ANALYSIS OF ROLE-BASED ACCESS CONTROL METHODS IN NOSQL DATABASES

By

Heena Khan

A thesis submitted in partial fulfillment

of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

Middle Tennessee State University

April 2019

Supervised by

Dr. Suk Seo

ACKNOWLEDGEMENTS

I would first like to thank my thesis advisor, Dr. Suk Seo, for her constant support and guidance. She was always available to guide me whenever I ran into trouble. Her constant input and remarks, helped me to improve my writing and my work.

I would also like to thank my two reviewers, as the second reader of this thesis, and for their valuable comments and suggestions.

ABSTRACT

An increasing number of companies are using non-relational NoSQL (Not only SQL) databases today because of growing, unmanageable data. However, data security is the main concern while using a NoSQL database. One of the most common methods of data security in NoSQL is assigning permission to files and data. Assigning permission with respect to users role in the company is known as Role-Based Access Control (RBAC). It is a mechanism to restrict access to unauthorized users. Different NoSQL databases have different working conditions and characteristics of an RBAC. The main goal of this paper is to analyze RBAC implementation in four types of NoSQL databases, which are MongoDB, Cassandra, Redis, and Neo4j. In this project, we are implementing a health care management system which manages patient data. The RBAC implementation for each database is analyzed and its performance is evaluated with respect to the other NoSQL databases, and the traditional RDBMS SQL Server.

Keywords - Database, NoSQL, Role based access control, RBAC, MongoDB, Cassandra, Redis, Neo4j, SQL Server.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

- CQL - Cassandra Query Language
- SQL - Standardized Query Language
- JSON - JavaScript Object Notation
- RBAC - Role Based Access Control
- DBMS - Database Management System
- NoSQL - Not Only SQL
- RDBMS - Relational Database Management System

CHAPTER I.
INTRODUCTION

In the last few years, there has been a tremendous increase in data either structured, semi-structured, or unstructured. Such an enormous and complex set of Big Data is difficult to manage by the traditional Relational Database Management System (RDBMS). Traditional RDBMS like MySQL, Oracle, and SQL Server are based on a relational schema. NoSQL, on the other hand, is a non-relational database that does not require any specific schema for the data to be stored. Any sort of data without any specified structure can be stored and accessed making it quick and efficient to access, and it also supports sharding i.e Horizontal partitioning of data. It enables NoSQL to store and handle a large amount of data [11]. There are different types of NoSQL database that take different approaches. What they have in common is that theyre not relational [17]. The four types of NoSQL are as follows.

- Document Store Databases (E.g. MongoDB, CouchDB, OrientDB)
- Column Oriented Databases (E.g. Cassandra, Hbase)
- Key-Value Store Databases (E.g. DynamoDB, Redis)
- Graph Databases (E.g. Neo4j)

Though NoSQL is an optimal solution for handling Big Data, it lacks polish access control and data privacy protection. A majority of Big Data platforms integrate quite basic access control enforcement system [8]. In this paper, we are examining how fine-grained is the existing RBAC implementation in four types of NoSQL databases.

CHAPTER II.
BACKGROUND

Data security has been the main concern related to NoSQL databases. Many papers have been published about NoSQL data integrity issues, schemes, and solutions have been proposed for maintaining data confidentiality and integrity [15]. External attacks like Injection attacks, HTTP trespassing attacks, and Trojan horse attacks can lead to an unauthorized flow of data and can cause confidentiality or integrity violation [12, 19]. Lack of polished access control is one of the reasons for unauthorized access of data.

The access control is a common method to achieve data security. Access control is used to identify the user, authenticate them, and then provide access only to the information or data that they are authorized to use. Access control systems are of three types: Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Role Based Access Control (RBAC). A polished access control system in NoSQL is still at a very early state. Only a few access control methods have been suggested until now [10]. There are various papers that propose different methodologies supporting RBAC implementation in NoSQL. Colombo and Ferrari, proposed a purpose based access control method for MongoDB NoSQL [9]. Kulkarni suggested key-value access control (K-VAC) [14], which has been already implemented in Cassandra. Shalabi and Gudes, suggested cryptographic encryption method for column-oriented database [18]. Morgado proposed a security model for access control in graph-based NoSQL [16]. Their common goal is to design a fine-grained access control method which helps in maintaining data completeness and correctness. To the best of our knowledge, there is no previous work where all four types of the NoSQL databases are studied and a comparative result is drawn with respect to access control achieved by the existing RBAC system implemented in them.

CHAPTER III.
APPROACH AND METHODS

A NoSQL database is profoundly known to be implemented for applications managing Big Data. With time, NoSQL has become more commercial and is used even by applications that are more user-centric. It is very important to have good user access control. To analyze the implementation of RBAC in different NoSQL databases, we are going to implement an application that has a lot of users with different roles. Each user will have different permissions to execute the operation on the object, depending on the roles assigned to them. Figure 1 shows the relationship between users, roles and permission.[6]

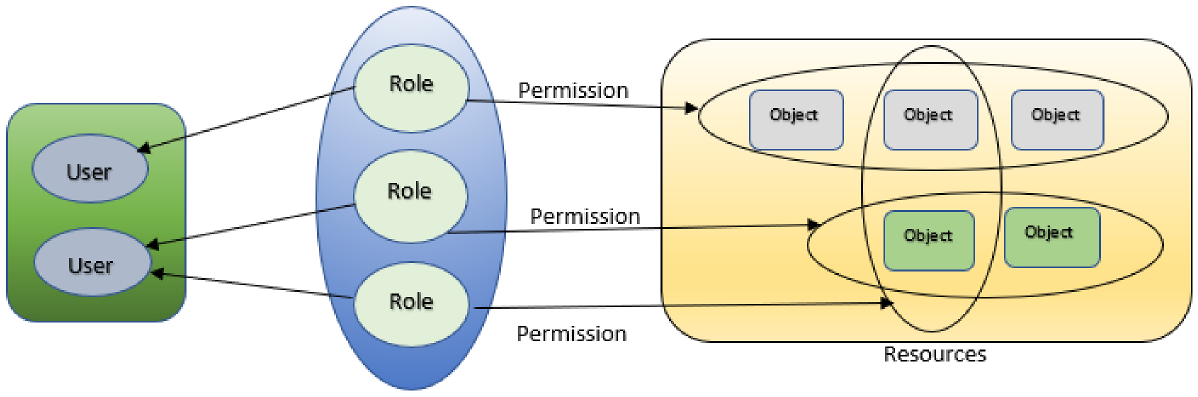


Figure 1: Roles and permission in RBAC

In this project, we have implemented an application that is a small model of the health care management system. Although a traditional relational database would have been more appropriate for this application, it is a good application to test the RBAC capabilities of NoSQL. The model contains five tables with doctors, nurses, pharmacists, patients, and patients' medication information. The application has three roles names including a nurse, a doctor and a pharmacist. Each role has five users. A nurse should have read permission on patients and their medication information; a doctor should have both read and write permission, and a pharmacist should only be allowed to view patients' medicines' details. A pharmacist should not have any access to patients' names, diseases or symptoms and he should only have access to the patients for whom he is providing medicine.

Table 1 represents permission assigned to the roles.

Table 1: Represents roles permission

Resource	Roles		
	Doctor	Nurse	Pharmacist
Patients_Data	R, W	R	-
Patients_Medication	R, W	R	-
vwPatients_Medication	-	-	R (Partial)

We are evaluating the RBAC implemented in four different types of NoSQL databases with respect to RBAC granularity provided by Relational Database: SQL Server. We implemented the health care management system in SQL Server [7]. Figure 2 represents the application implementation schema in SQL Server.

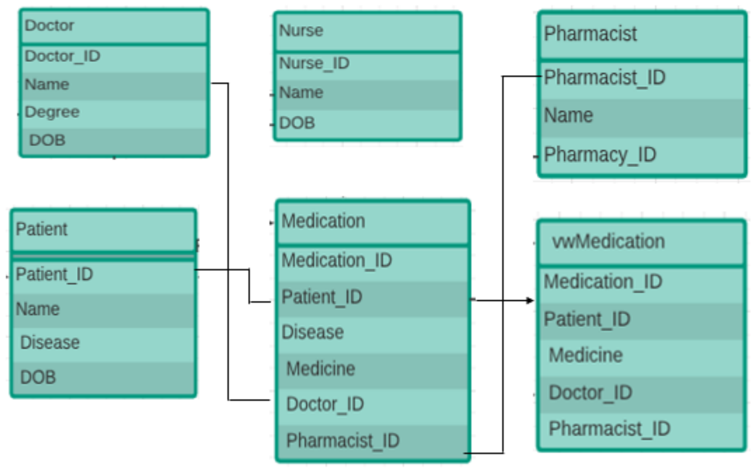
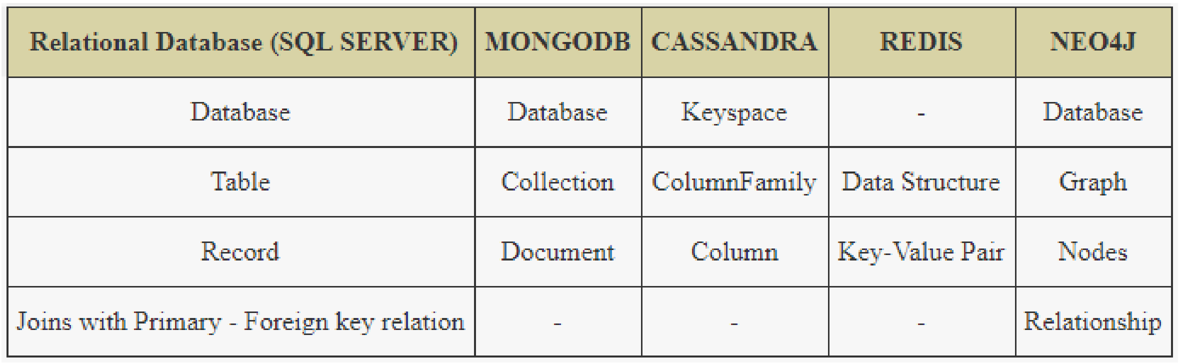


Figure 2: Health Care Management System Schema

We implemented the above schema in MongoDB, Cassandra, Redis, and Neo4j. As NoSQL does not require a schema definition, data is stored in different forms. In MongoDB, they are stored in the form of collection and document. In Cassandra, they are stored as tables and fields. In Redis, they are stored using a hash map and key-value pair. In Neo4j, they are stored in the form of Nodes and relationships. The RBAC model in our application requires cell-level granularity of access control. Figure 3 represents structure of different NOSQL



Relational Database (SQL SERVER)	MONGODB	CASSANDRA	REDIS	NEO4J
Database	Database	Keyspace	-	Database
Table	Collection	ColumnFamily	Data Structure	Graph
Record	Document	Column	Key-Value Pair	Nodes
Joins with Primary - Foreign key relation	-	-	-	Relationship

Figure 3: Structure of NoSQL databases

To implement the access control mechanism we chose the following databases.

1. SQL Server - Microsoft SQL Server 2017
2. MongoDB - MongoDB version 4.0
3. Cassandra - Cassandra version 3.11
4. Redis - Redis version 5.0.4
5. Neo4j - Neo4j Desktop 1.1.18

Implementation

There are three important steps to implement RBAC in your application: 1) identify an individual who does a specific job (user), 2) Assign role to the individual, and 3) then assign permission and privileges to the role. The user should be authorized to access the permission and privileges assigned to his role. This section describes the implementation of RBAC in SQL Server, MongoDB, Cassandra, Redis, and Neo4j respectively.

SQL Server

SQL server is one of the oldest and most popular relational database system with one of the finest RBAC implementation [2]. SQL Server provides a good set of built-in fixed server and database roles with predefined privileges and permissions. It also support custom roles where permission is assigned using GRANT and DENY command. RBAC in SQL Server supports DENY statement to restrict access of roles.

For the RBAC implementation in SQL Server, we created a schema named ”Hospital” with five tables: tblDoctor, tblNurse, tblPharmacist, tblPatient and tblMedication [7]. We created three roles: Doctor, Nurse and Pharmacist. ”Doctor” is assigned read and write permission on ”tblMedication” and ”tblPatients”. ”Nurse” is assigned read permission on both ”tblPatient” and ”tblMedication” table. ”Pharmacist” should not have access to patients name and disease. We created a view named ”vwMedication” from ”tblMedication” hiding patients name and disease. The pharmacist should only be allowed to see patients’ to whom he is providing medicine and not all patients.

Figure 4 shows SQL commands for role creation and permission assignment.

```
//--Create Role named Doctor
CREATE ROLE Doctor;
//--Assign Permission to Role doctor
GRANT SELECT, INSERT, UPDATE, DELETE ON Hospital.tblMedication TO Doctor;
GRANT SELECT, INSERT, UPDATE, DELETE ON Hospital.tblPatient TO Doctor;
DENY DELETE ON Hospital.tblPatient TO Doctor;
//--Assign user Doctor1 to the role Doctor
ALTER ROLE Doctor ADD MEMBER Doctor1
```

Figure 4: Permission assignment to roles in SQL Server.

In this project, we created 5 users for each role. The permission of a user is decided according to the role assigned to them. We will discuss whether the users had access as expected and the level of access control supported by SQL Server in the result analysis section.

MongoDB

MongoDB is an open-source document store NoSQL database [3]. MongoDB supports Role-based authorization at a quite granular level. The RBAC model for MongoDB is characterized by concepts like privilege, action, role, data resource, and user [8]. Because MongoDB is schema-less it does not store data in the form of tables and rows. Data is stored in the form of collection and documents. Figure 5 represents MongoDB Data structure.

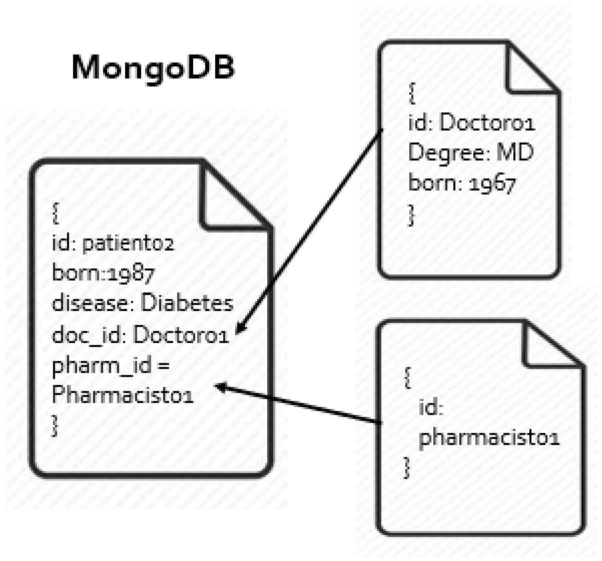


Figure 5: Document in MongoDB NoSQL

For the RBAC implementation in MongoDB, we created a database named ”Hospital” with five collections: Doctor, Nurse, Pharmacist, Patient, Medication. Data is stored in the form of documents using JSON [3]. We created three roles: doctor, nurse and pharmacist. The ”pharmacist” should not have access to patients’ name and disease. We created a view named ”vwMedication” hiding patient’s name and disease. The pharmacist should only be allowed to see patients’ for whom he is providing medicine and not all patients. Figure 6 shows command for role creation and permission assignment in MongoDB.

```
//--CREATE ROLE DOCTOR AND ASSIGN PERMISSION
db.createRole({ role: "doctor",
  privileges: [
       { resource: { db: "Hospital" , collection: "Medication" },
         actions: [ "find", "update", "insert", "remove" ] },
       { resource: { db: "Hospital", collection: "Patient" },
         actions: [ "insert", "update", "find" ] },],
  roles: []})
//--CREATE USER AND ASSIGN ROLE DOCTOR
db.createUser({ user : 'Doctor01', pwd : 'doctor',
                roles : [ { role : 'doctor', db : 'Hospital' }]
              })
```

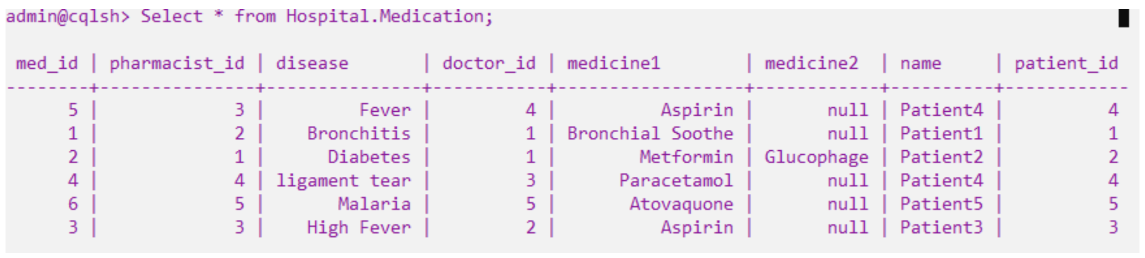
Figure 6: Permission assignment to roles and user in MongoDB.

The command used to create a role is db.CreateRole(), permission is assigned by command ”actions: ”, and object on which permission is assigned is defined by command ”resource: ”. The command used to create user in MongoDB is db.CreateUser().

On implementation, the permission assigned to the users with role ”doctor” and ”nurse” were as expected, but the permission assigned to the ”pharmacist” violates the actual permission a pharmacist should have. Because MongoDB does not provides permission assignment on the document level. The role ”pharmacist” was assigned permission to read entire ”vwMedication” view.

Cassandra

Cassandra is a distributed storage system which manages very large volume of structured data spread out across many commodity servers while providing highly available service with no single point of failure [13, 18]. In Cassandra, Data is stored in the form of a keyspace and Column Family. Column Family is formed by combining number of columns together. Records within the column family are stored in different nodes using partition keys, which makes Cassandra fast and reliable [1]. Cassandra does not provide a large range of built-in roles. ”Superuser” is the only administrative built-in role provided by Cassandra. Cassandra supports user-defined custom roles. Permission is granted on object using (ALTER, DROP, GRANT, REVOKE) commands. Cassandra does not support DENY command. In Cassandra syntax, users are represented as roles. Cassandra roles can be assigned to other roles using permission assignment commands [18]. Figure 7 represents Cassandra Data structure.



```
admin@cqlsh> Select * from Hospital.Medication;

 med_id | pharmacist_id | disease       | doctor_id | medicine1        | medicine2  | name     | patient_id
--------+---------------+---------------+-----------+------------------+------------+----------+------------
      5 |             3 |         Fever |         4 |          Aspirin |       null | Patient4 |          4
      1 |             2 |    Bronchitis |         1 | Bronchial Soothe |       null | Patient1 |          1
      2 |             1 |      Diabetes |         1 |        Metformin | Glucophage | Patient2 |          2
      4 |             4 | ligament tear |         3 |      Paracetamol |       null | Patient4 |          4
      6 |             5 |       Malaria |         5 |       Atovaquone |       null | Patient5 |          5
      3 |             3 |    High Fever |         2 |          Aspirin |       null | Patient3 |          3
```

Figure 7: Column Family in Cassandra NoSQL

For the RBAC implementation in Cassandra, we created a Keyspace named ”Hospital” with five column-Family (tables): ”Doctor”, Nurse, Pharmacist, Patient, and Medication. Data is stored in the form of tables known as Column Family where data is stored in columns as key-value pair. We created three roles: doctor, nurse and pharmacist. The ”pharmacist” should not have access to patients' name and disease. We created a materialized view named ”vwMedication” to hide name and disease of patients. The pharmacist should only be allowed to see patients' to whom he is providing medicine.

Figure 8 shows commands for role creation and permission assignment in Cassandra.

```
//Create Role Doctor
    CREATE ROLE Doctor;
//Assign Permission to role
    GRANT Select, Modify on Hospital.Patient To Doctor;
    GRANT Select, Modify on Hospital.Medication To Doctor;
    GRANT Select, Modify on Hospital.vwMedication To Doctor;
//Create User Doctor1
    CREATE ROLE Doctor1 WITH Login = true and password = 'doctor';
//Assign Role Doctor
    GRANT Doctor TO Doctor1;
```

Figure 8: Permission assignment to roles and user in Cassandra.

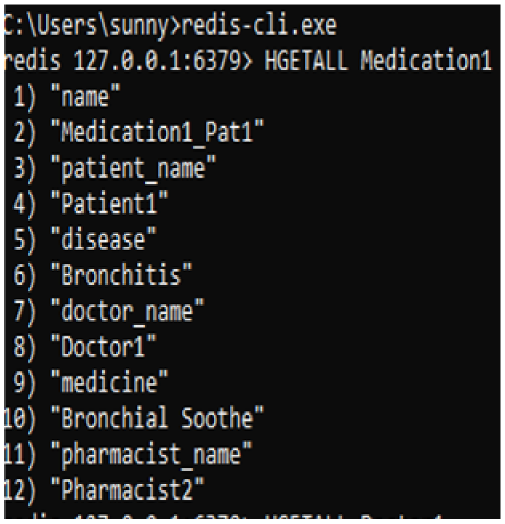
Cassandra supports Cassandra Query Language (CQL) and it is similar to the Standardizes Query Language (SQL). User is created as role in cassandra. The command used to create role and user is CREATE ROLE RoleName.

We observed that the permission assigned to the users' with role "doctor" and "nurse" where as expected, but the permission assigned to the pharmacist violates the actual permission a pharmacist should have. With command "RESTRICT ROWS ON Hospital.vwMedication USING PharmacistID;" We were able to restrict user permission to certain rows with respect to the values in column PharmacistID. On using RESTRICT command we had to grant permission individually to each user for their access. There was no way to assign row level permission in Cassandra to a role. So, the "pharmacist" was assigned read permission on entire "vwMedication" view.

Redis

Redis is an open source, in-memory data structure store, used as a database, cache and message broker [5]. It supports data structures such as strings, hashes, lists, sets, sorted sets and stores data in the form of key-value pair [5]. Redis does not support custom user-defined roles. A Redis Enterprise cluster is composed of identical nodes. Redis support RBAC at cluster level. Users that are connected on the same cluster can be assigned built-in roles.

Figure 9 represents Data structure in Redis



```
C:\Users\sunny>redis-cli.exe
redis 127.0.0.1:6379> HGETALL Medication1
 1) "name"
 2) "Medication1_Pat1"
 3) "patient_name"
 4) "Patient1"
 5) "disease"
 6) "Bronchitis"
 7) "doctor_name"
 8) "Doctor1"
 9) "medicine"
10) "Bronchial Soothe"
11) "pharmacist_name"
12) "Pharmacist2"
```

Figure 9: HashMap in Redis NoSQL

Since there is no user-defined custom roles, we couldn't create roles for doctor, nurse or pharmacist. On cluster we assigned built-in role named DB viewer to all users whether doctor, nurse or pharmacist. While we were working with single node there was no roles. To enable authentication we used 'CONFIG SET requirepass password123' to add password for authentication while connecting to the server.

Neo4j

Neo4j is a graph based NoSQL Database [4]. Unlike other databases, relationships take the first priority in graph databases. A graph is composed of two elements: a node and a relationship [4]. Cypher Query Language is the query language supported by Neo4j.Figure 10 shows data structure of Neo4j.

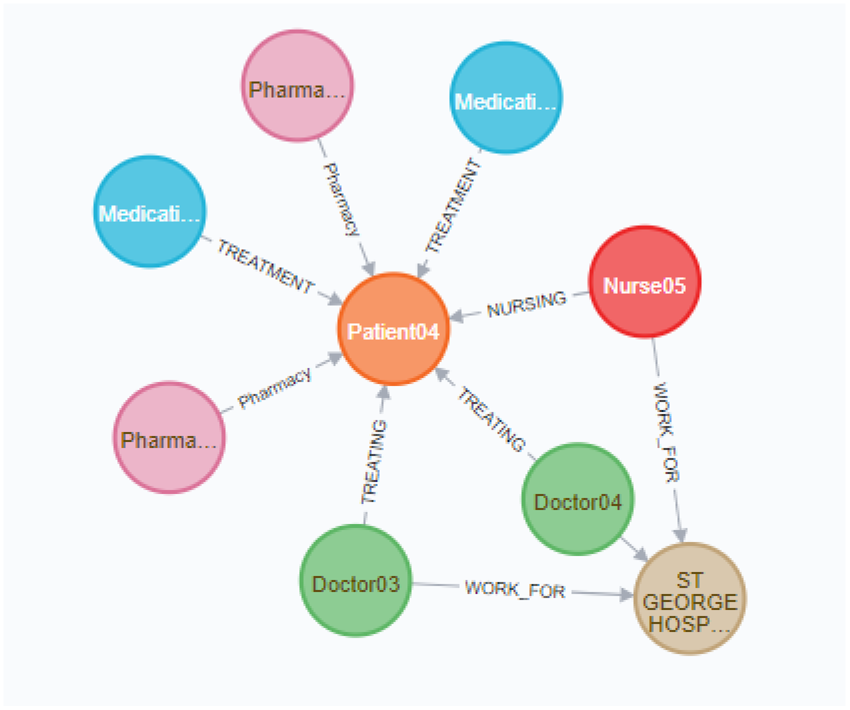


Figure 10: Nodes and Relationship in Neo4j NoSQL

The RBAC model in Neo4j has built-in roles with read, write, modify and administrative permission on the graph and its configuration. User with no assigned roles have no access to graphs data, not even read privileges. Users who have been assigned more than one role have privileges of both the roles. Neo4j also supports user-defined custom roles. The users' to whom custom roles are assigned, do not have access to the graph. These users just have access to the custom user-defined procedures to which they have permission. Node level security is not yet supported in the RBAC introduced in Neo4j 3.1 [4]. However, the only option we have now to provide graph access to custom roles is by implementing a procedure-based security model. In this method, the definition of the sub-graph is encoded into the code of the procedure [4].

Neo4j also support security through user clearance at different security level. Blacklist is the command to prevent the roles from accessing certain properties. To blacklist a property you first have to enable ”Blacklist”. Blacklisting a property is a little cumbersome process as you have to set different values for different security levels thus, making it in-efficient method to implement RBAC.

Figure 11 shows how user clearance can be used to blacklist certain properties and figure 12 shows permission assignment and role creation in Neo4j.

```
dbms.security.property_level.blacklist=Secret=doctor;
Confidential=Nurse,doctor;Unclassified=pharmacist,doctor,nurse;
//create node with different clearance
CREATE (u:Medication {name:'Medication01_Pat01', patient_id:'Patient01',
    doctor_id:'Doctor01', Medicine: 'Bronchial Soothe',
    pharmacist_id:'Pharmacist02', disease:'?' doctor: 'Bronchitis', nurse
    :'Bronchitis!'})
```

Figure 11: Example to show blacklisting certain properties in Neo4j

```
//--Create user
CALL dbms.security.createUser(pharma01, abc123);
//--Create custom Roles
CALL dbms.security.createRole('Pharmacist')
//--Assign permission to role:
(Pharmacist { type: 'role', name: 'Pharmacist' }),
Pharmacist-[r1:BELONGS_TO]->reader,
//--Add user to rule
CALL dbms.security.addRoleToUser('pharmacist', 'pharma01')
```

Figure 12: Permission assignment to users in Neo4j

CHAPTER IV.
RESULT ANALYSIS

The goal of this project was to analyze the implementation of RBAC in NoSQL databases. Our method was to implement an application that contains large number users with different privileges. In this project, We tried to analyze the ease and feasibility of implementation in each database. We analyzed how fine-grained they are, the built-in roles and custom roles provide by each of them, the different data structure and query language supported by each database, and their behaviour on insertion and deletion.

Our expectation for the RBAC model in our application are as follows: ”Doctor” should have read/write/update permission on patient and his medical information. A ”Nurse” should have read permission on patient and his medical information. A ”Pharmacist” should have restricted read permission on medical information. A ”Pharmacist” should not be allowed to view patient’s disease/symptoms and name.

In SQL Server, role-based access control is achieved at a cell-level granularity. This can be achieved with the help of views and security policies provided by SQL Server. When we implemented our health care management system model in SQL Server, it was observed that all the users had privileges that were expected by their respective roles. The relational database had stayed in the market for almost half a century now, because of their security performance. The RBAC implementation in SQL Server was able to provide the granularity required by a user based application, such as Health care management system. Thus, SQL Server can be a good benchmark to test access control in other databases. The observation we made during the implementation of our application in NoSQL databases are discussed below in detail.

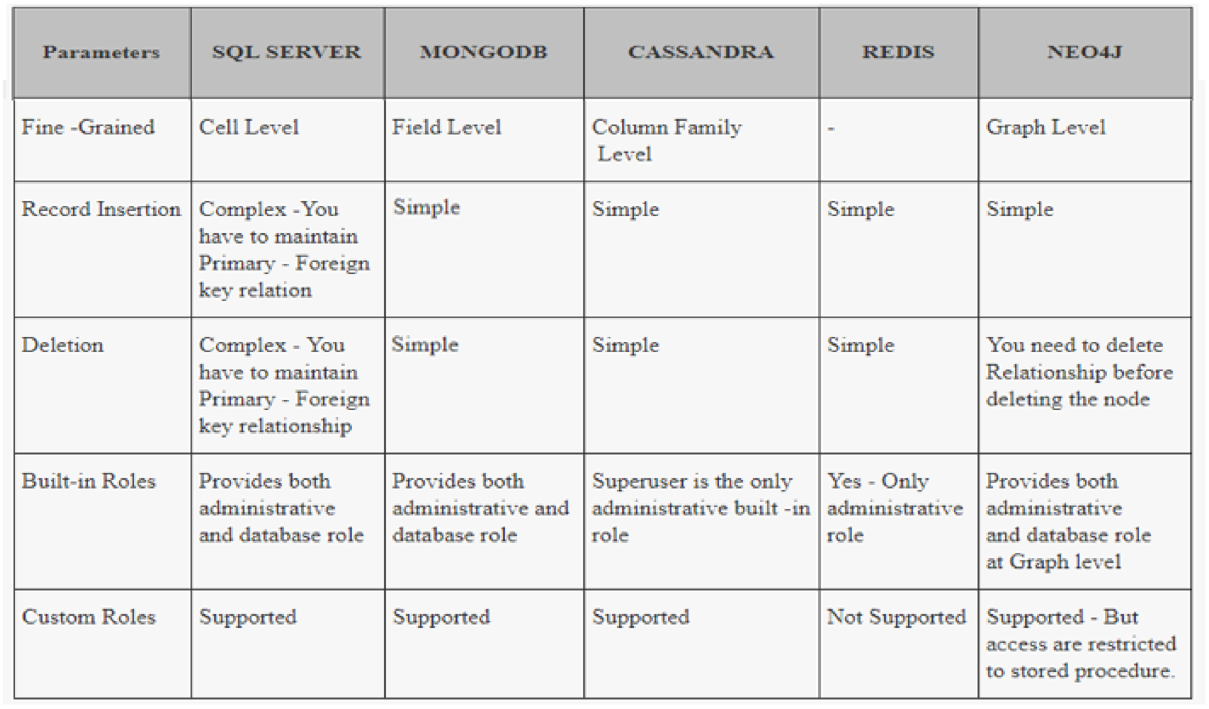
MongoDB provides built-in roles for both database administration and database users. The user-defined custom roles are also supported. Role-based access control is not enabled by default and it must be configured by the database administrator [3]. RBAC supported by

MongoDB is comparatively advance, with respect to other NoSQL. It provides a huge set of built-in roles and it is comparatively easier to create and manage custom user-defined roles in MongoDB. Database-level and collection-level security can be achieved easily in MongoDB, but as of now, it does not support document-level security. MongoDB also provides field-level access control. In this project, we were able to create a view using query filters to hide certain fields within a document. During the permission assignment, we were able to assign permission to ”doctor” and ”nurse” as required. The permission assigned to the pharmacist violates our RBAC model expectation. Because the pharmacist could see the information of all the patients’, though he couldn’t see their name and diseases. In MongoDB insertion and deletion of data is simple as there are no primary key-foreign key dependencies.

Cassandras’ RBAC model, does not contain a large set of built-in roles. ”Superuser” is the only built-in role in Cassandra with administrative privileges. Users in CQL are also defined as roles. In Cassandra, roles are first-class database objects and so they have permission defined on themselves too [18]. A role may be assigned to other roles [18]. In our implementation, we created a materialized view ”vwMedication” from the column-family Medication. This view was created to hide patients name and disease column from the pharmacist. As the ”doctor” has read/write permission on Medication he should not be required to have permission on ”vwMedication”. But we observed that the doctor must have modified permission on the view ”vwMedication” in order to modify table Medication. Also, we were not able to assign row-level access to ”pharmacist”. We assigned ”pharmacist” with reading permission on the entire view. We then observed that the pharmacist should have read access to the Medication table too, to access the view, which negates the entire purpose of creating a view and also violates our RBAC requirement.

Redis does not support RBAC in single node application. Redis Enterprise supports administrative built-in roles for cluster management. The permission assignment through built-in roles is at the database and cluster level. A cluster can have 100 of databases. In our RBAC implementation, there were no custom roles like doctor, nurse or pharmacist. Thus, we assigned our users with 'DB viewer' role with this role they can view info and configuration of all the databases. We were not able to implement our RBAC model in Redis. Redis is a database where data access should be provided only to trusted people. Redis has a good password authentication mechanism, but it is suggested to have a complex password for your server. Because with the Redis speed it will be easy to break the password with various permutation.

In Neo4j, RBAC implementation is quite basic. Neo4j introduced a new RBAC framework in version 3.1 which was introduced in March 2017, this version provides additional privileges to admin, and have extended built-in roles. They have also introduced user-defined custom roles, but the user assigned to custom roles does not have access permission on any node or their relationships. These users can only access user-defined stored procedure to which they have permission. A user who has no role assigned will not have any rights or capabilities regarding the data, not even read privileges. The privileges assigned through built-in function are at the Graph level. Node level security is still not supported in Neo4j. On assigning built-in read/write roles to our user's. The "doctor" had read/write access on the entire graph, "nurse" and "pharmacist" had read permission on the entire graph which violates are RBAC model expectation. Insertion in Neo4j is simple no foreign and primary key maintenance is required, but on deletion, you cannot just delete a node without deleting relationships. You have to either explicitly delete the relationships, or use DETACH DELETE command [4].



Parameters	SQL SERVER	MONGODB	CASSANDRA	REDIS	NEO4J
Fine -Grained	Cell Level	Field Level	Column Family Level	-	Graph Level
Record Insertion	Complex -You have to maintain Primary - Foreign key relation	Simple	Simple	Simple	Simple
Deletion	Complex - You have to maintain Primary - Foreign key relationship	Simple	Simple	Simple	You need to delete Relationship before deleting the node
Built-in Roles	Provides both administrative and database role	Provides both administrative and database role	Superuser is the only administrative built -in role	Yes - Only administrative role	Provides both administrative and database role at Graph level
Custom Roles	Supported	Supported	Supported	Not Supported	Supported - But access are restricted to stored procedure.

Figure 13: Comparison of RBAC in NosQL databases

CHAPTER V.
DISCUSSION

After comparing RBAC implementation in different NoSQL databases with each other, and also by the RBAC implementation available in a relational database like SQL Server. We realized that while most of NoSQL databases have just a basic implementation of RBAC, some of them has come a long way in their access control model from the time they were first introduced. But they are still not as fine-grained as an RDBMS system. Among the different type of NoSQL databases, MongoDB is the most efficient at the granularity level. NoSQL is basically preferred for its performance as a quick and reliable data source with properties like sharding and partial data distribution among nodes. Access control can be considered as a lower priority for these databases. But in the last few years as these databases are getting commercial, and claims to support all types of businesses. Most of the businesses are extremely user based. Thus, implementing good role-based access control is important. To create a secure application, it is equally important how the administrator implements access control.

Security has been the main concern related to NoSQL databases. There are various method to maintain data security in a database, RBAC being one of them. There are many more NoSQL databases in the market with a different implementation that could be studied. Their implementation on Cloud could be analyzed with respect to data integrity and confidentiality provided by each of them. In the future, we would like to analyze the user authentication method implemented by different types of NoSQL databases.

BIBLIOGRAPHY

[1] Cassandra official documentation. `https://www.datastax.com/dev/blog/role-based-access-control-in-cassandra`.

[2] Microsoft Azure official documentation. `https://docs.microsoft.com/en-us/azure/role-based-access-control/overview`.

[3] MongoDB official documentation. `https://docs.mongodb.com/manual/core/authorization/`.

[4] Neo4j official documentation. `https://neo4j.com/docs/operations-manual/current/tutorial/role-based-access-control/`.

[5] Redis introcuction. `https://redis.io/topics/introduction`.

[6] Bindiganavale, V. and Ouyang, J. Role based access control in enterprise application-security administration and user management. In *2006 IEEE International Conference on Information Reuse & Integration*, pages 111–116. IEEE, 2006.

[7] Burman, S. D. *Comparison of RBAC Implementation for Open Source Databases*. PhD thesis, Rochester Institute of Technology, 2016.

[8] Colombo, P. and Ferrari, E. Privacy aware access control for big data: a research roadmap. *Big Data Research*, 2(4):145–154, 2015.

[9] Colombo, P. and Ferrari, E. Enhancing mongodb with purpose-based access control. *IEEE Transactions on Dependable and Secure Computing*, 14(6):591–604, 2017.

[10] Colombo, P. and Ferrari, E. Access control in the era of big data: State of the art and research directions. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies*, pages 185–192. ACM, 2018.

[11] Gómez, P., Casallas, R., and Roncancio, C. Data schema does matter, even in nosql systems! In *Research Challenges in Information Science (RCIS), 2016 IEEE Tenth International Conference on*, pages 1–6. IEEE, 2016.

[12] Hou, B., Shi, Y., Qian, K., and Tao, L. Towards analyzing mongodb nosql security and designing injection defense solution. In *Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS), 2017 IEEE 3rd International Conference on*, pages 90–95. IEEE, 2017.

[13] Kapadia, G. S. Comparative study of role based access control in cloud databases and nosql databases. *International Journal of Advanced Research in Computer Science*, 8(5), 2017.

[14] Kulkarni, D. A fine-grained access control model for key-value systems. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 161–164. ACM, 2013.

[15] Kumar, J. and Garg, V. Security analysis of unstructured data in nosql mongodb database. In *Computing and Communication Technologies for Smart Nation (IC3TSN), 2017 International Conference on*, pages 300–305. IEEE, 2017.

[16] Morgado, C., Baioco, G. B., Basso, T., and Moraes, R. A security model for access control in graph-oriented databases. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 135–142. IEEE, 2018.

[17] Nayak, A., Poriya, A., and Poojary, D. Type of nosql databases and its comparison with relational databases. *International Journal of Applied Information Systems*, 5(4):16–19, 2013.

[18] Shalabi, Y. and Gudes, E. Cryptographically enforced role-based access control for nosql distributed databases. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 3–19. Springer, 2017.

[19] Uzun, E., Parlato, G., Atluri, V., Ferrara, A. L., Vaidya, J., Sural, S., and Lorenzi, D. Preventing unauthorized data flows. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 41–62. Springer, 2017.