \subseteq V \times V$, a set of vertex attribute requirements $\Gamma_V \subseteq V \times VA \times 2^{\mathcal{T} \times \mathcal{T}} \times \mathcal{T}$, a set of edge attribute requirements $\Gamma_E \subseteq E \times EA \times 2^{\mathcal{T} \times \mathcal{T}} \times \mathcal{T}$, a category c, a function $G_{ACT} : CA(c) \rightarrow V$ that maps the actors in $CA(c)$ to the vertices in V, and a set of vertices $Ret \subseteq V$ specifying the return statement. To simplify notation when necessary, we use $V(GP)$, $E(GP)$, $\Sigma(GP)$, $\Gamma_V(GP)$, $\Gamma_E(GP)$, $C(GP)$, $G_{ACT}(GP)$, and $Ret(GP)$ to refer to the components of the graph pattern GP.

The sets V and E specify the vertices and edges, respectively. An edge $id(u, v) \in E$, with $id \in ID$ and $u, v \in V$, is an edge from u to v with label id. The intuition behind an edge $id(u, v)$ is that, during a query evaluation, if n_1 is mapped to u and n_2 is mapped to v, where $n_1, n_2 \in O$, then it must be the case that $id(n_1, n_2) \in AG$.

Mutual exclusion constraints $\Sigma \subseteq V \times V$ are used to specify further restrictions for query evaluations. The intuition behind a mutual exclusive constraint $(u, v) \in \Sigma$, with $u \neq v$, is that during a query evaluation, if n_1 is mapped to u and n_2 is mapped to v, where $n_1, n_2 \in O$, then it must be the case that $n_1 \neq n_2$.

Γ_V specifies the set of vertex attribute requirements for the graph pattern. A vertex attribute requirement, $\gamma_V = (v, va, R, val) \in \Gamma_V$, with $v \in V$, $va \in VA$, $R \in \theta(VT(va))$, and $val \in \lambda(VT((va))$, specifies an attribute requirement for vertex v. The intuition behind the vertex attribute requirements is that, during a query execution, if a node n is mapped to vertex v, then it must be the case that $Attr_O(n, va) \neq \perp$ and $(Attr_O(n, va), val) \in R$. We say that n is a valid candidate for v if it satisfies all of the attribute requirements associated with v. Similarly, Γ_E specifies edge attribute requirements for the graph pattern.

The remaining components c, G_{ACT}, Ret do not add further restriction for the graph pattern, but rather serve as utilities for working with graph patterns. c and G_{ACT} facilitate in computing the enforced access control policies (Section 3.2) and combining graph patterns (Section 4.2). The set Ret is used to obtain the return value of the query execution.

A graph pattern query is executed against an AReBAC protection state and is satisfied by sets of objects and relationships that together satisfy the graph pattern. We refer to each set as a map, $M : V \cup E \rightarrow O \cup AG$, which is a complete function that maps vertices and edges from the graph pattern to objects and relationships in the protection state. For example:

Example 10. $M = \{u \mapsto 123, v \mapsto 124, \ldots e_1 \mapsto treating\_clinician(125, 126), \quad e_2 \mapsto parent(127, 128), \ldots\}$,

where $u, v \in V$, $e_1, e_2 \in E$, $123, \ldots, 128 \in O$, and $treating\_clinician(125, 126)$, $parent(127, 128) \in AG$. The set $Ret \subseteq V$ is used to form a restriction on each map, $M\,|_{Ret}$: $V \rightarrow O$, which forms the return value of the query. For example:

Example 11. Continuing from Example 10, if $Ret = \{v\}$, then $M\,|_{Ret} = \{v \mapsto 124\}$.

For a fixed protection state, a map M satisfies a graph pattern $GP = (V, E, \Sigma, \Gamma_V, \Gamma_E, c, G_{ACT}, Ret)$, denoted $M \vDash GP$ if and only if the following statements are true:

(1) $\forall v \in V, M(v)$ is defined. Each vertex is mapped to a node in O.
(2) $\forall i(u, v) \in E, M(i(u, v)) = i(M(u), M(v))$. Each edge is mapped to a relationship in AG, with u and v mapped to $M(u)$ and $M(v)$.
(3) $\forall (u, v) \in \Sigma, u \neq v$ then $M(u) \neq M(v)$. Any two mutually exclusive vertices are not mapped to the same object.
(4) $\forall (v, va, R, val) \in \Gamma_V, Attr_O(M(v), va) \neq \perp$, and $(Attr_O(M(v), va), val) \in R$. Each vertex attribute requirement is satisfied.
(5) $\forall (e, ea, R, val) \in \Gamma_E, Attr_{AG}(M(e), ea) \neq \perp$, and $(Attr_{AG}(M(e), ea), val) \in R$. Each edge attribute requirement is satisfied.

3.2 Policies

In this section, we discuss access control policies and related concepts. We first define access control policies, followed by the process of computing the **enforced policy** for a given authorization request.

An access control policy is a graph pattern used for specifying authorization requirements for accessing resources. The requirements are specified in the form of a graph along with mutual exclusion constraints and attribute requirements. Essentially, a policy is a graph pattern.

Definition 5. A policy is a graph pattern used for specifying authorization requirements for access resources. Given an AReBAC protection state, $CP : C \nrightarrow \mathsf{P}$ provides the policy specified for a given category.

The above definition ensures that for each category at most one policy is specified. A result of this requirement is that methods of the same category are restricted in the same way. Recall that the category hierarchy is a refinement relationship (partial ordering) between the categories in C. Thus, for two categories $c_i, c_j \in C$, if $c_i \geq c_j$, then c_i is more refined than c_j and $CA(c_i) \supseteq CA(c_j)$. This relation implies that methods with higher-level categories provide access to more specific resources and therefore enforce stricter restrictions.

To address this situation, one could demand that policy designers ensure this restriction manually when designing policies, however, it is much simpler to leverage the category hierarchy to achieve the same result. This is obtained by the superior category policies inheriting all of the requirements from their inferior counterparts. The function $\sqcup_c : 2^{\mathsf{P}} \rightarrow \mathsf{P}$ combines a set of graph patterns into a single graph pattern that specifies all of the requirements from input set. $\sqcup_c$ is formally defined in Section 4.2. For a category $c \in C$, we use $\sqcup_c$ to combine $CP(c)$ with all $CP(c_i)$ such that $c_i \in C$ and $c \geq c_i$. For an authorization request (m, s) with $MC(m) = c$, the function $Pol : C \rightarrow \mathsf{P}$ returns the **enforced policy**.[3]

Definition 6. $Pol(c) = \sqcup_c \{GP_i \mid c_i \in C, c \geq c_i \wedge CP(c_i) = GP_i\}$.

Being able to combine graph patterns allows us to have implicitly defined policies. A combined policy will enforce the requirements of all of its components, thereby ensuring the most relevant requirements are enforced based on what is explicitly defined in the protection state. There exists one and only one enforced policy for each category; however, $CP(c)$ does not need to be defined for every $c \in C$. If a category c does not require any new restrictions that have not already been specified by the policies of its inferior categories, then $CP(c)$ is undefined.

Figure 2 presents a category hierarchy where the policies are defined as follows: $CP(c_1) = GP_1$, $CP(c_3) = GP_3$, $CP(c_4) = GP_4$, and $CP(c_2)$ is undefined. The following three examples demonstrate the scenarios for computing the enforced policy for a given authorization request (m, s).

[3]This process can be optimized by the system computing $Pol(c)$ for each category once and storing them alongside the protection state.

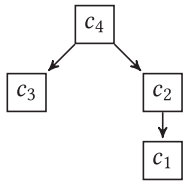


Fig. 2. Category hierarchy. $c_1, c_2, c_3, c_4 \in C$ and $CH = \{(c_2, c_1), (c_4, c_2), (c_4, c_3)\}$.

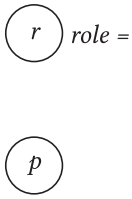


Fig. 3. GP_x.

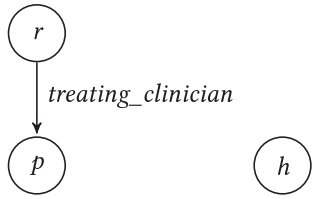


Fig. 4. GP_y.

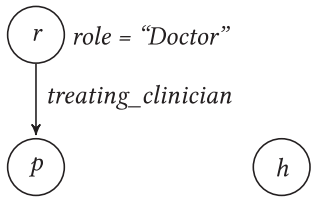


Fig. 5. The enforced policy $\sqcup_{c_{read\_hr}}\{GP_x, GP_y\}$.

Example 12. If $MC(m) = c_1$, then GP_1 is the appropriate enforced policy for the request. This is because GP_1 is explicitly defined and there are no other categories that are inferior to c_1:

$$Pol(c_1) = GP_1.$$

Example 13. If $MC(m) = c_2$, then the enforced policy for c_2 depends on the policy for c_1, since $CP(c_2)$ is undefined and c_2 is only superior to c_1 (other than itself). $Pol(c_2)$ is computed as follows:

$$Pol(c_2) = \sqcup_{c_2}\{GP_1\} = GP_1.$$

Example 14. If $MC(m) = c_4$, the system will combine the requirements for $CP(c_1)$, $CP(c_3)$, and $CP(c_4)$ to form the enforced policy. Since c_4 is superior to itself, c_3, c_2, and c_1, $Pol(c_4)$ is composed of its sub-parts:

$$Pol(c_4) = \sqcup_{c_4}\{GP_4, GP_3, GP_1\}.$$

If the access control policies are explicitly defined for all $\geq -minimal$ categories, then we say that CP is sufficient for C. We denote this relationship by $\text{sufficient}_C(CP)$. The sufficiency condition allows us to ensure that the enforced policy for each category is composed of explicitly defined requirements. For a protection state to be usable, it must be the case that CP is sufficient for the category set (i.e., $\text{sufficient}_C(CP)$). The sufficiency condition can be checked in time linear to the size of the category hierarchy.

Example 15 extends Example 9 and presents the enforced policy that will be applied to the *read_hr* method, which is obtained through $\sqcup_{MC(read\_hr)}$.

Example 15. The applicable policy graph pattern for the *read_hr* method is defined as:

$$Pol(c_{read\_hr}) = \sqcup_{c_{read\_hr}}\{GP_x, GP_y\},$$

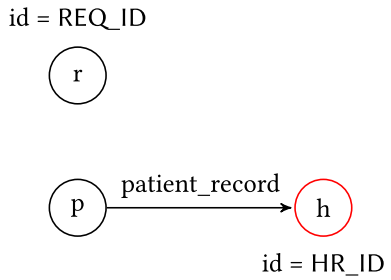


Fig. 6. Graph pattern representation of the *read_hr* method specified in Example 9.

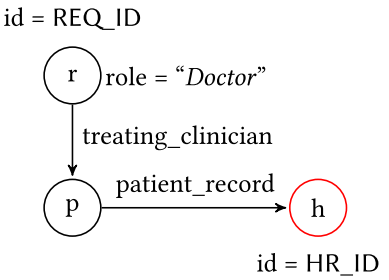


Fig. 7. Combined graph pattern for the *read_hr* method and the corresponding policy.

where GP_x enforces a role restriction of "Doctor" for the requestor, and GP_y enforces a *treating_clinician* relation between the requestor and the patient. Figures 3, 4, and 5 present the graph pattern representations for GP_x, GP_y, and the enforced policy ($\sqcup_{c_{read\_hr}}\{GP_x, GP_y\}$), respectively. The nodes r, p, and h represent the requestor, patient, and health record, respectively.

3.3 Weaving Policies and Methods

Now that we have defined access control policies and the process of computing enforced policies, we can define how to weave methods with their corresponding enforced policies.

We first translate the authorization request, $(m, s) \in MD \times S$, which is specified using Nano-Cypher, to a pair, $(GP_r, Info)$, where GP_r is the Nano-Cypher query translated to a graph pattern and $Info : CA(C(GP_r)) \nrightarrow O$ is a partial function used to assign some of the vertices to objects for the query evaluation. Intuitively, *Info* represents the contextual information gleaned from the request (e.g., the identity of the requestor) to reduce the search space for the query evaluation. Section 4.1 describes how to translate a Nano-Cypher query to a graph pattern. In the next step, we combine GP_r with the enforced policy $Pol(MC(m)) = GP_p$ and obtain the resulting graph pattern, GP_{rp}. More formally:

$$GP_{rp} = \sqcup_{MC(m)}\{GP_r, GP_p\}.$$

We evaluate the pair, $(GP_{rp}, Info)$, using the *GP-Eval* algorithm (Section 4.3) to find mappings from the protection state that satisfy the graph pattern. During this step, we also generate the result sets by collecting the vertices stated in the return statement.

This model can be simplified if we express both the methods and policies using either Nano-Cypher or graph patterns. In fact, if we also express the policies using Nano-Cypher, then we will need to translate the policies to the internal graph pattern data structure that we use for evaluating queries. Alternatively, we can express the methods using graph patterns, in which case, we no longer require the translating step, as we are working with graph patterns directly. We chose to present the model in this manner so we can cover both cases and provide a very brief description on how to switch to one of the variations. Figure 6 presents a visual graph pattern representation of the read_hr query specified in Example 9. Note that the REQ_ID and HR_ID are vertex attribute requirement values to be provided at start of the query execution, and the vertex h (colored red) is the only member of *Ret*. Example 16 extends Example 9 and presents the combined query and the enforced policy for the *read_hr* method. Figure 7 presents the combined graph pattern for the query and the enforced policy.

Example 16. Combining the Nano-Cypher query for the *read_hr* method and the corresponding access control policy results in the following Nano-Cypher query:

```
Match (requestor)
Match (patient) − [: patient_record]− > (health_record)
Where requestor.id = REQ_ID AND health_record.id = HR_ID
Match (requestor) − [: treating_clinician]− > (patient)
Where requestor.role = "Doctor"
Return health_record.
```

We discuss how to translate between Nano-Cypher and graph patterns as well as provide more examples of Nano-Cypher queries and visual representations of their equivalent graph patterns in Section 4.1. Although using Nano-Cypher adds an additional step in the whole process, it has been reported that the Cypher query language is easy to use for specifying graph traversals [40]. Since Nano-Cypher is based on the Cypher query language, it should not be any more complicated than Cypher for specifying simple graphs.

4 WORKING WITH GRAPH PATTERNS

In this section, we present how to translate a Nano-Cypher query to a graph pattern and vice versa, how to combine a set of graph patterns into a single graph pattern, and the algorithm for evaluating a graph pattern, that is, finding all of the distinct results for a given graph pattern query.

Given a function f, we use the notations $Dom(f)$ and $Ran(f)$ to refer to the domain and range of f, respectively.

4.1 Translating between Nano-Cypher and Graph Patterns

Nano-Cypher to Graph Pattern. Given a method m representing a Nano-Cypher query Q, with (VX, EX) as vertex and edge variable names, respectively, we can generate an equivalent graph pattern, $GP = (V, E, \Sigma, \Gamma_V, \Gamma_E, c, G_{ACT}, Ret)$, as follows: First, for each vertex variable $v \in VX$, generate a corresponding vertex $v' \in V$. Maintain this corresponding relationship through a mapping function $f : VX \rightarrow V$. For each edge variable $e \in EX$, if e is of type (v)-[$e : i$]->(u), then create an edge $i(f(v), f(u)) \in E$. Otherwise, if e is of type (v)<-[$e : i$]-(u), then create an edge $i(f(u), f(v)) \in E$. For each v<>u type clause in `Where` statements, create a mutual exclusion constraint $(f(u), f(v)) \in \Sigma$. For each vertex attribute clause, $v.va\ R\ val_{va}$, in `Where` statements, create a vertex attribute requirement $(f(v), va, R, val_{va}) \in \Gamma_V$. Similarly, for each edge attribute clause, $e.ea\ R\ val_{ea}$, in `Where` statements, create an edge attribute requirement $(i(f(u), f(v)), ea, R, val_{ea}) \in \Gamma_E$. Let $c = MC(m)$. Specify the function G_{ACT} using the variables names (in VX) that match the actor names in $CA(c)$. Finally, generate Ret based on all the vertices mentioned in the `Return` statement of Q and their corresponding mapping through f. Thus, $GP = (V, E, \Sigma, \Gamma_V, \Gamma_E, c, G_{ACT}, Ret)$.

Graph Pattern to Nano-Cypher. Given a graph pattern $GP = (V, E, \Sigma, \Gamma_V, \Gamma_E, c, G_{ACT}, Ret)$ one could generate a Nano-Cypher query, with (VX, EX) as vertex and edge variable names, respectively, as follows: For each vertex $v \in V$, create a corresponding vertex variable $v' \in VX$, and maintain the corresponding relationship through a mapping function $f : V \rightarrow VX$. For each edge, $id(u, v) \in E(G)$, create a corresponding edge variable $e' \in EX$, and maintain the corresponding relationship through a mapping function $f' : E \rightarrow EX$. For each edge, $id(u, v) \in E(G)$, with $f'(id(u, v)) = e'$, $f(u) = u'$, and $f(v) = v'$, generate a corresponding `Match` statement, "`Match` (v')-[$e' : id$]->(u')". For all vertices, $v \in V$, generate a `Match` statement, "`Match` (v)", only if v is not a member of any edges in E. Translate all mutual exclusion constraints, $(u, v) \in \Sigma$, with

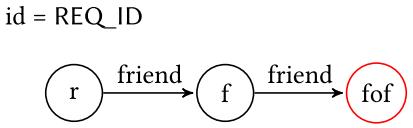


Fig. 8. Friend-of-friend.

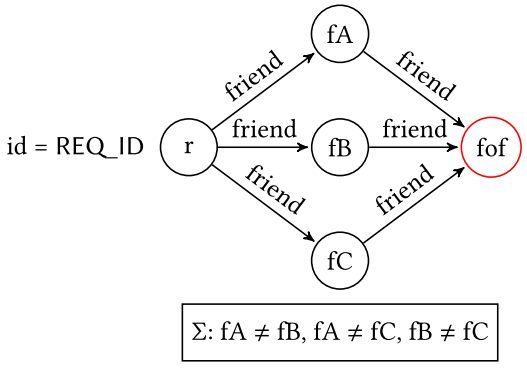


Fig. 9. At least 3 mutual friends.

$f(u) = u'$, and $f(v) = v'$, to "u'<>v'" clauses in a `Where` statement. Translate all vertex attribute requirements $(v, va, R, val_{va}) \in \Gamma_V$ to `Where` statements "`Where` $v.va\ R\ val_{va}$", and all edge attribute requirements $((id(u, v), ea, R, val_{ea}) \in \Gamma_E$, with $f'(id(u, v)) = e'$, to `Where` statements "`Where` $e.ea\ R\ val_{ea}$". Finally generate the `Return` statement based on the members of *Ret*.

Example 17 along with Figure 8 and Example 18 along with Figure 9 present the *friend-of-friend* and *at least 3 mutual friends* as Nano-Cypher queries as well as their visual graph pattern representations, respectively.

Example 17. Friend-of-Friend

```
Match (requestor) − [: friend]− > (f) − [: friend]− > (fof)
Where requestor.id = REQ_ID
Return fof
```

Example 18. At least 3 mutual friends

```
Match (requestor) − [: friend]− > (fA) − [: friend]− > (fof)
Match (requestor) − [: friend]− > (fB) − [: friend]− > (fof)
Match (requestor) − [: friend]− > (fC) − [: friend]− > (fof)
Where requestor.id = REQ_ID AND fA ≠ fB AND fA ≠ fC
AND fB ≠ fC
Return fof.
```

Note that translating a Nano-Cypher query to a graph pattern results in only one possible graph pattern, while translating a graph pattern to a Nano-Cypher query has multiple possibilities. For a Nano-Cypher query Q, if we permute the `Match` statements while maintaining correctness, that is, each variable appears in a `Match` statement before appearing in a `Where` statement, then we have another query Q'. Q and Q' produce the same result when evaluated; however, the Neo4j query evaluation engine might not evaluate them in the same order. This difference in evaluation order might cause one version of the query to perform significantly faster than the other, while the results remain the same. Thus, the two queries are semantically equivalent but operationally inequivalent. Similarly, a Nano-Cypher query and its graph pattern translation are semantically equivalent but operationally inequivalent.

4.2 $\sqcup_c : 2^{\mathbf{P}} \rightarrow \mathbf{P}$

$\sqcup_c : 2^{\mathrm{P}} \rightarrow \mathrm{P}$ combines a set of graph patterns into a single graph pattern. Definition 8 specifies how to translate the input graph patterns, $\{GP_1, \ldots, GP_k\}$, into a single graph pattern, GP'. The general idea here is that the combined graph pattern has a match if and only if every component pattern has a match. This is achieved by having a corresponding vertex in the combined graph pattern for each of the original vertices, where equivalent vertices are mapped together, and then generating the rest of the elements based on the vertex mapping.

To facilitate discussion, we will first define an equivalence relation between vertices across multiple graph patterns.

Definition 7. **Equivalence Relation (≡)**: Given an AReBAC protection state, the attribute support schema χ, and a set of graph patterns $\{GP_1, \ldots, GP_k\}$ with category $c_1, \ldots, c_k$, respectively, such that there exists $c \in \{c_1, \ldots, c_k\}$, $c \geq c_i$, and all $V(GP_i)$ are pairwise disjoint for all $1 \leq i \leq k$. Let

$$V_{\cup} = \bigcup_{1 \leq i \leq k} V(GP_i).$$

Then $\forall v \in V_{\cup}$ the equivalence relation (denoted ≡) is defined as follows:

$$v \equiv v, \text{and}$$
$$\forall 1 \leq i, j \leq k, \forall act \in CA(c_i) \cap CA(c_j), (G_{ACT}(GP_i))(act) \equiv (G_{ACT}(GP_j))(act).$$

Given a vertex $v \in V_{\cup}$, let $[v]_{\equiv}$ denote the set of all vertices that are equivalent to v.

Definition 8. $\sqcup_c : 2^{\mathrm{P}} \rightarrow \mathrm{P}$: Given an AReBAC protection state, the attribute support schema χ, and a set of graph patterns $\{GP_1, \ldots, GP_k\}$ with category $c_1, \ldots, c_k$, respectively, such that there exists $c \in \{c_1, \ldots, c_k\}$, $c \geq c_i$, and all $V(GP_i)$ are pairwise disjoint for all $1 \leq i \leq k$. Then $\sqcup_c\{GP_1, \ldots, GP_k\} = GP' = (V', E', \Sigma', \Gamma'_V, \Gamma'_E, c, G_{ACT}', Ret')$ is a graph pattern defined as follows:

$$V_{\cup} = \bigcup_{1 \leq i \leq k} V(GP_i), \tag{1}$$

$$V' = \{[v]_{\equiv} \mid v \in V_{\cup}\}, \tag{2}$$

$$G'_{ACT} : CA(c) \rightarrow 2^{V_{\equiv}} \text{is defined as} \tag{3}$$

$$G'_{ACT}(act) = [(G_{ACT}(GP_j))(act)]_{\equiv}$$

$$\text{where } 1 \leq j \leq k \text{ s.t. } act \in CA(GP_j),$$

$$f : V_{\cup} \rightarrow V' \text{ is defined as} \tag{4}$$

$$f(v) = [v]_{\equiv},$$

$$E' = \{id(f(u), f(v)) \mid \exists i \in [1, k].\ id(u, v) \in E(GP_i)\}, \tag{5}$$
$$\Sigma' = \{(f(u), f(v) \mid \exists i \in [1, k].\ (u, v) \in \Sigma(GP_i)\}, \tag{6}$$
$$\Gamma'_V = \{(f(v), va, R, val) \mid \exists i \in [1, k].\ (v, va, R, val) \in \Gamma_V(GP_i)\}, \tag{7}$$
$$\Gamma'_E = \{(id(f(u), f(v)), ea, R, val) \mid \exists i \in [1, k].\ (id(u, v), ea, R, val) \in \Gamma_E(GP_i)\}, \tag{8}$$
$$Ret' = \{f(v) \mid \exists i \in [1, k].\ v \in Ret(GP_i)\}. \tag{9}$$

We start by generating the set $V_{\cup}$, which collects the set of vertices from each input graph pattern (line 1). We then define the set V', which combines equivalent vertices (line 2) and also

serves as the vertex set for the combined graph pattern. Next, we generate the function G'_{ACT} for our combined graph pattern (line 3). Note that, since c is superior to all of the input graph pattern categories, then for $1 \leq i \leq k$ $CA(C(GP_i)) \subseteq CA(c)$. Before we generate the remaining components of the combined graph pattern, we first define the function $f : V_{\cup} \rightarrow V'$ that maps a vertex from an input graph pattern to a vertex in V' (line 4). Specifically, it maps a vertex v to the vertex $[v]_{\equiv}$. We can now generate the rest of the components easily, using the function f when needed (lines 5–9).

4.3 Graph Pattern Evaluation

Graph pattern evaluation is essentially a graph homomorphism problem with additional constraints; hence, we refer to the problem as *constrained graph homomorphism*. The graph homomorphism problem asks: Given graphs G and H, does there exist a mapping $M : V(G) \rightarrow V(H)$ such that if $(u, v) \in E(G)$, then $(M(u), M(v)) \in E(H)$? However, constrained graph homomorphism introduces mutual exclusion constraints and attribute requirements to the graph homomorphism problem. Here, G is the input graph pattern and H is the authorization graph. Recall, to avoid ambiguity, we use the terms "vertex" and "edge" to refer to the entities in the input graph pattern, and "node" and "relationship" to refer to the entities in the authorization graph.

The algorithm *GP-Eval* describes how to find the mappings that satisfy the graph pattern and provide the correct response to the requested query execution. *GP-Eval* is a Forward Checking algorithm [31], augmented with Live-End Directed Backjumping (FC-LBJ), which is a novel contribution of this work. Live-End Directed Backjumping is described in Section 4.3.1. *GP-Eval* is motivated by two main design objectives: (1) We wish to reduce/minimize the number of database accesses. This idea was motivated by the performance evaluation conducted by Rizvi et al. for their implementation of ReBAC in OpenMRS [36]. In their experiments it was revealed that the majority of the performance overhead came not from the backtracking within the hybrid logic model checker, but rather database accesses. This idea was also reinforced by our preliminary experiments, where an earlier version of *GP-Eval* focused on minimizing the number of memory operations rather than the number of database accesses. (2) Given a graph pattern $GP = (V, E, \Sigma, \Gamma_V, \Gamma_E, c, G_{ACT}, Ret)$, we are not interested in all of the mappings that satisfy the graph pattern, but rather only the mappings that are unique in mapping the vertices specified in *Ret*. This concept is important, as it reduces the amount of work required to compute the result set. For example, consider the query friend-of-friend where there are three vertices, requestor, common-friend, and friend-of-friend, and we are only interested in the friend-of-friend vertex. If Alice, the requestor, has Bob as a friend-of-friend through Carol, Dan, and Eve, then we only need a single mapping for the result set $\{friend\text{-}of\text{-}friend \mapsto Bob\}$, rather than three equivalent mappings. We refer to these instances as *equivalent classes* [17, 26].

Another way to look at graph pattern evaluation is as a *Constraint Satisfaction Problem (CSP)* [37] that asks, given a set of variables and constraints, assign values to those variables such that the assignments satisfy the constraints. Here, vertices of the graph pattern V are the variables; the edges E, mutual exclusion constraints Σ, and attribute requirements Γ_V, Γ_E are the constraints; and the nodes of the attribute-supporting edge-labeled and directed graph V' (i.e., the authorization graph $G' = (V', E', A_V, A_E)$) are the possible assignment values, that is, the domain for each variable in V. Since we are working with an authorization graph, it would be unreasonable to allow every node as a candidate for every vertex as a starting point. Thus, we leverage the authorization request context to glean additional information about the nodes involved, such as the identity of the requestor. In some cases, we may have more than one fixed vertex, as shown in Examples 6 and 8 where Alice is the requestor and bob_hr is a unique identifier for the actor health_record. In addition to the fixed vertices, we utilize the relationships of already assigned nodes to generate a smaller domain for their corresponding neighboring vertices. This way, every time we assign

a new node, we do not need to verify if the assignment is consistent with the current mapping. Below, we provide an overall description of the FC-LBJ algorithm (Section 4.3.1). We then break the algorithm, *GP-Eval*, down to three important components and present the function for each component: the initialization phase (Algorithm 1) (Section 4.3.2), the backtracking search phase (Algorithm 2) (Section 4.3.3), and the forward checking (Algorithm 3) (Section 4.3.4). We also discuss the variable ordering heuristic (Section 4.3.5) and the hub encounter limitation (Section 4.3.6).

4.3.1 FC-LBJ. FC-LBJ is based on the principles of LBJ, which is a novel contribution of this work: Forward Checking (FC), Conflict-Directed Backjumping (CJB), and FC-CBJ. FC is a "look ahead" scheme that extends the variable assignment by performing a single assignment, looks ahead by visiting the remaining unassigned variables and removing all candidates from their corresponding domains that are inconsistent with the current assignment. Thus, when the algorithm moves forward with further variable assignments, we already know that all of the remaining candidates for the unassigned variables are consistent with the current assignment [31]. FC is further described in Section 4.3.4.

CBJ is a backtracking algorithm that builds upon Backjumping (BJ). In the case of a deadend, BJ jumps back directly to the most recent conflicting variable, i.e., the variable, u, which precluded one or more values from the domain of the current variable, v. If, on jumping back to u, its domain does not have any values remaining, then BJ chronologically backtracks. This is where CJB extends BJ. If, on jumping back to u, its domain does not have any values remaining, then CJB jumps back to the next most recent variable that conflicted with either u or v. As the name suggests, FC-CBJ combines the "look ahead" scheme of FC with the backtracking scheme of CJB [31].

FC-LBJ augments FC by providing a backjumping scheme to improve performance for the graph pattern evaluation problem. While working with the graph pattern evaluation problem, we found a new form of thrashing, which we dub *live-thrashing*, where the algorithm finds several results that belong to the same equivalent class (i.e., the values assigned to variables in *Ret* does not change). Therefore, instead of chronological backtracking, we backjump to the most recently assigned variable that either (i) is a member of *Ret*, (ii) affected the domain of one or more members of *Ret*, or (iii) affected the domain of the current variable. It is easy to see that LBJ shares similarities with CBJ. In the case of a deadend, LBJ performs backjumping in the same manner as CJB. However, if the current variable assignment could be extended, LBJ performs backjumping to find a new result that belongs to a different equivalence class than the most recent result, while CBJ would perform chronological backtracking [11]. This behavior is inspired by the second design objective behind *GP-Eval*. In Section 4.3.3, we describe how we achieve the goals of LBJ.

It is important to note that LBJ depends on the condition that not all variables are part of the result (i.e., $Ret \subset V$). In fact, if $Ret = V$, then LBJ behaves in exactly the same way as CBJ. Give two almost identical graph pattern queries, $GP_1 = (V, E, \Sigma, \Gamma_V, \Gamma_E, c, G_{ACT}, Ret_1)$ and $GP_2 = (V, E, \Sigma, \Gamma_V, \Gamma_E, c, G_{ACT}, Ret_2)$ such that $Ret_1 \subset Ret_2$, GP_2 will have no fewer equivalent classes than GP_1, and fewer equivalent classes yields a greater potential for backjumps. In the extreme case where $Ret = V$, each complete and valid assignment produces an equivalent class of its own, therefore the overall algorithm is no different than FC-CBJ.

4.3.2 Initialization Phase. *GPEvalInit* (Algorithm 1) functions as the initialization phase for *GP-Eval*. The function consumes an input graph pattern *GP* and a non-empty mapping for fixed nodes *Info*. We also introduce some local and global variables: $G' = (V', E', A_V', A_E')$ represents the authorization graph, *RS* is the result set, and *Cand* is the map from each vertex in the graph pattern *GP* to a set of candidate nodes in the authorization graph V'. Note that global variables are available to all functions called throughout the overall algorithm, while the local variables are only available to the function where they are declared. We start off by initializing the global and local

ALGORITHM 1: GPEvalInit

Input: The graph pattern: $GP = (V, E, \Sigma, \Gamma_V, \Gamma_E, c, G_{ACT}, Ret)$
Input: Context information: $Info : CA(c) \rightarrow V'$
Global: Attribute-supporting edge-labeled and directed graph: $G' = (V', E', A_V', A_E')$
Global: The result set RS is a finite set of functions, each having the function signature $Ret \rightarrow V'$)
Local: The candidate sets for the vertices: $Cand : V \rightarrow 2^{V'}$

```
1 Initialize RS and Cand to ∅;
2 foreach v ∈ Dom(Info) do
3 |   Cand(v) ← {Info(v)};
4 if PreCheck {GP, Cand} then
5 |   GPEvalRec (GP, Cand, ∅, ∅);
6 |   return RS;
7 else
8 |   return ∅;
```

variables (line 1). We then start populating the vertex candidate sets, *Cand*, based on the contextual information provided by *Info* (the for loop on line 2). The *PreCheck* function checks if the candidate provided for each vertex satisfies the attribute requirements for the corresponding vertex. If the attribute requirements are satisfied, then *PreCheck* returns $\top$, else it returns $\bot$. If *PreCheck* returns $\top$ (line 4), then we invoke the backtracking search phase and finally return the result set *RS*; otherwise, we return an empty result set (lines 7–8) to signal the fact that no match is found.

4.3.3 Recursive Phase. GPEvalRec (Algorithm 2) is the backtracking search phase of *GP-Eval*. Along with the graph pattern, *GP*, and the candidates sets, *Cand*, the function consumes the vertex assignments, *Assn*, and sets of incoming conflicts, *ConfIn*, as input (described with Algorithm 3). *GPEvalRec* starts by checking if we already have a solution, i.e., complete and successful assignment that satisfies the graph pattern (lines 9–11). If we do, then we update *RS*.[4] Note that *RS* is only updated if the assignment is unique in mapping the members of *Ret* (i.e., the result produces a new equivalent class). After the update, we step back to the previous call stack by returning *Ret* to indicate that we wish to backjump to the most recent member of *Ret*. If we have not yet found a solution, then we continue the search phase by initializing the local variables (lines 12–14), and picking the next vertex v (line 15). The restrictions on v are (1) v is not already assigned to a node, and (2) the candidate set of v is *populated* (i.e., at least one of the immediate neighbors of v is already assigned to a node, or the candidate for v was provided by *Info*). We then *filter* out the candidates for v that would violate mutual exclusion constraints against already assigned vertices (line 16) and then iterate through the remaining candidates.[5] Note that we pass parameters by reference, and thus, *MexFilter* alters $Cand(v)$ and $ConfIn(v)$. We obtain $Cand'$ by removing v from the domain of *Cand*, $Assn'$ by updating *Assn* with $\{v \mapsto n\}$, and $ConfIn'$ by copying *ConfIn* (lines 18–20). We then perform forward checking with the updated assignment (line 21). The *FC* function updates $Cand'$, $ConfIn'$, and *ConfOut* as a side-effect. If the forward checking is valid, then we update the *deadEnd* flag (line 23) and recurse with the updated maps (line 24). If the recursive call returned with a nonempty set *Jump*, then this indicates that the search either found a solution or ran into a deadend and needs to backjump. Thus, if *Jump* is not an empty set and $v \notin Jump$, then we continue

[4] We do not store every vertex in the *Assn* map, but rather only the vertices that are part of the return statement.
[5] The terms *populate* and *filter* are described in Section 4.3.4.

ALGORITHM 2: GPEvalRec

Input: *GP* and *Cand* as specified in Algorithm 1
Input: The vertex assignment: $Assn : V \to V'$
Input: The incoming conflict sets for the vertices: $ConfIn : V \to 2^V$
Local: A candidate sets for the vertices: $Cand' : V \to 2^{V'}$
Local: A vertex assignment: $Assn' : V \to V'$
Local: The set of vertices for backjumping: $Conflicts \subseteq 2^V$
Local: The set of outgoing conflicts: $ConfOut \subseteq 2^V$
Local: The current target vertex: $v \in V$

9 **if** $|V| = |Dom(Assn)|$ **then**
10 $\quad RS \leftarrow RS \cup Assn\,|_{Ret}$;
11 $\quad$ **return** *Ret*;
12 $deadEnd \leftarrow \top$;
13 $ConfOut \leftarrow \emptyset$;
14 $Conflicts \leftarrow \emptyset$;
15 $v \leftarrow$ **PickNextVertex** (*GP*, *Assn*, *Cand*);
16 **MexFilter** (*GP*, *Assn*, $Cand(v)$, *ConfIn*);
17 **forall the** $n \in Cand(v)$ **do**
18 $\quad Cand' \leftarrow Cand \setminus \{v\}$;
19 $\quad Assn' \leftarrow Assn \cup \{v \mapsto n\}$;
20 $\quad ConfIn' \leftarrow ConfIn$;
21 $\quad valid \leftarrow$ **FC** (*GP*, *Assn*′, *Cand*′, *v*, *ConfIn*′, *ConfOut*);
22 $\quad$ **if** *valid* **then**
23 $\quad\quad deadEnd \leftarrow \bot$;
24 $\quad\quad \mathcal{J}ump \leftarrow$ **GPEvalRec** (*GP*, *Cand*′, *Assn*′, *ConfIn*);
25 $\quad\quad$ **if** $\mathcal{J}ump \neq \emptyset \wedge v \notin \mathcal{J}ump$ **then**
26 $\quad\quad\quad$ **return** $\mathcal{J}ump$;
27 $\quad\quad$ **else**
28 $\quad\quad\quad Conflicts \leftarrow Conflicts \cup \mathcal{J}ump$;
29 **if** (*deadEnd*) **then**
30 $\quad Conflicts \leftarrow Conflicts \cup ConfIn(v) \bigcup_{u \in ConfOut} ConfIn(u)$;
31 $\quad$ **return** *Conflicts*;
32 **else**
33 $\quad Conflicts \leftarrow Conflicts \cup Ret \bigcup_{u \in Ret} ConfIn(u)$;
34 $\quad$ **if** $ConfOut \neq \emptyset$ **then**
35 $\quad\quad Conflicts \leftarrow Conflicts \cup ConfIn(v) \bigcup_{u \in ConfOut} ConfIn(u)$;
36 $\quad$ **return** *Conflicts*;

backjumping (lines 25–26); otherwise, either $\mathcal{J}ump$ is an empty set or $v \in \mathcal{J}ump$, we update the set *Conflicts* (line 28) and continue with the next candidate for v.

Once we have exhausted all of the candidates for v, we need to decide where to backjump. If none of the candidates assignments for v could be extended (i.e., deadend) (line 29), then we update the conflicts set using CBJ (line 30) and backjump (line 31). However, if at least one of the candidates assignments for v could be extended, then we update the conflicts set using LBJ (lines 33–35) and

ALGORITHM 3: FC

```
   Input: GP and Cand as specified in Algorithm 1
   Input: Assn, v, ConfIn, and ConfOut as specified in Algorithm 2
   Local: The set of relevant edges: Edges ⊆ 2^E
   Local: The result of neighborhood retrieval query: neighbors ⊆ 2^V'
   Local: The other vertex from an edge: u ∈ V
37 Edges = getRelevantEdges(V, E, v);
38 foreach e ∈ Edges do
39     u = getOtherVertex{e, v};
40     if u ∉ Dom(Assn) then
41         neighbors = getNeighbors(GP, e, v, Assn(v));
42         if (Cand(u) = ∅) ∨ (Cand(u) ⊈ neighbors) then
43             ConfIn(u) ← ConfIn(u) ∪ ConfIn(v) ∪ {v};
44         if Cand(u) = ∅ then
45             Cand(u) ← neighbors;
46         else
47             Cand(u) ← Cand(u) ∩ neighbors;
48         if Cand(u) = ∅ then
49             ConfOut ← ConfOut ∪ {u};
50             return ⊥;
51 return ⊤;
```

backjump (line 36). The inclusion of LBJ within the algorithm provides a significant performance boost over simply using FC or FC-CJB. Table 1 presents the results of the experiments comparing the FC-LBJ version of *GP-Eval* with FC-CJB and FC.

4.3.4 Forward Checking. Forward checking is a "look ahead" CSP technique that extends the variable assignment by performing a single assignment, looks ahead by visiting the remaining unassigned variables (i.e., future variables) and removing all candidates from their corresponding domains that are inconsistent with the current assignment, i.e., the partial solution. Thus, when the algorithm moves forward with further variable assignments, we already know that all of the remaining candidates for the variable are consistent with the current assignment. If the "look ahead" step results in the candidate set, for a future variable, being emptied, then the algorithm tries a new value for the current variable. If there are no remaining values for the current variable, then the algorithm backtracks [31]. We use forward checking to reduce/minimize the number of database accesses, as specified by our first design objective. Note that for *GP-Eval*, most of the database accesses occur within the *FC* function calls.[6] If we performed forward checking in its original form, then we would have to check consistency for every unassigned vertex in the graph pattern; however, our problem only requires us to check if the given assignment is consistent with the vertex's immediate neighbors. This is achieved through a novel application of forward checking that we dubbed *Populate and Filter*. We will first discuss the application of forward checking to *GP-Eval* and then describe the *FC* function (Algorithm 3).

For *GP-Eval*, forward checking serves three important functions: *populate*, *filter*, and *validate*. A naive approach for evaluating graph pattern queries would be to start off with V' as the candidate set of each vertex in V. Even with forward checking, it is easy to see how this would lead

[6]The *PreCheck* function also performs database accesses, but the function is only called once.

to a lot of filtering of candidate sets based on partial solutions. Thus, the candidate set for each vertex is empty until one of its immediate neighbors is assigned a node; i.e., when a node n is assigned to vertex v, the algorithm visits all of the unassigned immediate neighbors of v, and if the candidate set of neighbor u is empty, then $Cand(u)$ is *populated* by neighbors of n that satisfy the corresponding relationship and attribute requirements, N:

$$Cand(u) = N.$$

However, if $Cand(u)$ is already populated, the function *filters* the candidates set by computing its intersection with the neighbors of n that satisfy the required relationship and attributes:

$$Cand(u) = Cand(u) \cap N.$$

This *validates* that the candidates for the vertex u are also consistent with the assignment (v, n). If, in either case, the resulting candidates set is empty, then a deadend has been encountered. Thus, the assignment cannot be extended with (v, n), and the function returns false along with the updated candidates map. Alternatively, if all of the resulting candidate sets are non-empty, then the function returns true along with the updated candidates map.

A limitation forced by this approach is that *GP-Eval* depends on having at least one fixed vertex to use as the starting point. Note that this is not mandatory; however, it does have a significant impact on the performance [30]. In cases where no fixed vertices are provided, the initial candidates set is obtained using the attribute requirements. Second, each vertex should be connected to some fixed vertex. This allows *GP-Eval* to populate the candidate set for each vertex.

The *FC* function (Algorithm 3) performs forward checking for *GP-Eval*. The function consumes the graph pattern, *GP*, the candidate sets, *Cand*, the vertex assignments, *Assn*, the current target vertex, v, and the sets of incoming and outgoing conflicts, *ConfIn* and *ConfOut*, respectively. We start by obtaining the relevant edges for v[7] (line 37) and iterate through each edge (line 38). For an edge e, we first obtain the *other* vertex u[8] (line 39), and only consider the edge if u is not already assigned (line 40). If u is not already assigned, then we obtain the *neighbors* of $Assn(v)$ that satisfy the edge and attribute requirements (line 41). If *neighbors* will alter $Cand(u)$ (line 42), then we add v as an incoming conflict for u (line 43). Next, we update $Cand(u)$ with *neighbors*. If $Cand(u)$ is empty, then we populate it ($Cand(u) = N$), otherwise, we filter it ($Cand(u) = Cand(u) \cap N$) (lines 44–47). After the update, if $Cand(u)$ results in an empty set, then $\{v \mapsto n\}$ is not a valid extension to the assignment. We then update the outgoing conflicts for v and return $\perp$ (if statement line 48). Alternatively, if we are able to iterate through all of the edges without failing, then $\{v \mapsto n\}$ is a valid extension to the assignment, and we end the function by returning $\top$ (line 51).

4.3.5 Variable Ordering. The algorithm orders the variables dynamically based on the size of the candidates set. According to the wisdom of Reference [39, §4.6.1], we pick the variable with the smallest candidate set (i.e., domain size). Note that this strategy does allow us to pick the fixed nodes first, since they have the smallest candidate size possible. Other variable ordering strategies could be to pick the variable with most/least assigned/unassigned immediate neighbors, or some heuristic involving the variables in *Ret*. We leave the evaluation of variable ordering heuristics for future work.

4.3.6 Hub Encounter Limitation. Recall that the first design objective for *GP-Eval* is to reduce/minimize database accesses due to the performance cost of loading data from secondary storage. However, the algorithm does end up loading and storing part of the dataset in memory as

[7] An edge $id(x, y)$ is relevant to vertex v if $v = x$ or $v = y$.
[8] For a relevant edge $e = id(x, y)$, if $v = x$, then $u = y$, else $u = x$.

the neighborhood retrievals (*neighbors*) and candidate sets (*Cand*). If the algorithm ends up storing a significant portion of the dataset in memory, then we lose the benefit of the first design object and possibly suffer expensive operations (e.g., $Cand(u) \cap N$). Note that the candidate set for vertex, u, is largest when it is populated, and its size is based on the node, n, which is assigned to u's immediate neighbor v. Therefore, if n has a relatively high degree (i.e., n is a hub [25]), then there is a chance that $Cand(u)$ will store a significant portion of the dataset in memory, rendering *GP-Eval* to be an undesired algorithm for the problem. Although hubs are uncommon in scale-free networks [9, 27], by their nature, it would be unreasonable to assume that they are never encountered. In Section 5.2, we explore how often we encounter hubs given various datasets and test cases.

5 *GP-EVAL* EVALUATION

We conducted two sets of experiments for evaluating *GP-Eval*. The first is an extension of the performance evaluation experiments conducted in Reference [35], where we compare the updated version of *GP-Eval* (with FC-LBJ) against the original version (with only FC) as well as the intermediate version (with FC-CJB) [11]. These experiments are presented in Section 5.1. The second set of experiments is there to address the hub encounter limitation of the algorithm (Section 4.3.6). These experiments are presented in Section 5.2.

5.1 Performance Evaluation

We conducted an empirical study to compare the proposed query evaluating and authorization checking algorithm, *GP-Eval* (FC-LBJ) (Section 4.3), to its original version as presented in Reference [35, §4.2] (FC), as well as an FC-CJB version of the algorithm. We compared the performances under various configurations. The study was conducted in a simulated environment using synthetic data. Below, we describe our experimental set-up followed by the results of the experiments comparing the algorithm schemes.

The Protection State. For the protection state, we used the *"soc-Slashdot0922"* social network dataset, obtained from the Stanford Large Network Dataset Collection (SNAP) [5], to construct the authorization graph. The graph consisted of 82,168 nodes and 948,464 directed edges. We generated seven relationship identifiers and randomly assigned a relation identifier to each edge. We followed the experimental design of Rizvi et al. [34, 36] and chose seven relation identifiers, which were based on the case study presented by Fong [21]. In addition to an identifier, each edge was also randomly assigned a *weight* attribute with an integer value (between 1 to 10). The attributes for each node were provided by *"Census-Income (KDD) Dataset,"* obtained from the UCI Machine Learning Repository [28]. This dataset consisted of 199,523 (non-test) rows of data and 40 attribute types, with each row representing a single entity; however, not every entity had all 40 attributes. For a single entity, each attribute had at most one value that was treated as either a String or an integer. Since this dataset consisted of more entities than the number of nodes in the authorization graph, we randomly sampled 82,168 entities from the dataset and assigned attributes to nodes using a randomly generated one-to-one mapping between the nodes and the sampled entities.

Generating Test Cases. We extracted a subset of the authorization graph to generate the graph patterns used for matching. The purpose of this step was to ensure that the result set of each test case was non-empty. Each test case consists of a graph pattern *GP*, which is evaluated by the algorithm. Table 1 summarizes the various profiles of test cases, where each profile is composed of test cases of a certain size, i.e., the number of vertices in the graph patterns. The order of the profiles reflects the increasing size of test cases, where Profile 1 consists of the smallest graph patterns, with each graph pattern containing 5 vertices, and Profile 6 consists of the largest graph patterns, with each graph pattern containing 13 vertices. We vary the graph pattern sizes to observe its

Table 1. The Size of the Profiles and the Average Performance Time for Each Evaluation Scheme

	$\|GP\|$		GP-Eval (FC-LBJ)	GP-Eval (FC-CBJ)	GP-Eval (FC)
Profile 1	5	# Finished	1000	1000	1000
		Avg. Time	0.002 sec	0.003 sec	0.003 sec
Profile 2	7	# Finished	999	993	993
		Avg. Time	0.019 sec	0.041 sec	0.039 sec
Profile 3	9	# Finished	992	934	941
		Avg. Time	0.064 sec	0.195 sec	0.204 sec
Profile 4	10	# Finished	978	892	897
		Avg. Time	0.041 sec	0.289 sec	0.285 sec
Profile 5	11	# Finished	976	831	831
		Avg. Time	0.069 sec	0.311 sec	0.305 sec
Profile 6	13	# Finished	972	683	692
		Avg. Time	0.107 sec	0.502 sec	0.512 sec

impact on the algorithm performance. For each graph pattern, we also randomly pick 1, 2, or 4 vertex attribute requirements, 1, 2, or 4 edge attribute requirements, 0, 1, or 2 mutual exclusion constraints, and 1, 2, or 4 members of *Ret*. Appendix A describes the process of extracting the graph patterns for each test case. In total, we generated six test case profiles with 1K test cases per profile. The test cases are available online [33].

Running Test Cases. We first ran 250 randomly generated tests of various sizes to warm up the cache. After the warmup tests, we ran all 6K test cases, in order, using the targeted algorithm version. Note that the warmup test cases were generated independently from the experiment test cases, and thus, the experiment test cases do not rely on the required data to be already cached. We chose to conduct the warmup tests based on the results reported in Reference [36], which stated that the neighborhood retrieval queries start off slow and stabilize after 250 invocations. We set a six-second time limit for each test case based on experiments in Reference [35]. If the test case did not finish within the time limit, we simply halted the algorithm. If the test case did complete within the time limit, then we recorded its completion time.

Results and Discussions. The experiments were conducted on a desktop machine with an AMD FX-8350 8-core processor (16 MB cache), 16 GB RAM (1,866 MHz, DDR3), and 840 EVO solid state drive (SSD) running Windows 10, with 10,240 MB dedicated to the JVM. The project was implemented in Java 8 using Neo4j version 3.4.0. Table 1 also shows the number of instances that were successfully evaluated within the six-second time limit, for each algorithm, as well as the average time over the successful instances.

As mentioned above, these experiments are an extension of the performance evaluation experiments conducted by Rizvi et al. [35]. The performance evaluation in Reference [35] compared *GP-Eval* (FC) against Neo4j's Cypher evaluation engine and showed that *GP-Eval* (FC) performed significantly faster than Neo4j's Cypher evaluation engine. This was especially the case for graph patterns with higher number of nodes, e.g., Profile 7 [35, §5], where *GP-Eval* (FC) was able to complete 616 evaluations out of 1K with an average completion time of 0.630 second. For the same set of queries, Neo4j's Cypher evaluation engine was able to complete only 71 evaluations and had an average completion time of 3.063 seconds. Table 1 shows that *GP-Eval* (FC-LBJ) is several orders of magnitude faster than *GP-Eval* (FC), and thus significantly more competitive than Neo4j's Cypher evaluation engine.

Surprisingly, there is no significant difference between the FC and FC-CBJ versions of *GP-Eval*. The observation implies that the CBJ scheme does not contribute towards these types of problem

Table 2. Performance Evaluation Metrics

	Duplicate Results Ratio		Total Number of Assignments	
Profile	FC-LBJ	FC	FC-LBJ	FC
	Avg. (Std. Dev.)	Avg. (Std. Dev.)	Avg. (Std. Dev)	Avg. (Std. Dev.)
1	1.204 (3.864)	17.351 (303.225)	47.600 (356.238)	119.028 (1.6×10^3)
2	1.122 (1.466)	6.8×10^4 (1.8×10^5)	283.238 (2.5×10^3)	1.7×10^4 (2.5×10^5)
3	1.261 (1.897)	1.2×10^5 (1.7×10^6)	3.5×10^3 (5.3×10^4)	2.5×10^5 (2.3×10^6)
4	1.422 (2.641)	2.4×10^5 (2.2×10^6)	5.5×10^3 (6.4×10^4)	5.4×10^5 (2.8×10^6)
5	1.689 (3.881)	3.0×10^5 (1.8×10^6)	4.6×10^3 (3.5×10^4)	9.1×10^5 (3.6×10^6)
6	1.495 (4.193)	6.9×10^5 (2.9×10^6)	6.1×10^3 (5.2×10^4)	2.1×10^6 (5.1×10^6)

instances. However, there is a significant difference between the FC-LBJ and the other versions of *GP-Eval*. This is because FC-LBJ aims to reduce finding multiple results that belong to the same equivalent class (design objective 2). To measure this phenomenon, we introduced the Duplicate Results Ratio, as presented in Table 2. The purpose of the Duplicate Results Ratio is to measure how often the algorithm finds a solution that satisfies the graph pattern, but does not contribute the Result Set. This situation occurs when the solution belongs to an already-discovered equivalent class. The Duplicate Results Ratio is computed by taking the ratio of "the total number of solutions found" to "the number of equivalent classes (distinct results)" for the test cases. For all six profiles, the Duplicate Results Ratio for FC is orders of magnitude greater than the Duplicate Results Ratio for FC-LBJ. Hence, *GP-Eval* (FC) does a lot of work that is eventually thrown away. Recall that this observation reflects back to the motivation regarding performing query evaluation and authorization checking simultaneously. In both cases, we wish to prevent doing work that is eventually thrown away.

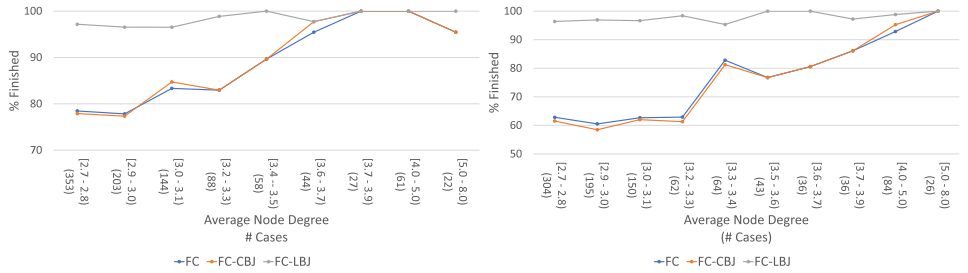
Based on the duplicate results observation, we also recorded the total number of vertex assignments made by the algorithm for each test case. The Total Number of Assignments columns in Table 2 provide the average total number of vertex assignments made by the algorithm for the test cases in each profile. This provides yet another metric for comparing the performance of the FC-LBJ scheme to FC. Once again it is evident that *GP-Eval* (FC-LBJ) is able to obtain the same results more efficiently than *GP-Eval* (FC). Note that the results in Table 2 were obtained through rerunning the test cases, but with a 60-second timeout limit. We included the results for each test case regardless of whether the test case evaluation was completed or not. For the cases where the evaluation had not completed, we use the partial results (of the work completed within the time limit).

To study how the topology of the graph pattern affects the performance of *GP-Eval*, we reclassified the original test cases based on (i) the diameter of the graph pattern and (ii) the average vertex degree. Table 3 shows the performance of the algorithms when the test cases are classified based on the diameter of the graph patterns. Note that, since the test cases were not originally designed with the diameter in mind, the classifications do not contain the same number of test cases. As expected, the graph patterns with the smaller diameter are evaluated faster than the test cases with the larger diameter. This is because the smaller diameter graph patterns have a stronger connectivity than the larger ones. For example, the graph patterns with a diameter of 1 have an edge between every pair of vertices. The phenomenon allows the algorithm to filter candidate sets frequently, thus reducing the search space. A surprising result of this classification is how the performance of FC-LBJ compares to FC-CBJ and FC. For the largest diameter test cases, 95.82% of the cases were able to complete under FC-LBJ, while only 77.16% and 75.55% of the cases completed under FC-CJB and FC, respectively.

To further study the phenomenon observed in Table 3, we reclassify the test cases based on the average vertex degree. However, instead of using all 6K cases, we only use the test cases

Table 3. The Algorithm Performance Based on the Diameter of the Test Cases

		FC-LBJ	FC-CBJ	FC
Diameter: 1	# Finished (% Finished)	244 (100%)	241 (98.77%)	241 (98.77%)
# Cases: 244	Avg. Time	0.014 sec	0.013 sec	0.009 sec
Diameter: 2	# Finished (% Finished)	1,845 (99.46%)	1,774 (95.63%)	1,780 (95.96%)
# Cases: 1,855	Avg. Time	0.023 sec	0.096 sec	0.096 sec
Diameter: 3	# Finished (% Finished)	2,328 (99.02%)	2,078 (88.39%)	2,092 (88.98%)
# Cases: 2,351	Avg. Time	0.041 sec	0.237 sec	0.233 sec
Diameter: 4	# Finished (% Finished)	1,156 (97.06%)	963 (80.86%)	969 (81.36%)
# Cases: 1,191	Avg. Time	0.080 sec	0.281 sec	0.302 sec
Diameter: 5–8	# Finished (% Finished)	344 (95.82%)	277 (77.16%)	272 (75.77%)
# Cases: 359	Avg. Time	0.176 sec	0.542 sec	0.520 sec



a) Profile 5 Results b) Profile 6 Results

Fig. 10. Performance evaluation based on average vertex degree.

from Profiles 5 and 6 and still keep them separate. This approach allows us to observe how the connectivity between the vertices affects the performance of the algorithms. Since the number of vertices remains consistent, we know that the higher average vertex degree means higher connectivity within the graph pattern. Figure 10 presents percent of test cases that are able to finish within the six-second time limit, where the test cases are separated based on the average vertex degree. Once again, since the test cases were not originally generated with the average vertex degree in mind, the number of test cases between classifications varies. The results here support the observations from Table 3, i.e., the stronger the connectivity between the vertices, the easier it is to evaluate the graph patterns.

5.2 Hub Encounter Evaluation

The primary design objective of *GP-Eval* is to reduce/minimize the number of database accesses while evaluating graph pattern queries. A naïve approach here would be to load a significant portion of the dataset into memory and avoid querying the database explicitly. However, the focus of our contribution is on Neo4j, a secondary storage database, and loading a significant portion of the dataset into memory is not a valid option.

Unfortunately, we cannot get away with loading absolutely none of the dataset into memory. Whenever *GP-Eval* makes a neighborhood retrieval query, we load some part of the dataset into memory. This data persist as part of the candidate sets. Our claim is that *GP-Eval* is a viable approach to solving graph pattern queries, as long as we do not encounter a neighborhood retrieval

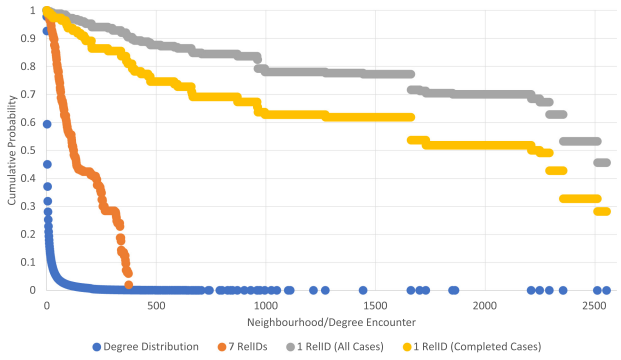
query with result size at least k, where $k \in \mathbb{Z}$ is some large integer dependent on the environment in which AReBAC is deployed. If a neighborhood retrieval query with result size at least k is encountered, then *GP-Eval* fails and we refer to another mechanism to solve the graph pattern query (e.g., the Cypher query evaluation engine provided by Neo4j).

The Protection State & Test Cases. We used two datasets and two configurations for each dataset to evaluate the hub encounter limitation. The first dataset was the *"soc-Slashdot0922"* authorization graph described above (Section 5.1). Instead of generating new test cases, we randomly picked a total of 250 test cases from profiles 4, 5, and 6. For the second configuration, we made a more difficult version of the 250 test cases by removing the relation identifiers, the attribute requirements (except for the one fixed vertex), and the mutual exclusion constraints. The purpose of the second configuration is to provide a set of more challenging test cases for *GP-Eval* (FC-LBJ) where the neighborhood retrieval queries have potentially larger result sizes, thereby increasing the chance of encountering a hub and making it more difficult to prune candidate sets, which results in larger candidate set sizes. For the second dataset, we used the *"soc-Pokec"* social network dataset obtained from SNAP [5] to generate the authorization graph. The graph consisted of 1,632,803 nodes and 30,622,564 directed edges. We generated seven relationship identifiers and randomly assigned a relation identifier to each edge; however, we did not assign any attributes to either nodes or relationships, other than a unique *id*. For the first configuration, we generated 250 test cases of size 10, 11, or 13 nodes, with 0, 1, or 2 mutual exclusion constraints, and 1, 2, or 4 members of *Ret*. For the second configuration, we made a more difficult version of the 250 test cases by removing the relation identifiers and mutual exclusion constraints. We also set the timeout limit to 300 seconds instead of 60, because without the relation identifiers and additional constraints, the test cases take longer to complete.

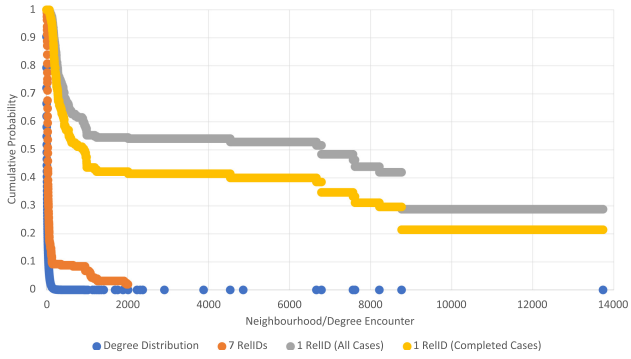
Results and Discussions. Figure 11 summarizes the results of the hub evaluation tests. Instead of providing the raw numbers, we normalized the results to achieve a more comparable view of said results. Each data point presents the probability of picking a node with either an in or out degree of x or higher, based on the source of the data point. We included the degree distribution of both datasets; however, since the relationships are directed, we treated the in and out degrees separately (i.e., each node had two degrees). All of the test cases completed within the time limit for the first configuration (7 RelIDs); however, that was not the case for the second configuration (1 RelID), hence the two data plots in each graph (All Cases and Completed Cases). For the *"soc-Slashdot0922"* dataset Configuration 2, 110 out of 250 test cases completed, and for the *"soc-Pokec"* dataset Configuration 2, 135 out of 250 test cases completed.

The first obvious observation here is that introducing multiple relation identifiers reduces the maximum degree encountered when evaluating graph pattern queries. Second, encountering a large neighborhood has a significantly steeper decline with multiple relation identifiers as compared to having only a single relation identifier. Note that this is different than having a lower maximum degree. The observation implies that having multiple relation identifiers produces a much more favorable situation for *GP-Eval* (FC-LBJ). Additionally, the observation implies that *GP-Eval* is not a suitable algorithm when the underlying authorization graph only consists of a single relation identifier.

The fact that *GP-Eval* was able to complete all test cases for Configuration 1 but not for Configuration 2, implies we cannot guarantee the performance of *GP-Eval* under such conditions. We believe the long running time for such graph patterns was due to performing expensive in-memory operations (e.g., $Cand(u) \cap N$)) or working with significantly large candidate sets leading to large search trees, or perhaps a combination of multiple factors. We leave the exploration of the performance bottleneck(s) of *GP-Eval* as future work.



a) soc-Slashdot0922 Results



b) soc-Pokec Results

Fig. 11. Hub encounter evaluation results. The "Degree Distribution" data points describe percent of nodes from the underlying dataset, with either in or out degree of at least x. The "7 RelID" data points describe the percent of neighborhood retrieval queries performed (for the test cases with 7 relation identifiers) where the results contained at least x nodes. The "1 RelID" data points describe the percent of neighborhood retrieval queries performed (for the test cases with a single relation identifier) where the results contained at least x nodes. We present the results of "1 RelID" test cases in two ways: (i) presenting the results of all 250 test cases (having partial results from instances that did not complete within 300-second time limit) and (ii) only presenting the results of instances that were able to complete the evaluation within the 300-second time limit.

6 RELATED WORKS

A number of ReBAC models have been proposed in the literature. Fong [21] proposed a ReBAC model that tracks a directed and edge-labeled graph as its protection state and only allows relationships between users. The connection between resources and users is maintained using a system-depended *owner* function. This model supports modal logic as its policy language, and with an extension to hybrid logic [10]. Reference [22] added a temporal dimension that requires users (and resources) to be related in a certain way in the past for access to be granted. Reference [38] extended the hybrid logic policy language to impose relationship constraints over people located in certain geographical neighborhoods.

Crampton and Sellwood [18] proposed *RPPM*, a ReBAC model that not only tracks relationships between users, but also resources. This allowed for a wider class of relationships than

References [10, 21]. RPPM uses *path conditions*, which are similar to regular expressions, as its policy language. Reference [18] also proposed *authorization principals,* which are a relationship-based analog of roles. When an authorization request is generated, the system computes the principal membership for the requestor and resource pair. The authorization decision is then based on principal membership and conflict-resolution policies. Crampton and Sellwood then proposed the Administrative RPPM (*ARPPM*) [19] model. ARPPM extended RPPM by being able to handle operational and administrative requests.

Rizvi et al. [36] combined authorization principals of *RPPM* [18] with hyrbid logic [10] policy language and demonstrated for the first time that ReBAC can be implemented in a production-scale medical records system. They chose OpenMRS [4] as their subject and maintained backwards compatibility with the system's original RBAC mechanism. They provided two authorization semantics and two algorithms for evaluating authorization requests. Reference [36] also included an empirical evaluation of computing authorization decisions under various profiles and reported that the fastest profiles took between 0.016 to 0.037 second per authorization check on average. An interesting insight of their work was that using the original datastore for OpenMRS (MySQL [2]) neighborhood retrieval queries took 0.002 second, and so they used Neo4j [3] for tracking relationships, which took 0.0001 second on average for neighborhood retrieval queries. Rizvi et al. then extended their model to interoperate with RBAC and formalized their model as *ReBAC2015* [34].

Pasarella and Lobo [30] also combined Reference [18] with Reference [10]. However, unlike Reference [36] they extended the hybrid logic of Reference [10] to express the path conditions of Reference [18] and dubbed the language Extended Hybrid Logic (EHL). Along with the extension, they presented a subset of Datalog with equality constraints for specifying ReBAC policies, which was then extended to allow negative authorization and handle temporal policies. Similar to our work, they demonstrate that ReBAC models are not restricted to their specific policy language by defining how to translate an EHL formula to a set of Datalog rules and vice versa. While discussing the performance of their proposed system, they mention how fixing one or two of the variables to specific nodes in the dataset significantly impacts the search space and thus the performance of executing a query. This phenomenon is also exploited in *GP-Eval*. Although discussed only briefly, Pasarella and Lobo's ReBAC datalog model is also able to express attribute requirements [30, Policy 4].

Attribute-Based Access Control (ABAC) is an access control paradigm that bases its authorization decisions on attributes [23], while ReBAC evaluates relationships for computing its authorization decisions. It has even been argued that ReBAC is an extension of ABAC that supports entity attributes. Ahmed et al. [7] presented a comparative analysis of ABAC and ReBAC. They presented a family of ReBAC models and a family of ABAC models and demonstrated how the ABAC models were either equally or more expressive than ReBAC models. According to their finding, AReBAC is less expressive than an ABAC model, which supports entity and non-entity attributes with structured values. Other extensions to ReBAC have also been proposed to support attributes. Cheng et al. proposed an attribute-aware ReBAC model [14] as an extension to user-to-user relationship-based access control (UURAC) [13]. The original UURAC model uses a regular expression-based policy language, and the extended model updated the language to incorporate attributes for making authorization decisions.

A common limitation between these models is that they depend on the traditional definition of authorization requests, (u, r, a) where user u requests to execute action a on resource r. As mentioned above, this approach requires a two-step process of query execution and authorization checking. For search queries that yield multiple results this can be costly, since authorization checking needs to be performed between the requestor and each member of the result set. AReBAC remedies this limitation by performing the query execution and authorization checking simultaneously.

Several database systems offer an access control mechanism that performs authorization checking with query executions. Ferrari provided an overview of access control data management systems for Discretionary (DAC), Mandatory (MAC), and RBAC models [20]. Authorization checking is performed during query execution based on the administrative state of the database, and thus, the results of a query execution are ones that match the search parameters and the requestor has access to. Following the example of traditional databases, Neo4j recently introduced RBAC features. Query rewriting is a common approach for providing access control features for databases. Rizvi et al. [32] rewrite queries by altering the target base relation to authorized views so the query result is composed to entities that the user is allowed to access. Colombo and Ferrari [15, 16] proposed Mem and ConfinedMem as access control models for MongoDB. Mem and ConfinedMem use query rewriting to introduce finer-grained constraints, such as privacy policy enforcement, context awareness, and content-based access control. Both of these models provide finer-grained access control than MongoDB's default RBAC. We agree with Colombo and Ferrari that NoSQL datastores operate on various languages and data models, which makes a general approach for finer-grained access control for NoSQL datastores a very ambitious goal. Thus, to tackle this challenge, we also focus on a single datastore. Oracle Virtual Private Database also employs query rewriting to enforce access control policies along with query evaluation [24]. Although AReBAC follows in the footsteps of these access control models, what sets AReBAC apart is that it focuses on a graph database model and enforces ReBAC policies that support attributes. To the best of our knowledge, this has not been conducted before.

7 CONCLUSION AND FUTURE WORKS

AReBAC is an attribute-supporting ReBAC model for controlling access in Neo4j, a graph database, and is motivated by three factors: (i) combine database queries with authorization checking to obtain acceptable performance for query executions coupled with authorization checking, (ii) connect the model with technologies that are used in industry, and (iii) promote accessibility through easy-to-use policy languages and providing the required tools.

There are still a few explorable areas in this work. First, *GP-Eval* can be optimized by exploiting techniques used in constraint-satisfaction problems such as variable ordering and decomposing the problem into subproblems. Since database communication is limited to the forward checking component of the algorithm, AReBAC can be extended to other database systems such as relational databases and other graph databases by creating new forward checking modules for the algorithm. When one or more extensions of AReBAC exist, a performance comparison would be a natural step. Nano-Cypher can also be extended to incorporate more features offered by Cypher, such as attribute comparison between vertices instead of comparison to strict values. For AReBAC to be usable, we also need to introduce capabilities for manipulating the database.

APPENDIX

A GENERATING GRAPH PATTERN

Based on the parameters provided, the graph pattern is extracted as follows: If a seed node is provided, then we use it as the starting point by adding it to the pool of available nodes (*nodesPool*), otherwise, the starting point is a random node from the database. During the first phase, we choose a node from the *nodesPool* and query the database for all of its neighbors. We then pick one of its neighbors, uniformly at random, and if it is not already a member of *nodesPool*, we add it to the *nodesPool*. We repeat this process until we have the required number of nodes ($|GP|$) in the pool.

In the second phase, we first randomly shuffle *nodesPool*, then iterate through each node in the set. For each node, v, we query the database for all of its relationships and iterate through each

relationship in the result. For each relationship, r, if the neighbor of v is also in *nodesPool*, we then add the relationship to the set *relsPool*. We limit only one relationship between any pair of nodes, regardless of direction and relation identifier, so if $ID_i(u, v) \in relsPool$ and $r = ID_j(v, u)$, we will not add r to *relsPool*. Hence, the reason for shuffling *nodesPool* at the beginning of this phase. Restricting the graph pattern to allow only one relationship between pairs of nodes reduces the constraints on the possible candidates for the corresponding vertices during evaluation, thereby allowing larger candidate sizes and larger search trees.

After the second phase, we check if we have the required connectivity between the nodes. We fixed the connectivity requirement to $|relsPool| \geq 1.5(|nodesPool| - 1)$ to avoid tree-structured graph patterns. If the completeness requirement is not satisfied, we fail and start all over again. Otherwise, we generate the corresponding graph pattern based on the nodes and relationships. At this stage, we also generate the vertex (edge) attribute requirements by randomly picking a node (edge) from *nodesPool* (*relsPool*) and querying the database for one of its attributes. We repeat this until we have obtained the desired number of attribute requirements. Once again, we avoid repeats. Next, we generate the mutual exclusion constraints. This is done by simply picking pairs of nodes $u, v \in nodesPool$ with $u \neq v$. Last, we generate the *resultSchema* by picking the number of nodes specified for *Ret*.

REFERENCES

[1] [n.d.]. Intro to Cypher. Retrieved from https://neo4j.com/developer/cypher-query-language/.

[2] [n.d.]. MySQL. Retrieved from http://www.mysql.com/.

[3] [n.d.]. Neo4J. Retrieved from http://neo4j.com/.

[4] [n.d.]. OpenMRS. Retrieved from http://openmrs.org/.

[5] Stanford Large Network Dataset Collection. 2014. Retrieved from http://snap.stanford.edu/data.

[6] Rafiul Ahad, James Davis, Stefan Gower, Peter Lyngbaek, Andra Marynowski, and Emmanuel Onuegbe. 1992. Supporting access control in an object-oriented database language. In *Proceedings of the Conferences on Advances in Database Technology (EDBT'92)*. Springer Berlin, 184–200.

[7] Tahmina Ahmed, Ravi Sandhu, and Jaehong Park. 2017. Classifying and comparing attribute-based and relationship-based access control. In *Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy (CODASPY'17)*. ACM, New York, NY, 59–70. DOI: https://doi.org/10.1145/3029806.3029828

[8] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Comput. Surv.* 40, 1, Article 1 (Feb. 2008), 39 pages.

[9] Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *Science* 286, 5439 (1999), 509–512. DOI: https://doi.org/10.1126/science.286.5439.509

[10] Glenn Bruns, Philip W. L. Fong, Ida Siahaan, and Michael Huth. 2012. Relationship-based access control: Its expression and enforcement through hybrid logic. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy (CODASPY'12)*.

[11] Xinguang Chen and Peter van Beek. 2001. Conflict-directed backjumping revisited. *J. Artif. Int. Res.* 14, 1 (Mar. 2001), 53–81. Retrieved from http://dl.acm.org/citation.cfm?id=1622394.1622397.

[12] Yuan Cheng, Jaehong Park, and Ravi Sandhu. 2012. Relationship-based access control for online social networks: Beyond user-to-user relationships. In *Proceedings of the 4th IEEE International Conference on Information Privacy, Security, Risk and Trust (PASSAT'12)*.

[13] Yuan Cheng, Jaehong Park, and Ravi Sandhu. 2012. A user-to-user relationship-based access control model for online social networks. In *Proceedings of the 26th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec'12) (LNCS)*, Vol. 7371.

[14] Yuan Cheng, Jaehong Park, and Ravi Sandhu. 2014. Attribute-aware relationship-based access control for online social networks. In *Proceedings of the 28th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy XXVIII—Volume 8566 (DBSec'14)*. Springer-Verlag New York, Inc., 292–306. DOI: https://doi.org/10.1007/978-3-662-43936-4_19

[15] P. Colombo and E. Ferrari. 2016. Towards virtual private NoSQL datastores. In *Proceedings of the IEEE 32nd International Conference on Data Engineering (ICDE'16)*. 193–204. DOI: https://doi.org/10.1109/ICDE.2016.7498240

[16] P. Colombo and E. Ferrari. 2017. Enhancing MongoDB with purpose-based access control. *IEEE Trans. Depend. Secure Comput.* 14, 6 (Nov. 2017), 591–604. DOI: https://doi.org/10.1109/TDSC.2015.2497680

[17] Jason Crampton and Gregory Gutin. 2013. Constraint expressions and workflow satisfiability. In *Proceedings of the 18th ACM Symposium on Access Control Models and Technologies (SACMAT'13)*. ACM, New York, NY, 73–84. DOI: https://doi.org/10.1145/2462410.2462419

[18] Jason Crampton and James Sellwood. 2014. Path conditions and principal matching: A new approach to access control. In *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies (SACMAT'14)*. ACM, New York, NY, 187–198.

[19] Jason Crampton and James Sellwood. 2016. ARPPM: Administration in the RPPM model. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY'16)*. ACM, New York, NY, 219–230. DOI: https://doi.org/10.1145/2857705.2857711

[20] Elena Ferrari. 2010. *Access Control in Data Management Systems*. Morgan and Claypool Publishers.

[21] Philip W. L. Fong. 2011. Relationship-based access control: Protection model and policy language. In *Proceedings of the 1st ACM Conference on Data and Application Security and Privacy (CODASPY'11)*. ACM, New York, NY, 191–202.

[22] Philip W. L. Fong, Pooya Mehregan, and Ram Krishnan. 2013. Relational abstraction in community-based secure collaboration. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*. 585–598.

[23] Vincent C. Hu, David Ferraiolo, Rick Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. 2014. Guide to attribute based access control (ABAC) definition and considerations. *NIST Spec. Pub.* (Jan. 2014).

[24] Patricia Huey. 2014. *Oracle Database Security Guide 11g Release 1 (1.11)*. Oracle Corp.

[25] Matthew O. Jackson and Brian W. Rogers. 2007. Meeting strangers and friends of friends: How random are social networks? *Amer. Econ. Rev.* 97, 3 (June 2007), 890–915. DOI: https://doi.org/10.1257/aer.97.3.890

[26] Daniel Karapetyan, Andrew J. Parkes, Gregory Gutin, and Andrei Gagarin. 2016. Pattern-based approach to the workflow satisfiability problem with user-independent constraints. *CoRR* abs/1604.05636 (2016).

[27] Jure Leskovec, Lars Backstrom, Ravi Kumar, and Andrew Tomkins. 2008. Microscopic evolution of social networks. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'08)*. ACM, New York, NY, 462–470. DOI: https://doi.org/10.1145/1401890.1401948

[28] M. Lichman. 2013. UCI Machine Learning Repository. Retrieved from http://archive.ics.uci.edu/ml.

[29] Jakob Nielsen. 2009. Powers of 10: Time Scales in User Experience. Retrieved from https://www.nngroup.com/articles/powers-of-10-time-scales-in-ux/.

[30] Edelmira Pasarella and Jorge Lobo. 2017. A datalog framework for modeling relationship-based access control policies. In *Proceedings of the 22nd ACM Symposium on Access Control Models and Technologies (SACMAT'17 Abstracts)*. ACM, New York, NY, 91–102. DOI: https://doi.org/10.1145/3078861.3078871

[31] Patrick Prosser. 1993. Hybrid algorithms for the constraint satisfaction problem. *Comput. Intell.* 9, 3 (1993), 268–299.

[32] Shariq Rizvi, Alberto Mendelzon, S. Sudarshan, and Prasan Roy. 2004. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*. ACM, New York, NY, 551–562. DOI: https://doi.org/10.1145/1007568.1007631

[33] Syed Zain R. Rizvi. 2018. Attribute-Supporting ReBAC Model. Retrieved from http://pages.cpsc.ucalgary.ca/ szrrizvi/projectAReBAC/.

[34] Syed Zain R. Rizvi and Philip W. L. Fong. 2016. Interoperability of relationship- and role-based access control. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY'16)*. ACM, New York, NY, 231–242.

[35] Syed Zain R. Rizvi and Philip W. L. Fong. 2018. Efficient authorization of graph database queries in an attribute-supporting ReBAC model. In *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy (CODASPY'18)*. ACM, New York, NY, 204–211. DOI: https://doi.org/10.1145/3176258.3176331

[36] Syed Zain R. Rizvi, Philip W. L. Fong, Jason Crampton, and James Sellwood. 2015. Relationship-based access control for an open-source medical records system. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies (SACMAT'15)*. ACM, New York, NY, 113–124.

[37] Francesca Rossi, Peter Van Beek, and Toby Walsh. 2007. *Handbook of Constraint Programming*. Elsevier.

[38] Ebrahim Tarameshloo and Philip W. L. Fong. 2014. Access control models for geo-social computing systems. In *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies (SACMAT'14)*. ACM, New York, NY, 115–126. DOI: https://doi.org/10.1145/2613087.2613098

[39] Peter van Beek. 2006. *Backtracking search algorithms*. In *Handbook of Constraint Programming*. Elsevier, 85–134.

[40] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. 2010. A comparison of a graph database and a relational database: A data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference (ACM SE'10)*. ACM, New York, NY, Article 42, 6 pages. DOI: https://doi.org/10.1145/1900008.1900067

Received August 2019; revised March 2020; accepted May 2020