Journal of Computer Science

Original Research Paper

Adaptivity in Role-Based Access Control During Stochastic Situations: A Comprehensive Study between Graph and Relational Databases

Kriti Srivastava, Dhruv Jain, Aarya Kamdar, Anuradha Yeole, Devam Shah and Sowmya Dadheech

Department of Computer Science and Engineering (Data Science), Dwarkadas J. Sanghvi College of Engineering, Mumbai, India

Article history
Received: 07-09-2024
Revised: 12-09-2024
Accepted: 07-10-2024

Corresponding Author:
Aarya Kamdar
Department of Computer Science and Engineering (Data Science), Dwarkadas J. Sanghvi College of Engineering, Mumbai, India
Email: aaryawork1@gmail.com

Abstract: Role-based control is straightforward to implement in static systems where there are specific policies for role and resource mapping. The main challenge is faced in dynamic and unpredictable systems with erratic workflows. Conventional methods prove to be inadequate in dynamic environments. Over the Years methods suggested include probabilistic, machine learning, ontology, and decision tree models to improve the adaptability. However, they fail to build a bridge between operational methods and flexible approaches. Role-based control systems are based on previously defined rules and policies. A truly flexible system should run without human interference, autonomously accessing the user's request and granting the requirements based on it is genuineness. This research introduces the need for a based control methodology, using a project management case study using Neo4j. It generates the responses promptly based on the authenticity of the user We have computed the time required to process the results in in SQL as well as Neo4j. A role-based control system is built to improve coordination among the departments present in the real-life corporate world and transfer data based on authenticated data requests. Graph databases outperform relational databases by nearly an average of 8 milliseconds on an average across the queries run. This framework demonstrates better flexibility, system adaptability, and precise computational efficiency across various scenarios.

Keywords: Graph Databases, Neo4j, NoSql, Role Based Access, RBAC, MYSQL, Relational Databases

Introduction

In the digital age, we are witnessing an unprecedented surge in data volume and complexity. This growth presents both opportunities and challenges, particularly in the realm of data management and security. Graph databases have emerged as a powerful solution for handling intricate relational data. Unlike a traditional database, holding data in a table structure, graph databases use nodes, edges, and properties to represent information. This would efficiently map out complicated relationships and produce good results for applications such as fraud detection and recommendation systems. That is, a growing necessity for increased strength and adaptiveness of database management solutions in order to deal with the rising complexity of digital ecosystems; among them, graph databases have been fitted into that gap. Robust data security has become very important with the growth of data management. An RBAC system is one of the most reliable ways of providing this. Basically, RBAC is a well-practiced model of security principles applied in the storage of data, especially in the corporate world. Such a system gives permissions for roles rather than users; this makes the means of handling access control at a large and complex organization easier and more efficient. This technique is particularly important in industries like banking, healthcare, or business information technology, where tight security is absolutely necessary due to very sensitive information. Traditionally, RBAC systems have been implemented using relational databases with access rights and role assignments stored in tables. While these systems are well-established and have served their purpose effectively, they often struggle to meet the scalability and complexity demands of modern RBAC implementations. The higher the number of roles, permissions, and user-to-role assignments, the higher the complexity and resource intensiveness required in queries by access control management. Performance bottlenecks can occur for static schema relational databases and the partial support of high-dimension relationship cases might endanger system efficiency. This paper is therefore aimed at exploring the potential of graph databases with respect to developing an RBAC system through performance benchmarking against traditionally favored relational database reasons that assess the manner by which Neo4j does the processing of complex queries and handling the intricate network of user-permission-role relationships using intrinsic advantages of graph databases. Such insights that the work is likely to provide reflect the potential benefits of using graph databases for RBAC implementations and open up real new ways toward better methods of data security and management. Through this comparative analysis, we hope to contribute to the ongoing dialogue about the evolution of database technologies and their role in addressing the complex challenges of modern data ecosystems. By examining the strengths and limitations of both approaches, we aim to offer valuable insights to organizations grappling with the dual challenges of data complexity and security in an increasingly interconnected world. In this review of the literature, an overview is given of a few studies that compare relational databases with graph databases, particularly focusing on Neo4j and SQL databases. Batra and Tyagi (2012) performs a performance comparison between Neo4j and MySQL, outlining the flexibility and higher performance of graph databases in answering complex interaction questions. It shows how relational databases become inefficient when one faces a scenario of rapid dynamic schema growth, along with a large number of joint operations. Neo4j will fit better in applications where advanced management of relationships is a necessity since it has a schema-less architecture and improved traversal algorithms. Sharma *et al*. (2018) explore the performance of PostgreSQL, MongoDB, and Neo4j in managing geotagged data within GIS applications. It involves various databases with query response times and handling large volumes of data. It was found that the query response time was fastest with MongoDB and next with PostgreSQL. Though Neo4j had a slow speed in queries compared to the first two, Neo4j handled complex interrelationships. The conclusion of the study shows that the graph style of Neo4j benefits applications, which are highly reliant on data association. Sholichah *et al*. (2020) show results for a performance evaluation between Neo4j and MySQL databases by applying a wide range of benchmark queries, revealing differences between the two databases due to different data modeling approaches. If MySQL is better for structured data with predefined schemas, Neo4j gives better results in cases where schema changes and updates are much more frequent. One can notice that query response times in Neo4j remain rather flat across data volume. Khan *et al*. (2017) compare RBAC implementations across NoSQL databases, focusing on Neo4j, MongoDB, Cassandra, and Redis. The study highlights Neo4j's schema-less structure and built-in roles, which facilitate managing complex relationships. Medhi and Baruah (2017) presented a development comparison between a relational and graph database performance on a simple Cricket application reaching results were obtained that favored the Neo4J system. However, their tests were conducted on a dataset that had 400 objects and 3 queries were implemented. Jain *et al*. (2023) compare Neo4j (graph database) and MySQL (relational database) performance using the Career Village dataset. It evaluates execution times for selection, aggregation, recursion, and pattern-matching queries for SQL and Cypher. The results show Neo4j outperforming MySQL by up to. However, the study acknowledges that database performance depends on specific use cases. Gupta and Aggarwal (2014) compare Neo4j and MySQL, highlighting Neo4j's efficiency in managing evolving schemas and complex queries. Bechberger and Perryman (2020) focus on data query performance based on selection, aggregation, pattern matching, and recursion between Neo4j and MYSQL. Kotiranta *et al*. (2022) present a comparison of Neo4j, MySQL, and Maria DB; Neo4j is the best for simple queries, and Maria DB outperforms in complex queries. Relational databases have improved their indexing and optimizations and are competent for complex tasks. The results of Neo4j and MySQL compared for graph data in the Vicknair *et al*. provenance system are presented in Vicknair *et al*. (2010). Neo4j was faster when there were graph traversals but slower on numerical queries due to Lucene indexing. MySQL did very well for numerical data and smaller sets. In the study Fan (2012); Nguyen and Kim (2017); Do *et al*. (2022); Huang *et al*. (1996) compare graph databases, notably Neo4j, with traditional relational databases like MySQL. Comparisons are focused on different query features: Recursion, partial matching, grouping, and path searches.

RBAC features are less developed than MySQL's robust security controls, making it less suitable for applications requiring stringent data security. Despite Neo4j's strengths in handling frequent schema updates, its security mechanisms remain underdeveloped compared to traditional relational databases. Research Gap One of the challenges is to find the right balance between the security of a system and its adaptability. While the framework presented above tries to offer flexible access control, the ideal balance still has to be found. Overly inflexible systems can impede their ability to respond quickly and effectively. However, excessive flexibility may jeopardize security. This is a delicate balance that needs to be carefully considered. Certain frameworks are intended to manage emergency scenarios requiring dynamic access granting. There might be shortcomings in the system's ability to accurately recognize true emergencies and stop malicious actors from abusing it by posing as authorized requesters. In an evolving emergency scenario, the adeptness of time as a factor in obtaining the results should also be of primary importance. Incorporating time as a factor allows the system to dynamically adjust permissions based on the current state and elapsed time, enhancing the system's responsiveness and effectiveness. Materials and Methods Data Collection Taking into account earlier research on the subject, the proposed system outperforms the traditional methods used for role-based. The approach makes use of graph databases, Neo4j to represent the role-based access. A detailed and highly structured tracheal dataset is designed to present 10,000 users, 500 hierarchical roles comprising 5000 allocated permission roles user-role assignments, role inheritance, and constraints like mutually exclusive roles and role prerequisites. To fully comprehend the interconnected and complex structure of the dataset, it is essential to illustrate the relationships between users, roles, and permissions:

1. Has role: Connects a User node to a Role node, representing the roles assigned to a user
2. Inherits from: Connects a Role node to another Role node (its parent), showing that one role inherits permissions from another
3. Requires: Connects a role node to another role node (its prerequisite), indicating that a role requires another role before it can be assigned
4. Visualizing this complicated information contributes to a better understanding of the complex hierarchies and restrictions inside this RBAC system, offering clarity on structural dynamics
5. Users: visualized as nodes
6. Roles: Visualized as nodes with hierarchical relationships with other roles
7. Permissions: Connected to roles, inherited through their hierarchy
8. Constraints: Outline special constraints like mutually exclusive roles or prerequisite roles

Materials and Methods

Graph Databases

This section describes the use of Role-Based Access Control when it comes to a graph database. Hence, it indicates Some of the main phases related to the application of the RBAC model, focusing on constraints and the formation and attachment of the nodes. The rest, together with the data loading, turn out to be equally important in ensuring proper recording and execution of a graph database with robust Access Control across this system. Figure (1) presents the system architecture.

Data Load

The initial step in this approach is to load the data. Neo4j imports all relevant data for the RBAC model from a structured JSON file. This data includes all the information about the users, roles, permissions, and restrictions that establish the RBAC system. The data import process utilizes the apoc.load.json method from the APOC library. This stage lays the foundation for developing a reliable and accurate RBAC model within the system.

Figure (2) explains the Principles of role-based access control model. Node and relationship creation User creation in this stage, User nodes are created in the graph database, where each User denotes a person who requires access control. Each user is given a distinct to ensure that every subject in the RBAC model is taken into consideration. Since they serve as the foundation for subsequently granting roles and permissions, these User nodes are built initially. To set up the basis for role assignments, the array of users from the 'users.json' file must be unwound. For each user, a corresponding User node must be created.

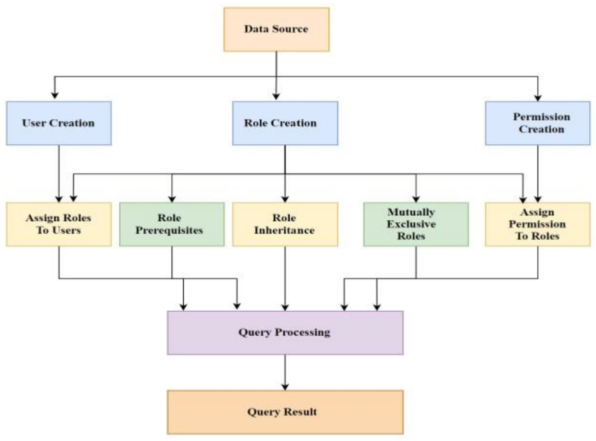


Fig. 1: System architecture of Neo4j

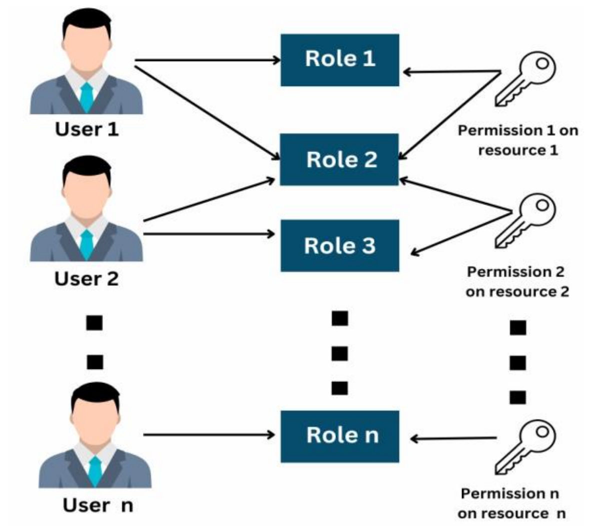

User 1
User 2
User n
Role 1
Role 2
Role 3
Role n
Permission 1 on resource 1
Permission 2 on resource 2
Permission n on resource n

Fig. 2: Principles of role-based access control model

Role Creation Role nodes are essential to the RBAC model because they allow a flexible, hierarchical access control structure by encapsulating permissions and allowing them to inherit permissions from other roles. Based on the role information from the JSON file, each

Role node is created. After that, _ permission relationships and, when applicable, inherits from relationships are used to link these nodes to Permission nodes in order to capture role inheritance. This step is essential in order to define the actions that fall under each position and to ensure that roles can be managed and maintained efficiently.

```
MATCH (u:User)-[:HAS_ROLE]->(r:Role)
RETURN u.id AS User, r.id AS Role;
```

1. Query for retrieving user-role assignments

Permission creation roles' permitted actions and resources are delineated by their permissions. The Permission nodes are created by unwinding the list of permissions linked to each role in the JSON file. Next, the nodes corresponding to the respective roles are connected, creating associations with permission. Permissions may be moved or changed without affecting the RBAC model, which makes the system more modular and scalable. This is achieved by decoupling permissions from roles within the graph structure.

```
MATCH (r:Role)-[:HAS_PERMISSION]->(p:
Permission)
RETURN r.id AS Role, p.id AS Permission;
```

2. Query for retrieving role-permission assignments

Role assignment and constraint definition role assignment Once the nodes are created, roles are assigned to users by establishing Hasrole relationships between user and role nodes. This step is crucial as it defines each user's access rights within the system. The role assignments are pulled from the JSON file and the corresponding relationships are created in the graph database to reflect these assignments. By setting up these relationships, the system ensures that users have the appropriate permissions according to their designated roles.

```
MATCH        (r:Role)-[:INHERITS_FROM*]->(parent:
    Role)
RETURN r.id AS Role, parent.id AS
    ParentRole;
```

3. Query for retrieving role inheritance

Role Prerequisites Some roles require users to hold specific prerequisite roles before they can be assigned. These dependencies are managed by creating required relationships between the relevant roles. This step enforces these prerequisites, ensuring that users meet all necessary criteria before gaining access to certain roles. This feature is essential for maintaining the integrity of the access control model, preventing users from bypassing critical role qualifications.

```
MATCH (r:Role)-[:REQUIRES]->(
    erequisiteRole:Role)
RETURN r.id AS Role, prerequisiteRole.id
AS PrerequisiteRole;
```

4. Query for retrieving role prerequisites

Mutually exclusive roles in certain scenarios, roles may be mutually exclusive, meaning a user cannot hold both roles at the same time. The system manages these constraints by marking specific roles as mutually exclusive, ensuring that conflicting roles are not assigned to the same user. This is critical for preventing security breaches or unintended access within the RBAC model. These constraints are enforced during the role assignment process, helping to maintain a consistent and secure access control environment.

```
MATCH (r: Role)
WHERE r.mutually_exclusive = true
RETURN r.id AS MutuallyExclusiveRole;
```

5. Query for retrieving mutually exclusive roles

Query Execution and Analysis

Retrieving users with specific permissions a crucial query in the RBAC system is identifying users who have been granted a specific permission, whether directly or through inherited role relationships. This query is vital for verifying that users possess the correct permissions and for auditing access within the system. By querying the graph, the system can efficiently determine which users have access to particular actions or resources, ensuring that access control policies are properly enforced.

```
MATCH (u:User)-[:HAS_ROLE]->(r:Role)-[:
HAS_PERMISSION*]->(p:Permission {id: 1})
RETURN u.id AS user_id;
```

6. Query for Retrieving Users with Specific Permissions

Listing roles inherited from a specific role an important query involves listing all roles that inherit permissions from a particular parent role. This query is essential for understanding the role hierarchy within the RBAC model. It enables system administrators to visualize how permissions are propagated through inherited roles, ensuring that the hierarchical structure is correctly maintained and that all inherited permissions are properly accounted for.

```
MATCH (r:Role {id: 1})
CALL apoc.path.subgraphNodes(r, { relationshipFilter:
'INHERITS_FROM', labelFilter: '+Role'
})
YIELD node
RETURN node.id AS role_id;
```

The query for listing roles inherited from a specific role and finding ancestor roles of a specific role this query identifies all ancestor roles of a given role, offering insight into the role's lineage and the inheritance path of permissions. This information is critical for debugging and verifying that role inheritance functions as intended. By examining the ancestor roles, the system can confirm that permissions are inherited correctly, ensuring consistency in the access control model.

```
MATCH (r:Role {id: 3})-[:INHERITS_FROM
    *]->(ancestor:Role)
RETURN ancestor.id AS ancestor_role_id;
```

7. Query for finding ancestor roles of a specific role

Detecting Users Missing Required Prerequisites This query identifies users who have been assigned roles without meeting the necessary prerequisite conditions. It is crucial to ensure that all role assignments adhere to the RBAC model's rules, especially those related to prerequisites. By running this query, the system helps maintain the integrity of role assignments, preventing users from acquiring roles for which they are not qualified and safeguarding the system against potential security risks.

Finding Users with Specific Permissions via Role Inheritance The system can also identify users who have acquired specific permissions through role inheritance. This query is crucial for auditing purposes, ensuring that inherited permissions are correctly assigned. It confirms that users have the appropriate access rights, even when those rights are obtained indirectly through inherited roles, thereby upholding the integrity of the access control model.

```
MATCH (p:Permission {id: 4533})<-[:
      HAS_PERMISSION]-(r:Role)<-[:HAS_ROLE
      *]-(u:User)
RETURN u.id AS user_id;
```

8. Query for finding users with specific permissions via role inheritance

Relational Databases

During the process of setting up data management, a fully functional instance of the MySQL server is needed. Specifically, five different CSV files exist within the dataset:

- Permissions.csv
- Users.csv
- User roles.csv
- Permissions csv.csv
- Constraints.csv

Among these, each file is sequentially imported into the MySQL Server. The server then converts them into different database tables. It does more than just an importation; it also provides detailed schema definitions for each of the created tables, such as defining data types, column characteristics, and relations among fields across tables.

Constraint Implementation

The import process is enhanced by the implementation of various constraints, such as:

- Primary keys
- Foreign keys
- Unique constraints

These constraints are crucial in the preservation of data integrity and consistency. They are well applied in ensuring that the database conforms to the rules for referential integrity, averting anomalies, and ensuring relationships between tables properly mirror the base data model.

Database Visualization

An entity-relationship diagram in Fig. (3) incorporates and represents the complex interrelations of the database. In essence, the ER diagram is a full-form graphical representation that maps out what each table represents and the relationships between them.

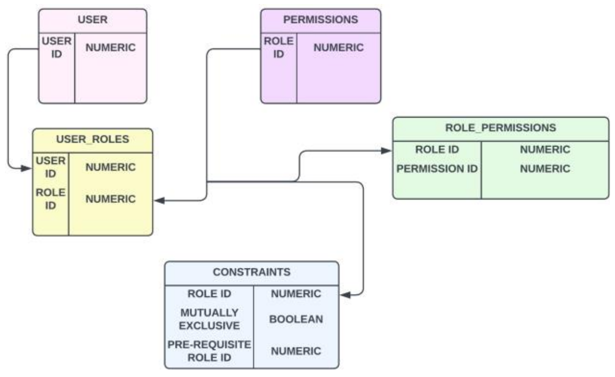


Fig. 3: ER-Diagram

The elements are interrelated in an ER diagram schematic representation, through which one obtains:

- A well-organized and clear description of the architecture of the database
- Enable deep understanding of the data model
- Insight into the logical flow of information within the system

This visualization is critical for both the design phase and ongoing database management, as it allows for:

- Identification of potential issues
- Optimization of database performance

Query execution and analysis retrieval of users with Specific Permissions One of the main issues it addresses is how to determine the users who have been assigned a particular privilege. In this regard, it deals with the direct privileges of the user but not those inherited through the associated roles. It would be very critical to confirm that a user is rightfully allocated permission; this shall be in addition to being very beneficial when carrying out user system access audits. It clearly defines who is to have what and how much of the specific resources are to be made available for each of the nominated users based on the access control policy implemented when this query is executed in MySQL.

```
SELECT DISTINCT ur.user_id
FROM UserRoles ur
JOIN RolePermissions rp ON ur.role_id = rp .role_id
JOIN Roles r ON ur.role_id = r.id
LEFT JOIN Roles parent ON r.parent_role_id
          = parent.id
LEFT JOIN RolePermissions prp ON parent.id
          = prp.role_id
WHERE rp.permission_id = ’1’ OR prp.
          permission_id = ’1’;
```

9. Query for retrieving users with specific permissions

Listing Roles Inherited from a Specific Role A useful query should include all the roles that inherit rights from a specific parent role. This query is very important in understanding the hierarchy of roles in the RBAC. It allows the system administrator to understand how permissions are propagated across inherited roles, ensuring the correctness of the role hierarchy with all the inherited rights.

```
SELECT @@GLOBAL.MAX_EXECUTION_TIME,
          @@SESSION.MAX_EXECUTION_TIME;
SET SESSION cte_max_recursion_depth =
          1000000000;
WITH RECURSIVE RoleHierarchy AS(
          SELECT id, parent_role_id
          FROM Roles
          WHERE id = ’role_id’
          UNION ALL
          SELECT r.id, r.parent_role_id
          FROM Roles r
          JOIN RoleHierarchy rh ON rh.
                parent_role_id = r.id)
SELECT id AS role_id FROM RoleHierarchy;
```

10. Query for listing roles inherited from a specific role

Finding Ancestor Roles of a Specific Role This returns all roles passed through by any given position, hence laying a chain to the genealogy and permission inheritance chain of the role. Such information is useful to debug and ensure that the role inheritance actually works. This guarantees that permissions are correctly inherited by checking the ancestor roles in the access control architecture.

```
SET SESSIONcte_max_recursion_depth =1000000000;
WITH RECURSIVE AncestorRoles AS (
          SELECT id, parent_role_id
          FROM Roles
          WHERE id = 3
          UNION ALL
          SELECT r.id, r.parent_role_id
          FROM Roles r
          JOIN AncestorRoles ar ON ar.
                parent_role_id = r.id)
SELECT parent_role_id AS ancestor_role_id
FROM AncestorRoles
WHERE parent_role_id IS NOT NULL;
```

11. Query for finding ancestor roles of a specific role

Detecting users missing required prerequisites this query detects users who are assigned a role but do not pass prerequisites associated with restrictions. Each role assignment must be totally compliant with the design of the RBAC model and, most significantly, with the governance needs. It is thus that this query also brings out the enhancement of integrity in role assignment because it will prevent placing the users under roles for which they can pose threats to security within the system.

```
WITH RolePrerequisites AS (
          SELECT role_id, prerequisite_role_id FROM
          Constraints
            WHERE prerequisite_role_id IS NOT NULL)
 SELECT DISTINCT ur.user_id, rp.role_id AS
          role_without_prerequisite_id, rp.
        prerequisite_role_id AS
        missing_prerequisite_role_id
```

```
FROM UserRoles ur
JOIN RolePrerequisites rp ON ur.role_id = rp.role_id
WHERE NOT EXISTS (
  SELECT 1
  FROM UserRoles ur2
  WHERE ur2.user_id = ur.user_id
  AND ur2.role_id = rp.prerequisite_role_id);
```

12. Query for detecting users missing required prerequisites

Finding users with specific permissions via role inheritance it can be used to determine who has been granted some rights by role inheritance. The following is a query that can be very useful, actually serving as an audit check to ensure such rights are correctly being inherited. It guarantees the user really has that right although these have been achieved by some indirect inheritance of roles so that the integrity of the access control model is not at stake.

```
SELECT DISTINCT ur.user_id
FROM UserRoles ur
JOIN RolePermissions rp ON ur.role_id = rp .role_id
JOIN Roles r ON ur.role_id = r.id
LEFT JOIN Roles parent ON r.parent_role_id
= parent.id
LEFT JOIN RolePermissions prp ON parent.id
= prp.role_id
WHERE rp.permission_id = 4533 OR prp.
permission_id = 4533;
```

13. Query for searching users with specific permissions from role inheritance

Results

This section presents a comparative analysis of the performance between Neo4j and MySQL databases based on the execution times of various queries related to RoleBased Access Control (RBAC) which is visible from Figs. (4-5). This research concentrates on the efficiency of execution for complex queries, with respect to hierarchical relationships, recursive operations, and inheritance of permissions.

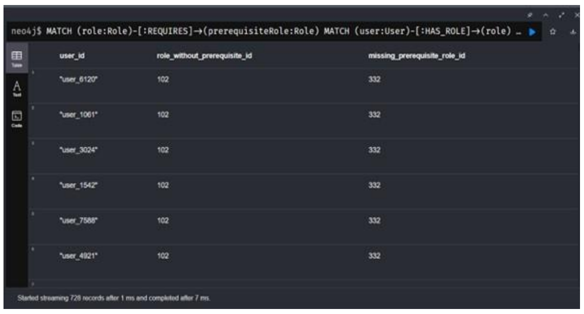
Query Execution Time Comparison

Table (1) condenses the execution time of five queries executed on both graphs as well as relational databases.

The study found that graph-based architecture particularly excels in the processing of queries that require deep relationship traversals typical of RBAC models. This is very evident through huge time differences in recursive and ancestor retrieval queries where relational databases either did not complete the query within a reasonable time or encountered a complete timeout. On the other hand, graph databases efficiently ran such queries, thus making them effective in managing the complex relationships germane to graphs.

Table 1: Comparison of query execution times between Neo4j and MySQL

Query	Neo4j Time (in ms)	MySQL Time (in ms)
Query 1	8	16
Query 2	7 – 42	1888000 - failed
Query 3	6 – 9	1800000 - failed
Query 4	1 – 7	32
Query 5	7 – 11	31

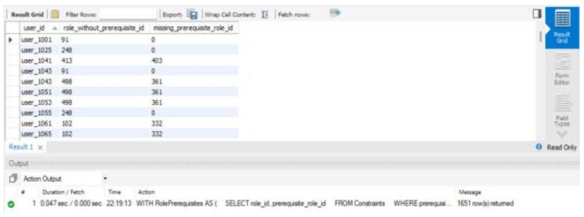


```
neo4j$ MATCH (role:Role)-[:REQUIRES]→(prerequisiteRole:Role) MATCH (user:User)-[:HAS_ROLE]→(role) …

user_id        role_without_prerequisite_id        missing_prerequisite_role_id
"user_6120"    102                                 332
"user_1061"    102                                 332
"user_3024"    102                                 332
"user_1542"    102                                 332
"user_7588"    102                                 332
"user_4921"    102                                 332

Started streaming 728 records after 1 ms and completed after 7 ms.
```

Fig. 4: Result of query 4 execution in Neo4j



```
user_id     role_without_prerequisite_id   missing_prerequisite_role_id
user_1001   91                             0
user_1025   248                            0
user_1041   413                            403
user_1043   91                             0
user_1043   498                            361
user_1051   498                            361
user_1053   498                            361
user_1055   248                            0
user_1061   102                            332
user_1065   102                            332

Action Output
#  Duration / Fetch       Time      Action                                                                                                 Message
1  0.047 sec / 0.000 sec  22:19:13  WITH RolePrerequisites AS (  SELECT role_id, prerequisite_role_id  FROM Constraints  WHERE prerequisi...  1651 row(s) returned
```

Fig. 5: Result of query 4 execution in SQL

The results also prove that graph databases are far better at enforcing the RBAC restrictions, for instance, identifying users who do not have the prerequisites required and permission inheritance through role hierarchies. The efficiency of graph databases in making such queries quickly without much latency proves its superiority over relational databases where robust and dynamic methods of access control are required.

Overall, the findings suggest that Neo4j is a more appropriate choice for systems that require efficient handling of complex relationships and hierarchical data structures. MySQL, with its relational model, struggles to perform at the same level, particularly in queries that involve deep recursion or intricate role hierarchies. This performance gap illustrates the inherent advantages of using a graph database like Neo4j for implementing advanced access control systems and managing intricate data relationships.

Discussion

The results indicate a huge performance gap between Neo4j and MySQL, at least in the case of queries that involve deep recursion and hierarchical traversal-a scenario most typical for Role-Based Access Control systems. Conceptually, Neo4j provides a graph-based architecture visible in Fig. (6) leads to faster and more efficient handling of relations, making dynamic access control feasible.

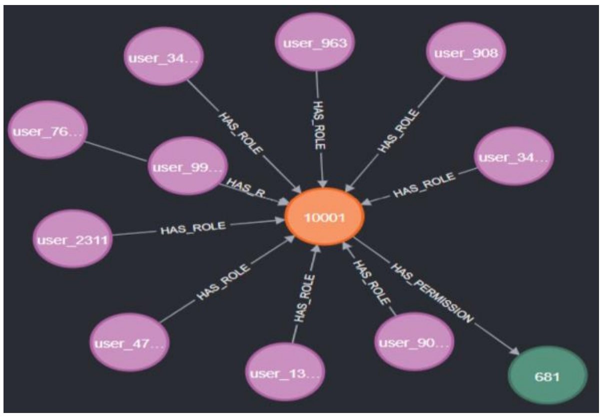


Fig. 6: Nodes and relationship in Neo4j

Indeed, the table and figures illustrate how graph databases outclass relational databases that either perform abysmally or fail in strict conditions of high complexity and execution-time constraints.

Neo4j graph databases support scalability and flexibility of access, allowing them to suit modern systems much better where rapid data access is a high priority, especially when dealing with complex structures. MySQL's relational model is less flexible and ranges, especially when dealing with some extremely recursive operations or multi-level permission hierarchies.

Conclusion

This research underscores the critical differences between graph databases, such as Neo4j, and traditional relational databases, like MySQL, in the context of Role-Based Access Control (RBAC).

By employing a project management case study, we illustrated that Neo4j significantly outperforms MySQL in managing complex relationships, offering superior query processing speed and enhanced flexibility in representing hierarchical structures and dynamic roles. While MySQL remains a robust solution, particularly in scenarios demanding strict and well-defined access controls, the inherent scalability and adaptability of Neo4j present compelling advantages for modern, large-scale RBAC systems.

Looking forward, the integration of graph databases in RBAC systems holds immense potential for further exploration, especially in the realm of real-time access control in dynamic and distributed environments. Future research could delve into the application of Neo4j in hybrid systems that combine relational and graph databases to harness the strengths of both, potentially leading to more robust, scalable, and adaptable RBAC frameworks. As organizational data structures continue to grow in complexity, the role of graph databases in securing and managing access will only become more pivotal, driving innovation in access control mechanisms.

Acknowledgment

Thank you to the publisher for their support in the publication of this research article. We are grateful for the resources and platform provided by the publisher, which have enabled us to share our findings with a wider audience. We appreciate the efforts of the editorial team in reviewing and editing our work, and we are thankful for the opportunity to contribute to the field of research through this publication.

Funding Information

There was no external funding for this article.

Author's Contributions

Kriti Srivastava: Designed the research plan and organized the study.

Dhruv Jain, Aarya Kamdar, Anuradha Yeole, Devam Shah and Sowmya Dadheech: Participated in all experiments, coordinated the data-analysis and contributed to the writing of the manuscript.

Ethics

The authors declare no ethical issues.

References

Batra, S., & Tyagi, C. (2012). Comparative Analysis of Relational and Graph Databases. *International Journal of Soft Computing and Engineering (IJSCE*, *2*(2), 509–512.

Bechberger, D., & Perryman, J. (2020). *Graph Databases in Action* (1st ed.). Manning. ISBN-10: 9781617296376.

Do, T.-T.-T., Mai-Hoang, T.-B., Nguyen, V.-Q., & Huynh, Q.-T. (2022). Query-based Performance Comparison of Graph Database and Relational Database. *The 11th International Symposium on Information and Communication Technology*, 375–381. https://doi.org/10.1145/3568562.3568648

Fan, W. (2012). Graph Pattern Matching Revised for Social Network Analysis. *Proceedings of the 15th International Conference on Database Theory*, 8–21. https://doi.org/10.1145/2274576.2274578

Gupta, M., & Aggarwal, R. R. (2014). Transforming Relational Database to Graph Database Using Neo4j. *Proceedings of the Second International Conference on Emerging Research in Computing, Information, Communication and Applications*, 322–331.

Huang, Y.-W., Jing, N., & Rundensteiner, E. A. (1996). Effective Graph Clustering for Path Queries in Digital Map Databases. *Proceedings of the Fifth International Conference on Information and Knowledge Management - CIKM '96*, 215–222. https://doi.org/10.1145/238355.238497

Jain, M., Khanchandani, A., & Rodrigues, C. (2023). *Performance Comparison of Graph Database and Relational Database*. https://doi.org/10.13140/RG.2.2.27380.32641

Khan, W., Ahmed, E., & Shahzad, W. (2017). Predictive Performance Comparison Analysis of Relational and amp; NoSQL Graph Databases. *International Journal of Advanced Computer Science and Applications*, *8*(5). https://doi.org/10.14569/ijacsa.2017.080564

Kotiranta, P., Junkkari, M., & Nummenmaa, J. (2022). Performance of Graph and Relational Databases in Complex Queries. *Applied Sciences*, *12*(13), 6490. https://doi.org/10.3390/app12136490

Medhi, S., & Baruah, H. (2017). Relational Database and Graph Database: A Comparative Analysis. *Journal of Process Management. New Technologies*, *5*(2), 1–9. https://doi.org/10.5937/jouproman5-13553

Nguyen, V.-Q., & Kim, K. (2017). Estimating the Evaluation Cost of Regular Path Queries on Large Graphs. *Proceedings of the Eighth International Symposium on Information and Communication Technology*, 92–99. https://doi.org/10.1145/3155133.3155160

Sharma, M., Sharma, V. D., & Bundele, M. M. (2018). Performance Analysis of RDBMS and No SQL Databases: PostgreSQL, MongoDB and Neo4j. *2018 3rd International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE)*, 1–5. https://doi.org/10.1109/icraie.2018.8710439

Sholichah, R. J., Imrona, M., & Alamsyah, A. (2020). Performance Analysis of Neo4j and MySQL Databases using Public Policies Decision Making Data. *2020 7th International Conference on Information Technology, Computer, and Electrical Engineering (ICITACEE)*, 152–157. https://doi.org/10.1109/icitacee50144.2020.9239206

Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., & Wilkins, D. (2010). A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective. *Proceedings of the 48th Annual Southeast Regional Conference*, 1–6. https://doi.org/10.1145/1900008.1900067