Comparison of RBAC Implementation for Open Source Databases

by

Sagarika Deb Burman

A Project Report Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Supervised by

Dr. Rajendra K. Raj

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences Rochester Institute of Technology Rochester, New York

August 2016
Dedication

To my parents- to whom I owe everything
Acknowledgments

I am grateful to my advisors, Dr. Rajendra K. Raj and Dr. Hans-Peter Bischof for their constant guidance and support without which this project would not have been complete.
Abstract

Comparison of RBAC Implementation for Open Source Databases

Sagarika Deb Burman

Supervising Professor: Dr. Rajendra K. Raj

With the rise in ransomware attacks, internal information leaks in the world of data management, the need for stronger security protocols had become more important than ever. One of the basic forms of security in a database management system is access control. Role-Based Access Control, also known as RBAC, is one of the more flexible, robust and popular forms of access control used in today’s data management systems. This project analyzes the RBAC model, compares the implementation of RBAC in both relational databases such as MySQL, PostgreSQL and non-relational databases such as MongoDB. I have implemented a health care management system which manages patient data for hospital personnel. The RBAC model has the roles of doctor, nurse and pharmacist where the user’s privilege to access patient information is decided by the role assigned to the user. The implementation of RBAC for each database is analyzed and its performance evaluated within its own environment as well as with respect to the other databases.
# Contents

Dedication .................................................. ii

Acknowledgments ........................................ iii

Abstract ....................................................... iv

1 Introduction ............................................... 1

2 Design ....................................................... 3

3 Implementation .......................................... 6
   3.1 MySQL ................................................. 7
   3.2 PostgreSQL ........................................... 9
   3.3 MongoDB ............................................. 12

4 Analysis .................................................... 15

5 Conclusions ............................................... 19
   5.1 Current Status ....................................... 19
   5.2 Future Work ......................................... 20
   5.3 Lessons Learned .................................... 21

Bibliography ............................................... 22
List of Figures

2.1 Roles and permissions in RBAC ............................................. 3
2.2 Tables showing role-specific attribute information ................... 4
3.1 Creating role ‘Doctor’ in Postgres using pgAdmin III ................. 10
3.2 Translation from RDBMS to NoSQL ....................................... 12
Chapter 1

Introduction

All organizations need information to perform operations, processes and functions. This information may be about the company directly or information that it needs from other organizations to perform their own functions and it needs to be stored in databases, sent over networks—all of which is distributed virtually and sometimes, geographically [2]. As the authors of [2] have explained, the need for access control has risen to protect the confidentiality and integrity of information. One of the key methods to maintain the confidentiality of information is to have access control as a part of the security administration in an organization. The most commonly followed methods of access control in civilian and defence organizations were Mandatory Access Control (MAC) and Discretionary Access Control (DAC) which were being replaced by younger and more flexible methods such as RBAC and Attribute-Based Access Control (ABAC).

In this project, I look deeper into RBAC which, given its versatile and flexible nature, has become a popular choice of access control in many organizations especially in large ones where more and more users are added and removed [1, 3]. The basic model of RBAC is analyzed, that is, role assignment (to a user), role authorization and transaction authorization which occurs when a user assigned a role has the authorization to perform the operation. This model is implemented in each of the databases—MySQL, PostgreSQL and MongoDB. The implementation varies with the manual design and implementation in a database (for example, MySQL) and the same as a built-in feature (for example, PostgreSQL). The differences between the two are compared along with the difference of implementing RBAC for the hospital information management system for a relational database
such as Postgres and a non-relational one such as MongoDB. The performances of each are also evaluated with RBAC as access control feature in all the databases.

The design section discusses the design of the application and implementation of the roles for the given application in each database. This section is followed by the implementation section where the implementation of RBAC is highlighted for every database. The ‘Analysis’ section analyzes the differences among the implementations, the structures of the databases and the performances. The final section, ‘Conclusion’, discusses the inferences drawn from these analyses and the scope of the future work on this topic.
Chapter 2

Design

The analysis of the RBAC model for different database required an application that would have a lot of users who would be assigned roles based on their role within the organization. The privileges for the roles would be decided on the operations that every role is required to perform and the extent of information that he/she would need to perform the operation. Figure 2.11 shows the relationship between users and roles and the permissions and roles.

Figure 2.1: Roles and permissions in RBAC

The application that has been implemented for this project is that of a hospital information management system as described in chapter 1. A hospital is one of the organizations that has a large number of users with roles that have access to sensitive information and therefore require greater security control. The information required to be read and edited by a doctor will be very different from that of a pharmacist and even that of a nurse. A nurse’s privileges will always be a subset of the doctor’s privileges. There may be other complexities such as a nurse from one department may require special permissions to see
the patient information being treated by a doctor from a different department. In this application, the roles defined are:

- Doctor
- Nurse
- Pharmacist

The users, that is, the doctors, nurses and pharmacists will have privileges decided by the operations required to be performed by each of them. Every user will have one of the above roles of doctor, nurse or pharmacist. Information about doctors, nurses and pharmacists are stored in separate tables with role specific attribute information. The tables for the roles above are shown in Fig 2.2.

<table>
<thead>
<tr>
<th>Patient</th>
<th>Doctor</th>
<th>Nurse</th>
</tr>
</thead>
<tbody>
<tr>
<td>+PatientID</td>
<td>+DocID</td>
<td>+NurseID</td>
</tr>
<tr>
<td>+PatientName</td>
<td>+DocName</td>
<td>+NurseName</td>
</tr>
<tr>
<td>+Disease</td>
<td>+DocDept</td>
<td>+NurseDept</td>
</tr>
<tr>
<td>+Doctor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pharmacist</th>
</tr>
</thead>
<tbody>
<tr>
<td>+PharmacistID</td>
</tr>
<tr>
<td>+PharmacistName</td>
</tr>
<tr>
<td>+PharmacyID</td>
</tr>
</tbody>
</table>

Figure 2.2: Tables showing role-specific attribute information

The medication information for patients is stored in a table called ‘Medications’ where the medID is the primary key and the foreign keys are patientID, docID, pharmID. The doctor can read and write information in this table. The nurse can read the information in this table and update the pharmacy ID only, not edit the medication. The pharmacist can
only view the patient’s name and the medication in this table. No other information will be visible to the pharmacist. The medication table contains the most private information such as the illness, the doctor treating the patient, the medications. As and when a patient’s medication is added or removed, an attribute column has to be added or removed respectively. This results in altering the schema of the table which is a cumbersome process. Updating the same medication is easy but not adding/removing them. This is a drawback of using a relational database for this application which will be analyzed in details in the ‘Analysis’ section of this report. This is easier achieved in MongoDB, a NoSQL database which allows addition or deletion without causing a change in the structure of the database. This ease of scalability is a major advantage that is further discussed in the ‘Analysis’ section although the relevance of using a NoSQL database for a health care application is debated as well, later in this report.

The hardware specifications for the implementation of this project is a 64-bit Windows 8.1 operating system with an i5 Intel core processor and 8 GB RAM.
Chapter 3

Implementation

The RBAC model has three basic parts—role assignment, role authorization and transaction authorization. These are elaborated as follows [2]:

• **Role Assignment**

  A user can execute a transaction if and only if the user has been assigned a role [2]. As mentioned in [2], a user logging into a system is not considered to be a “transaction” per se. It also states that an active user at a given point of time *must* have an “active role”.

• **Role Authorization**

  A user assigned a particular role *must* be authorized for that role which means that any assignments made have to be checked that are meaningful and hence, authorized.

• **Transaction Authorization**

  The final rule is that when a user has been assigned and authorized for a role, the user can make a transaction only when the user is authorized for *that* transaction. Thus, the predicate that a transaction takes place is true if and only if a user with an active authorized role is authorized for that transaction as well.

  This transaction authorization takes care of the security concerns of a person with multiple roles assigned. For example, if a new teacher is added to a school, the teacher may be assigned the role of “teacher” in the system but the role of “trainee” as well which will limit the teacher’s access to the institute’s systems because he/she
will have restrictions placed on his “teacher” role for being a trainee. Thus, although the new person will have the authorized role of “teacher”, he/she will not be authorized for all transactions for the role “teacher”. This example is similar to the corporate setting example of trainee/supervisor in [2] for transaction authorization.

The open-source databases selected for this study are MySQL, PostgreSQL and MongoDB. While MySQL does not offer any form of built-in access control, the other two databases offer RBAC as a built-in access control feature. Postgres and MongoDB being modern databases, allow the users to the reduced work of assigning permissions to users within an organization. This section describes the implementation of RBAC in MySQL then goes on to describe the implementations in Postgres and finally, MongoDB.

### 3.1 MySQL

MySQL is one of the older, more popular open-source relational databases supported in Windows and Linux operating systems (OSes) among the popular OS environments. Since it is a popular choice for web-based applications, the hospital information management system was implemented using this as the back-end database server. as has been mentioned before, since no prior access is provided by MySQL, the RBAC implementation would mean that every user created in the system would have to be specifically be assigned privileges and given permissions using the GRANT command [10]. Chapter 2 described the privileges of each role in the application, that is, a doctor has permission to add/delete patient medical information, a nurse has viewing privilege for the medical information while the pharmacist has viewing permission for the medication the pharmacist is allowed to sell to the prescribing patient.

The greatest advantage of not having a built-in access control feature is that one has the flexibility to select what kind of access control could be implemented for the application. However, this also increases the work of creating and granting privileges to every user in the system. the drawbacks of implementing access control will be discussed in the next
section. For this project, ten patient records were created with 5 users for each role. A review of what each role can do is given below:

- Doctor: select, insert, delete, update patient’s medical information
- Nurse: select patient’s medical information
- Pharmacist: view medication of patient who has selected that pharmacy

Since there are 5 users for each role, the entire application has 15 users which results in granting privileges to every user separately. This is implemented using that ‘GRANT’ command in SQL which grants the necessary permissions to the user that have been deemed suitable based on his role. The users are created along with passwords that are verified on login. However, that does not grant them perform operations on the database but verifies if they are allowed to access the server. The CREATE USER command is followed by the GRANT command which allows permissions to perform operations on the information the user can access. An example of bestowing privilege to a user with the role ‘role’ is shown below [7]:

```
GRANT ALL ON db.* TO 'Dr. Oh'@'localhost';
```

Since Dr. Oh has the role of doctor, he is granted all privileges, that is, SELECT, INSERT, DELETE, UPDATE. Similarly, the grant statement for nurse would be:

```
GRANT SELECT ON db1.* TO 'Patty Olsen'@'localhost';
```

The user with pharmacist role has viewing privilege for the list of medication that the pharmacist will dispense. Since the medicines are attributes found in the ‘Medication’ table which contains other sensitive information such as the name of the patient, the different illnesses the patient is receiving treatment for, the doctor(s) treating the patient all of which may not be relevant for him to see. Thus, to restrict the view of the information to the pharmacist to medications the pharmacist has been instructed to dispense, we use the ‘view’ command. For the users with the role of ‘pharmacist’, the following SQL statement is used to grant viewing permission [6]:

```
GRANT SELECT ON db.* TO 'Patty Olsen'@'localhost';
```
CREATE VIEW med AS SELECT med1 FROM Medications WHERE PharmacyID='RFG78994323' AND PharmacistID='YSC657658';

In the above SQL statement, the select statement selects the column med1 from the table 'Medications’ and displays that medication to the pharmacist whose ID is YSC657658 from the pharmacy that has the above ID . The PharmacistID attribute has been intentionally used instead of the name to avoid ambiguity. The PharmacyID has been used along with the PharmacistID to make the security stronger. A pharmacist may have multiple views for one patient granted to him depending on how many medications are being dispensed by the pharmacist to that particular patient.

As it says in [7], after the user has connected to the server with the user’s credentials, the user has been “identified” and is allowed to perform the operations that have been granted to the user previously. There is a separate GRANTS table in MySQL that stores the grants and privileges granted to the users. The privileges that have been granted to the doctor, nurse and pharmacist such as inserting, deleting et cetera are stored in this GRANTS table along with the scopes of their privileges such as if the privilege is row-level or column level [7].

### 3.2 PostgreSQL

PostgreSQL (also known as Postgres) is one of the most powerful modern day RDBMSes that is offers great flexibility of policy implementations, unlimited rows per table, unlimited number of users and above all, from a security point of view, provides RBAC as a built-in form of access control. The version of Postgres used in this project is Postgres 9.0.

Since the design of the database application has been explained in section 3.1, this section will start with the overview of the access control features in Postgres and then move on to the implementation of the roles for this application in the database [4]. Postgres has the concept of ‘Groups’ and ‘Roles’ where groups can one can assign privileges to a group as opposed to assigning privileges to individual users. This is the most important difference between MySQL explained in section 3.1 and Postgres. Postgres follows the concept of
‘Roles’ where a group role defines a role that is to be shared with multiple users. This makes it convenient for the DB Admin to grant privileges to a role that many users can have instead of granting the same privileges for users with the same role on an individual basis. Figure 3.1 shows the creation of the role ‘Doctor’ where the role is created using the GUI tool for Postgres known as pgAdmin. The SQL pane at the bottom of the window shows the SQL query that can be used in psql to create the same role.

Figure 3.1: Creating role ‘Doctor’ in Postgres using pgAdmin III

Figure 3.1 shows the pgAdmin III tool that is the Graphical User Interface (GUI) that is provided for free by Postgres community and is compliant with all popular platforms such as Linux, Unix, windows. The database can be managed for Postgres using psql or pgAdmin. For the purpose of this project, pgAdmin has been used to create and manage the database. Since the roles have to be associated with a database, the database has to be created first which has been named as ‘hospital’ for this project [8]. The roles of Doctor, Nurse and Pharmacist have been created using the ‘Group Roles’ option. Once the role has been created, it is associated with the database hospital. This is done using the ‘Privileges’
option for the database. The Group Role ‘doctor’ is set as one of the roles allowed to be connected to the database.

Since the role and the database have now been connected, the privileges for the role have to be granted for performing operation on the table data in the database. This is done using the ‘Grant Wizard’ tool available for the databases in pgAdmin [8]. This tool basically allows the DB Admin to grant the CRUD options to a role. The tables that the role can update are selected under the ‘Selection’ tab. Under the ‘Privileges’ tab in the Grant Wizard, the operation are to be selected such as SELECT, UPDATE, INSERT, DELETE. For the role of doctor, all of the operations and all the tables are selected. A similar process flow is followed for the other role of Nurse with the difference being the operations and the tables selected. For the role of Nurse, the medication table is selected but the privileges granted are SELECT only. The reason why the role of ‘Pharmacist’ is not associated with the database in the same way is that the pharmacist has viewing privileges to only one column in the Medication table. To do this, the ‘Views’ tab was selected from the public schema. Once the view was created and named ‘med’, the privileges were set under the ‘Privileges’ tab. The role associated with this view was that of ‘Pharmacist’ and the operation allowed was ‘SELECT’ only. This allows the user with the role of pharmacist to view the medication that the pharmacist is allowed to dispense [8, 4].

One of the other things that should be mentioned is that Postgres also allows for ‘Login Roles’ which allow a user to be created in the system with login credentials. The reason why the roles above were not created with a password is that every user’s login information is unique. Although password option exists in Postgres for a group role creation, it is more convenient to create a role with no separate passwords but to create users with unique passwords for each user account. When using an application server to authenticate users and connect to the database server, the same credentials used for the new ‘Login Role’ should be used at the time to authenticate oneself [8].
3.3 MongoDB

The third and the final database system that was analyzed was MongoDB which is another very powerful and advanced database system that is becoming increasingly popular today. This is the only NoSQL database being analyzed in this project. Although traditionally relational databases have been considered to be most appropriate for hospital information systems, with the ease of implementation offered by the schema-free structure of NoSQL databases, a new interest has grown in using NoSQL databases for such application as well. Before I explain the implementation of RBAC in MongoDB, I’ll give a brief overview of the structure of this database since the implementation varies to quite an extent due to its difference in structure form relational databases. The version of MongoDB being used in this project is 3.2.7.

MongoDB being a schema-free database, does not have tables and rows but instead has collections and documents. This can be understood easily in the diagram shown in Figure 3.3 which is a literal translation of the tenets of an RDBMS to a NoSQL environment.

For every table in a relational database, there is a corresponding ‘collection’ in MongoDB. Entries are inserted using the command `db.save()`. Unlike the RDBMS, a primary key was not required to be designed for this model since every document in MongoDB has a unique Object ID generated for it. This would be the PatientID in the Patient collection, the DoctorID for the Doctor table, the NurseID for the Nurse table and the PharmacistID for the Pharmacist table. Although one can create Object IDs themselves, the Object IDs
used for this project were the ones generated by default.

The two broad categories of permissions are read and readWrite. ‘Read’, as the word essentially means, grants the user reading documents only. the read function gives the privilege to read all the data contained in ‘non-system’ collections [5]. The ‘readWrite’ gives the user the privileges granted by the ‘read’ permission as well as the permission to “modify” all data on “non-system” collections [5]. The most common function of the read function is find which is the equivalent of the SELECT statement in a relational database. Some of the actions that are allowed by the ‘readWrite’ privilege are:

- find
- insert
- update
- remove

The roles that were defined for the MongoDB database were the same as the ones defined for the other databases, that is, Doctor, Nurse, Pharmacist. The privileges granted for each role is as follows:

- Doctor: readWrite
- Nurse: read
- Pharmacist: read

Since the above roles are not built-in but user-defined, the required actions for each of them have to be defined with along with the database and the corresponding collection on which the actions are to be performed.
For example, for the role of ‘doctor’, the privileges have been defined as [5]:

```json
privileges: [
    actions: [“find”, “update”, “insert”, “remove”],
    actions: [“find”, “update”, “insert”, “remove”],
]
```

For the role of Nurse, the privileges are defined as ‘read’ only for the medication and patient tables. One of the issues that was experienced was the ‘view’ equivalent in MongoDB. Since there is no way to create a view for a particular attribute in MongoDB, it was difficult for a user with the role of pharmacist to view the medication object only in the ‘Medications’ collection. Although the concept of virtual collection and query filters exist in MongoDB, it was difficult to do it for a single role only. There is a way in which the query filter can be applied to create a somewhat virtual collection of the medicines list only although it is not very efficient because it requires the pharmacist to be given read privilege for the Medications collection anyway which violates the scope of the pharmacist’s privilege [9]. This has been discussed in detail in the Analysis section.
Chapter 4

Analysis

The goal of this project was to analyze the implementation of RBAC in modern open source databases. As mentioned before, while MySQL did not have any access control method as a built-in feature and Postgres and MongoDB did, there are advantages and disadvantages to both the options. This section will analyze RBAC in detail in MySQL, Postgres and MongoDB respectively and then make a qualitative comparison of all the three databases.

MySQL requires that every user created in the system be granted privileges on an individual basis to every user. This policy remains the same for any kind of access control policy whether it is RBAC or ABAC or any other type of access control method. For the RBAC model implemented for this hospital information management system, there were a total of 15 users in the system, five users each for every role. Every user had to be granted the appropriate privileges every single time. Although it does not seem very tedious for fifteen users, in a real world application where there are thousands of users, the system can become very cumbersome. Such situations can become especially complex when certain users have multiple roles and users are constantly being added and/or removed from an organization. For example, in the context of the application implemented here, a nurse may also have the role of nurse practitioner which would result in the nurse having the privileges granted to a user having the role of nurse as well as that of some of the privileges of a doctor. To do this manually for users can become a very time consuming process for a company and also a method that would ensure security but one that wouldn’t be cost effective at all.

However, the advantage of not having a built-in access control as a part of a database
is that it allows the organization to implement any kind of access control that it may want to use. Although RBAC is a very popular form of access control, many institutions use ABAC, military organizations use MAC and DAC, some use a combination of RBAC and ABAC although the latter is still in the nascent phases of development. Building a single form of access control into a database limits the flexibility of implementing any other kind of access control feature or improvising on any other.

In the context of the example of assigning multiple roles to a user, let’s take the example of a nurse and nurse practitioner. The database application implemented for this project does not have a Nurse Practitioner as one of the roles for the sake of simplicity. However, if a Nurse Practitioner was to ‘inherit’ the properties or privileges of a doctor, granting these to more than hundred users would be an extremely cumbersome task. This is where Postgres provides a very efficient feature that allows ‘membership to roles’ to inherit from other group roles by using the word ‘INHERIT’. If a role is created with one of the options set as ‘INHERIT’, then any user created in the system will have privileges granted to the user directly along with the privileges granted to any other role if it is assigned that role as well [4]. This can be explained better with an example shown below:

```
CREATE ROLE Dr. Ramoray LOGIN INHERIT CREATE ROLE ADMIN INHERIT
GRANT ADMIN TO Dr. Ramoray
```

However, if line 12 had the option set to NOINHERIT, then even if another role was granted that role, the user wouldn’t be able to inherit its privileges because the former role was set to NOINHERIT. This can be done for roles with a lot of high-level privileges where the DB Admin wouldn’t want this role’s privileges to be granted to anyone unless done so explicitly.

One of the major drawbacks that was noticed for this kind of an application when relational databases were used was that for the Medications table, whenever a medicine was added for a particular patient, the schema of the table had to be altered. This is because every medicine is stored as a separate attribute in the Medications table. Although one could argue that the medicines could be saved in the Patient table along with the patient
information, doing that could make the table sizes impractically large. Normalizing the medication information allows for a more concise and efficient database design. Moreover, the addition of the medication information would not have solved the problem of altering the schema of the table every time a medicine was added. Since the a column has to be added every time a medicine is added, the ALTER TABLE command has to be used. The SQL statement to execute this is shown below:

```
ALTER TABLE Medications
ADD COLUMN med2 varchar;
```

This results in alteration of the schema which seemed to be the only solution but clearly not a good one because changing the structure of the database so often is not good design. This is where MongoDB scores over the other two databases. Due to the schema-free structure of MongoDB, as many values for an attribute (referred to as objects in MongoDB) can be added without worrying about changing the schema of the database. Since there is no concept of table in MongoDB, the medications list for a patient is a part of the patient’s details. All information pertaining to a particular patient is included in the same document which solves the problem for medicines, the number of doctors, the list of illnesses being treated for et cetera. Whenever a patient’s information is to be referred to, it can be used by referencing, that is, by using the object ID, all other patient information can be made use of [5].

The only difficulty experienced in implementing RBAC was for the role of pharmacist. Since the SQL equivalent of ‘view’ does not exist yet exist in NoSQL, restricting ‘read’ privilege to the medicine information only in the documents. Although the find() function can be used to create a virtual collection of medications only, the pharmacist would have to be granted read permission to do this operation. Allowing another user with a different role such as doctor or nurse defies the very purpose of implementing RBAC. The DB admin or a separate role could be created to create this virtual collection and then grant read privilege to users with the role of pharmacist [9]. However, this method is a complicated and cumbersome way and no solution to this problem could be figured out during the
implementation of this project. This is something that is going to be discussed further in the Future Work section.

Another point worth mentioning is that although MongoDB implements RBAC by default, it limits the flexibility of the organization to implement any other form of access control. This is a trait that is shared between Postgres and MongoDB and the part where MySQL outperforms them.

The final observation in the analysis is that there have been documentation that Postgres performs slower than other databases equivalent to its database capacity when the records cross over a million. Since having more than million records was not a feasible option for the scope of this project, it could not be verified. However when the number of records were scaled from twenty to eighty, no significant reduction in efficiency was noticed. This could be attributed to the fact that for such powerful databases, few hundred records do not make significant difference in performance. The timestamps used to measure performance were the same for both twenty and eighty records.
Chapter 5

Conclusions

The implementation and analysis of RBAC for this project highlighted several important factors about RBAC and its efficiency given a certain kind of application and a particular business organization. For an organization where a there is a dynamic personnel environment, RBAC is one of the best solution from a technical and business perspective [2]. This is because roles form the foundation of RBAC. In a large organization, users are constantly being added and/or removed. Also, the definition of roles may change without prior notice and may have to be implemented immediately. In this kind of a scenario, making modifications to each user in the company would be a very cumbersome task. However, the number of roles to the number of people would be very low ratio always [1, 2, 3]. Thus, implementing RBAC is a cost-effective solution for such organizations. This also explains why most modern databases are building in RBAC as the access control method.

5.1 Current Status

The study has implemented and analyzed three open source database systems that have different features such as relational versus NoSQL, RBAC as a built-in method of access control versus no built-in access control method. The ease of implementation was measured along with the scope of the security, privileges and flexibility of RBAC. It was found that while MySQL provided the flexibility of choosing the kind of access control to the organization, it made the process of assigning privileges very cumbersome. In an organization with a dynamic personnel system, this would be potential problem.
This could be solved by using more modern and powerful databases such as Postgres and MongoDB which offer RBAC as a security measure as a built-in feature. Although this takes away the flexibility of choice of access control method, it saves up on time and cost which overshadows the flexibility factor. Postgres has a very easy and effective method of implementing RBAC especially with the use of the pgAdmin GUI tool. MongoDB being a NoSQL database has a totally different data model which is very effective for an application that has new attributes added on a regular basis such as medicines for a patient. The schema-free structure of this database allows immense scope of expansion and contraction (when information such as medicine, doctor etc. are removed). However, the omission of ‘view’ creates problems for roles that have very restrictive access to information such as the role of Pharmacist.

5.2 Future Work

There is a lot of scope in extending this work in the future such as defining a framework that combines the best of RBAC and ABAc and analyzing if it can be implemented in MySQL. Since Postgres and MongoDB are open source databases, efforts can be made to include this framework in their environments. A NoSQL database offering a combination of both would have immense security potential in today’s world.

Since only one NoSQL database was analyzed, implementing RBAC in another NoSQL database such as Neo4j would make the analysis stronger especially because not only is Neo4j a NoSQL database but its graph structure makes it a very unique database system.

The lack of a ‘view’ option could be further researched and methods could be studied and implemented so that even NoSQL databases such as MongoDB can have the NoSQL equivalent of the SQL view.

The application has been implemented using pgAdmin III in this project. The same application could be implemented using pl/sql and a comparison could then be made to see if there is a difference in the performance.
5.3 Lessons Learned

One of the lessons learnt while doing this analysis was that although implementing RBAC has been made very easy by the inclusion of RBAC as a built-in feature in databases, it would not be as simple in a real world where there would be more number of roles, multiple roles assigned to a user. The question of scalability could potentially make the implementation of access control more complex. The most important lesson learnt is that every database has its share of advantage and disadvantages. It is quite often up to the database administrators to make use of the strengths of the database to implement a robust application. Sometimes it may be a matter of trade-off too where the decision has to be made keeping in mind all the technical as well business aspects of an organization.
Bibliography

[1] Surajit Chaudhuri, Raghav Kaushik, and Ravi Ramamurthy. Database access control privacy: Is there a common ground?


