UNLOCKING THE POWER OF QUERY LANGUAGE

Cristina ȚĂRNĂ[1], Tudor SCLIFOS[1] *,
Dan HARITON[1], Leonid CAMINSCHI[1],
Andrei CEBAN[1]

[1]Software Engineering and Automatics, Technical University of Moldova, Group FAF-211, Faculty of Computers, Informatics and Microelectronics, Chișinău, Republic of Moldova

**Corresponding author: Tudor Sclifos, tudor.sclifos@isa.utm.md*

Abstract. *This article explores the differences between Role-Based Access Control (RBAC), and Query language. The article delves into the technical aspects of each language, including its strengths and weaknesses. Additionally, the article highlights the developed Query language and its strong sides. The access is granted based on the role the user has and the accorded time. The base roles are select, delete, insert and update. By the end of this article, a comprehensive understanding of the access control policies will be established and how they can be combined with a Query language.*

Keywords*: authorization, privileges, security, enforcement, control*

Introduction

Access control policies are a cornerstone of information security, providing a means to restrict access to resources and data to only those users who are authorized to access them. Access control policies define the rules for granting and revoking access to resources and specify the conditions under which access is granted or denied. However, defining and enforcing access control policies can be a complex and challenging task, particularly in large and dynamic environments where access requirements are constantly changing.

Access control policy languages provide a formal framework for expressing access control policies in a machine-readable format. These languages enable organizations to define access control policies in a structured and consistent way, ensuring that policies are applied uniformly across the organization. Policy languages also enable automated enforcement and compliance monitoring, reducing the burden on security teams and improving the overall effectiveness of access control.

In this article, the key concepts and best practices of access control policy languages will explore. The various types of policy languages, including attribute-based access control (ABAC), role-based access control (RBAC), and mandatory access control (MAC) will be examined. The challenges of implementing access control policies and emerging trends and technologies that are shaping the future of access control policy languages will be explored. By the end of this article, a solid understanding of access control policy languages and the role they play in securing modern information systems will be established.

RBAC, frequently known as role-based security, is a technique for restricting system access to accredited users in computer system security. It is a strategy for enforcing either MAC or optional access control (DAC) [1].

A policy-neutral access-control method built around roles and privileges is known as role-based access control. It is simple to carry out user assignments thanks to RBAC's components, similar as role warrants, user roles, and role-role connections. To simplify security administration in big companies with thousands of users and hundreds of authorizations, RBAC can be a great tool. Even though RBAC is unlike MAC and DAC access control frameworks, it may effortlessly enforce these constraints.

1. Domain description and analysis

A domain-specific language (DSL) is a software language that's domain-specific. A software language is a language that is written and read by humans, but also automatically processed by computers. This section will touch upon the key aspects of a software language.

A domain is a focused area of knowledge. It comes with a group of domain experts owning the domain: people possessing, shaping, extending that area's knowledge, and sharing it with business stakeholders and other interested parties. Some could even claim that a domain frequently exists in the first place as such a group of experts in the field, with that group defining it ad hoc.
This pre-DSL is already domain-specific but needs some effort to become a software language. Looking at the key aspects of a software language, we can decide how to podcast the right DSL from the pre-DSL. With that information, starting to implement DSL as a software language is easy. That implementation takes the form of tools like the Domain IDE and the code generator. The Domain IDE allows experts in the field to write and read DSL content with DSL editors.

Adopting a DSL-based approach has many positive effects on software development. Here are some potential motivations for using a DSL:

- Abstraction: a DSL allows developers to abstract complex programming concepts and focus on a specific domain. This can make the code more readable and easier to maintain.
- Productivity: with a DSL, developers can create software quickly and efficiently. DSLs allow developers to express ideas in a concise and expressive way, which can speed up development time.
- Improved Communication: DSLs can improve communication between developers and non-technical stakeholders. With a DSL, business analysts or domain experts can more easily express their needs and requirements, without requiring a deep technical understanding.
- Safety: a DSL can help to catch errors early before they cause problems. By providing a language that is designed specifically for a domain, developers can write code that is less error prone.
- Customization: a DSL can be customized to a specific problem or domain, which can make it more powerful and easier to use.
- Domain-Specific Optimizations: a DSL can be perfected specifically for the domain it serves. This can lead to more efficient code and better performance.

The DSL chosen for this article is a Query language combined with RBAC.
A query language is a programming language designed to retrieve and manipulate data stored in a database. It is a specialized language that allows users to ask specific questions, or queries, about data in a structured way.

There are various query languages used for diverse types of databases, including SQL (Structured Query Language), which is used for relational databases, and NoSQL query languages like MongoDB query language and Cassandra Query Language, which are used for non-relational databases.

Query languages typically consist of statements or commands that are used to interact with a database, such as SELECT, INSERT, UPDATE, and DELETE. These statements allow users to retrieve specific data from a database, add new data, update existing data, and delete data.

Query languages are essential for working with databases, as they allow users to access and manipulate enormous amounts of data in a structured and efficient way.

2. Grammar

For greater comprehension, further the grammar for the Query language is represented according to an amazingly simple and textual program. The language allows the user to access, modify, insert and delete elements from a non-relational database. Next, each feature of grammar is shown in detail.

The DSL design includes a number of stages. Firstly, the definition of the programming language grammar is $G = (Vn, Vt, P, S)$:

Vn – a finite set of non-terminal symbols;

Vt - a finite set of terminal symbols.

P – a finite set of the production of rules;

S - the start symbol;

In Table 1 are meta-notations for specifying the grammar.

Table 1

Meta notation

Notation (symbol)	Meaning
<foo>	foo is a nonterminal.
"foo" | 'f'	means that foo and f are terminal i.e., a token or a part of a token.
x?	means zero or one occurrence of x, i.e., x is optional
x*	means zero or more occurrences of x
x+	one or more occurrences of x
{}	used for grouping
|	separates alternatives

```
<query> -> <selectquery> | <generatekeyquery> | <createuserquery> |
<loginquery> | <createquery> | <insertquery> | <updatequery> |
<deletequery>
| <help>
<selectquery> -> <select_clause> <from_clause> <where_clause>?
   <order_by_clause>?
<select_clause> -> "SELECT" <field_name> {"," <field_name>}* | "*"
<from_clause> -> "FROM" <collection_name>
<where_clause> -> "WHERE" <condition>
<order_by_clause> -> "ORDER BY" <field_name> ("ASC" | "DESC")?
<generatekeyquery> -> "generate_key" <privilege> <privilege> <privilege>
   <privilege> <time>?
<privilege> -> <boolean>
<time> -> <int>
<createuserquery> -> "create_user" <username> <password> <mail> <token>
<mail> -> <string> "@" <string> "." <string>
<token> -> <string>
<loginquery> -> "log_in" <username> <password>
<createquery> -> <create_clause> <codeblock>
<create_clause> -> "CREATE TABLE" <collection_name>
<codeblock> -> "(" <table_fields> ")"
<table_fields> -> <collection_collumn> {"," <collection_collumn>}*
<collection_collumn> -> <type> <field_name>
<insertquery> -> <insert_clause> <values_clause><insert_clause> -> "INSERT
INTO" <collection_name>
<values_clause> -> "VALUES" "(" <value> {"," <value>*} ")"
<updatequery> -> <update_clause> <where_clause> <set_clause>
<update_clause> -> "UPDATE" <collection_name>
<set_clause> -> "SET" <field_name> "=" <variable>
<deletequery> -> <delete_clause> <where_clause>?
<delete_clause> -> "DELETE FROM" <collection_name>
<help> -> "HELP"
<username> -> <string>
<password> -> <string>
<condition> -> <comparison> | <logical_op> | "(" <condition> ")" |
<comparison> -> <variable> <comparison_op> <variable>
<variable> -> <field_name> | <value>
```

```
<logical_op> -> <condition> <logical_op> <condition>
<comparison_op> -> "==" | "!=" | ">" | ">=" | "<" | "<="
<field_name> -> <string>
<collection_name> -> <string>
<value> -> <string> | <number> | <NULL>
<NULL> -> '0' | ''
<type> -> <string> | <int> | <float> | <char> | <digit> | <boolean>
<string> -> '"' <char>* '"'
<int> -> <digit>+
<float> -> <digit>+ "." <digit>+
<char> -> 'a' | 'b' | … | 'z' | 'A' | 'B' | … | 'Z'
<digit> -> '0' | '1' | ... | '9'
<boolean> -> 'True' | '1' | 'False' | '0'
```

Parse Tree

Statement example:

SELECT * FROM foo WHERE id=prev_id ORDER BY id

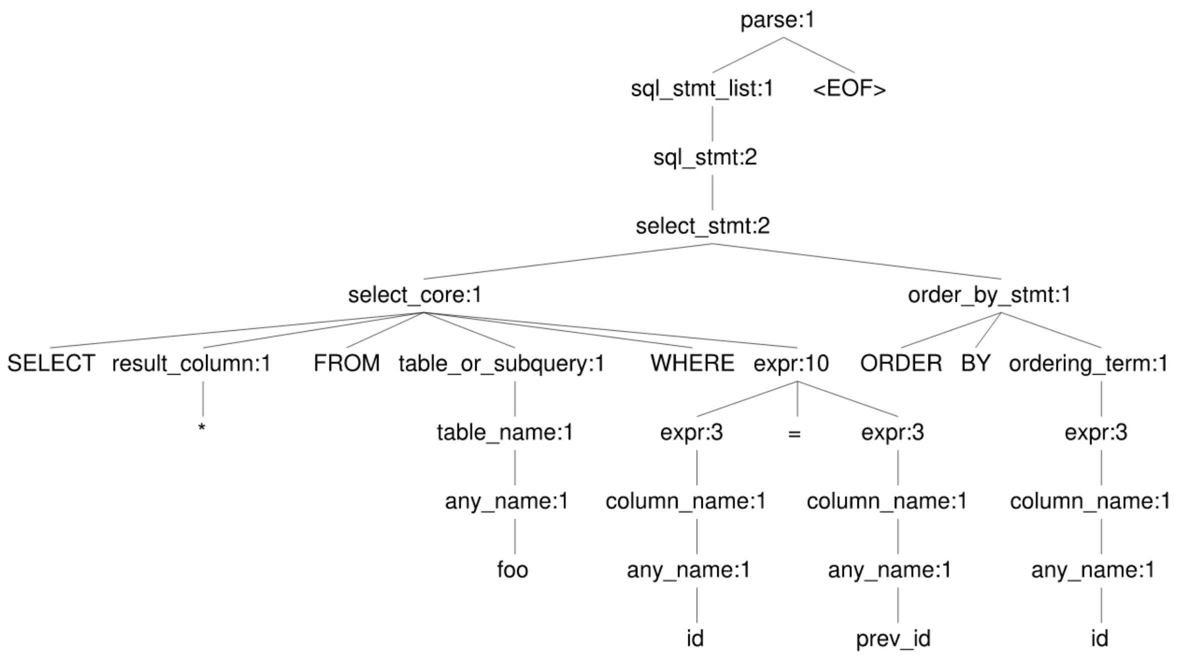

Figure 1. Parse tree for the statement example

Conclusions

To sum up, the query language presented is easy to understand and implement, as it is simpler than SQL. Overall, each language has its strengths and weaknesses, and the choice of which to use will depend on the specific needs and requirements of the organization. It is essential to carefully consider the advantages and limitations of each language before implementing an access control system.

References

1. FERRAIOLO, D.F. & KUHN, D.R. Creativity and Imagination: Role-Based Access. [online]. [accessed 5.03.2023]. Available at: https://csrc.nist.gov/publications/detail/conference-paper/1992/10/13/role-based-access-controls
2. MEINTE, B. Domain-Specific Languages Made Easy. [e-book] Available at: https://www.manning.com/books/business-friendly-dsls