Fachhochschule Hannover
Universidad Politécnica de Madrid

MASTER THESIS

Advanced Policy-based Access Control in RDBMS

Author:
Francisco Javier Revilla Linares

Examiners:
Prof. Dr. Carsten Kleiner
Prof. Dr. techn. Wolfgang Nejdl

Supervisors:
Dr. Daniel Olmedilla
M.Sc. Dipl.-Inf. (FH) Sergej Zerr

November 12, 2008
Participants

First examiner
Prof. Dr. Carsten Kleiner
Fachhochschule Hannover, Fakultät IV,
Ricklinger Stadtweg 120, 30459 Hannover, Germany
E-Mail: carsten.kleiner@fh-hannover.de

Second examiner
Prof. Dr. techn. Wolfgang Nejdl
L3S Research Center & Hannover University,
Appelstr. 9a, 30167 Hannover, Germany
E-Mail: nejdl@l3s.de

Supervisors
Dr. Daniel Olmedilla
L3S Research Center & Hannover University,
Appelstr. 9a, 30167 Hannover, Germany
E-Mail: olmedilla@l3s.de

M.Sc. Dipl.-Inf. (FH) Sergej Zerr
L3S Research Center & Hannover University,
Appelstr. 9a, 30167 Hannover, Germany
E-Mail: zerr@l3s.de

Author
Francisco Javier Revilla Linares
Universidad Politécnica de Madrid & Fachhochschule Hannover
C/Cueva de la Mora 7, vivieda 2,
28670 Villaviciosa de Odón (Madrid), Spain
E-Mail: franciscojavier.revilla@gmail.com
Acknowledgements

I would like to thank Daniel Olmedilla, Sergej Zerr and Carsten Kleiner for all their advice and guidance and all that I could learn from them during the six month I have spent working at my master thesis. Thanks again to Daniel Olmedilla for giving me the opportunity to write the thesis in collaboration with L3S Research Center.
Thesis Declaration

I hereby certify that this thesis is my own and original work using the sources and methods stated therein.

Selbständigkeitserkärung

Ich versichere, dass ich diese Diplomarbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe.

Declaración Jurada

Por la presente declaro que esta propuesta es mi propio trabajo y hasta donde yo sé y creo, no contiene material previamente publicado o escrito por otra persona, ni material que de manera substancial haya sido aceptado para el otorgamiento de premios de cualquier otro grado o diploma de la universidad u otro instituto de enseñanza superior, excepto donde se ha hecho reconocimiento debido del texto.

Hannover, November 12th, 2008.

..........................................
Francisco Javier Revilla Linares
## Contents

1 Introduction 9

2 State of the Art in Databases Access Control 13
   2.1 Access Control Models 14
      2.1.1 Classical Access Control Models 14
      2.1.2 Credential-Based Access Control 17
      2.1.3 Access Control through Encryption 18
   2.2 Vendors Implementations 19
      2.2.1 MySQL 20
      2.2.2 Informix 23
      2.2.3 Oracle 31
      2.2.4 Summary 34
   2.3 Research Approaches 35

3 Problem Solution: Concept 41
   3.1 High level rules introduction 42
   3.2 High level rules evaluation 46
   3.3 Policy Based Query Expansion 68

4 Problem Solution: Implementation 97
   4.1 Architecture 98
   4.2 Results 106
   4.3 Changing policies and queries 119

5 Problem Solution: Experiments and Evaluation 121
   5.1 Introduction 122
   5.2 Experiments 122
   5.3 Adding indexes to the column tables 126
   5.4 Conclusions 127
   5.5 Algorithm times 128

6 Conclusions and Future Work 129

Bibliography 133
Chapter 1

Introduction
A database is a structured collection of data and computer databases relies in the software to organize the storage of data, this software organized, structured and used the data following a database model. Nowadays the most common used is the relational model. Other model such as the hierarchical model or the network model are also common. The software used to organize and maintain the databases are called Database Management System (DBMS) and not only deal with the data model also with performance, concurrency, integrity, and recovering from hardware failures.

A relational database is a database that conforms to the relational model, and refers to a database’s data and schema (the database’s structure of how those data are arranged). The term Relational database management system technically refers to the software used to organize and maintain a relational database. Relational databases have become the first choice for the storage of the tabular information that supports the world economy, personnel data and much more. Relational databases replaced hierarchical databases, network databases and have survived challenges from more recent object databases and XML databases.

A relational database management system (RDBMS) is a software that lets you create, update, and administer a relational database. Most commercial RDBMS’s use the Structured Query Language (SQL) to access the database, although SQL was invented after the development of the relational model and is not necessary for its use.

Access control is the ability to permit or deny the use of a particular resource by a particular entity. Nowadays most used access control techniques in RDBMS are based on users or roles with privileges on the databases stored objects. Each object could have a security level assigned and if the user or role owns that security level then he will be able to access (read or write) the data contained in the object. These mechanisms fit very well in small or/and close environments like companies, and its effectiveness has already been proven in today’s vendors implementations.

However, these current database access control mechanisms are very restricted, in the best case, it is possible to specify access control to tables or columns to users or roles. That works very well in closed environments such as within an organization or a company. Again in an open environment this access control models are not very expressive, imagine the database administrator wants to allow data access to some important information only from 08:00 to 12:00 during working hours, but not out of this period time. That, is non data dependant information, it is context dependant and it is not currently possible with available technology offered by the relational database management systems. Another possible situation, let imagine the access to some object should be only allowed when his status (and his status info could be stored in a different object such a special table) is public. Thus the access is dependant on the objects data information, which is relatively possible nowadays in some vendors implementation, however the expressiveness is not really powerful enough.

To understand the problem better let us consider an scenario. Let say that Alice is the relational database administrator of the Health Government Agency, where all information concerning employees like patients, physicians, chemists, nurses and others employees are managed, as well as information concerning to researches projects, subdepartments or hospitals where employees work.

Such database should be accessed by each user that works in the Department of Health, for instance, a nurse wants to know where is a patient is located, or a physician wants to read the patient history, as well as other employees. The database may be also accessed by third
parties, for instance, the Economy Agency may want to obtain some information about salaries average, which such information should be only allow to restricted personnel, or the Statistic Agency wants to calculate some diseases statistics data from Alices database. Another example raising the complexity could be that a relative from a patient who is in a hospital wants to check before visiting the patient where he is located, that is, the room and floor in the hospital, as well as hospital address. In that case the relative will go direct to his target without asking to any personnel. In case the relative wants to know the estimate date when the patient is going to be discharged from the hospital may also access the database and read the date. This huge scenario overload the user management and Alice should be able to allow somehow this type of access to third parties, unknown users and user within the Department of Health.

The scenario is not only contemplating users management, now let us assume some access constraints for instance to the patients data. Patients location information such as hospital, room, or floor should be available to all kind of users, on the other hand, critical patient data such as illness should be only available to his doctor/s and the doctors who treat related problems, for example, a patient with a heart problem, his information should be available to the patients doctor and for all the cardiologist. And of course the nurses and physicians may access the drugs treatment for the patients.

Other constraints are extracted from the following examples. Alice suspects that the IT-Department employees forget to close some connections to the database system when they leave the work, and these connections decrease the performance of the system. Thus a good idea is to close IT-Department connection when they are not at work, a possibility is to allow only the access to the system between 08:00 and 17:00 to IT-Department members. Another example; chemists researchers asked Alice if it would be possible to allow access to documents that belongs to different projects among chemists, as long as the document status is public but never if document status is private.

Now if we analyze the Alice situation, she will have to face a lot of situations that technology from nowadays in relational database system does not solve. Such open and dynamic scenario where new users are added, unknown users can connect, or old user are deleted, a scenario which is always growing and the user management and constraint access made it from the administration point of view a chaotic situation. The size and dynamics problems of the user community in this scenario can not be easily solved by traditional access control models that RDBMS offer. It is often impractical to assume the creation and management of one account for each and every user on each system: it is complex on the providers side as well as on the client side. A partial solution to user account management are credential-based systems, but these solutions need a third party interaction such as creation of certificates, and from the point of view of the user the usability decreases.

After an analyze, Alice realizes that the current relational database system can not afford all requisites above described. At first the system does not allow to give access to unknown users, all users must be register in her database system, she will need an anonymous user in her database with a restricted access control conditions. She has a vast number of users and roles to assign, and to manage this high number of users with the corresponding privilege assignment to objects in the database will be a hard task, for instance, how to manage the physicians. Should she for each new physician create a user, and for the specialist physicians as cardiologist specialist create a new role and assign to the user the possibility to own such role? Maybe a good idea should be to present somekind of credential to the system that proofs that the requester user is a physician, or even more clever, the specialist physician presenting a
CHAPTER 1. INTRODUCTION

Cardiologist credential will be also available to access the same data as a normal physician and of course the data available to the cardiologist, that could be resolved including a role hierarchy and/or using a public key infrastructure, but that means to add an external mechanism to the system, with the corresponding load but will fit very well in the case above explained when the Statistic or the Economy Agencies want to access the data from the database.

Even more difficult tasks to Alice are to add the described constraints to data access, it is not possible to add time conditions to restrict the access. If Alice wants to protect the access to some object based on the time like a period of time on each day, or depending if the current day is an even or an odd day. There is no possibility to allow the access to the database depending on the current time or date. Therefore, relational databases systems does not support any kind of mechanism to allow/deny access based on contextual conditions such as time.

About the content dependant constraints, Alices should allow access to the patient data to the physicians depending on the disease type or if the physician is the usual patient doctor. In order to allow the cardiologist access to the patient, the disease type should be at first checked probably in other table such us the disease table. Other example is the documents to share among projects, they should be able to share if the document status is public, in other case such document must be private to the project members. Alice can not determines rules to based the access to database objects based on the same or other object data information, all she can do is to create views, one view for each different constraint, and allowing the access to the different views according to the constraints. This could be an initial solution when there are not too many constraints and users, however when there are a lot of users and constraints or the constraints depend on other object information, the view mechanism is not any more useful.

The goal of this thesis is to solve current requirements in database access control. The idea is to enhance this access control mechanism by specifying a set of more expressive access control policies in a way that (at least partially) can be enforced at the database level increasing expressiveness power, in order to represent more complex and sophisticated scenarios of the real world such as described above, where new features like context, credential and a more expressive content dependant conditions are able to use in the access control mechanism of the relational database management systems.
Chapter 2

State of the Art in Databases Access Control
Access control is the process of mediating every request to resources and data maintained by a system and determining whether the request should be granted or denied. Traditional access control models and languages result limiting for emerging scenarios, whose open and dynamic nature requires the development of new ways of enforcing access control. Access control is then evolving with the complex open environments that it supports, where the decision to grant an access may depend on the properties (attributes) of the requester rather than her identity and where the access control restrictions to be enforced may come from different authorities. These issues pose several new challenges to the design and implementation of access control systems. In this chapter, the emerging trends in the access control eld to address the new needs and desiderata of todays systems are presented as well as some vendor implementation.

2.1 Access Control Models

2.1.1 Classical Access Control Models

Classical access control models can be grouped into three main classes: discretionary access control (DAC), which bases access decisions on users identity; mandatory access control (MAC), which bases access decisions on mandated regulations dened by a central authority; and role-based access control (RBAC), which bases access decisions on the roles played by users in the models.

Discretionary Access Control

Discretionary access control is based on the identity of the user requesting access and on a set of rules, called authorizations, these authorization rules describes which use can perform which action on which resource from the database. Let describe the authorization as a triple (s, o ,a) where a user s can execute an action a on object o.

In the most basic form, an authorization is a triple (s, o, a), stating that user s can execute action a on object o. Let S, O, and A be a set of subjects, objects, and actions, respectively. The access matrix model represents the set of authorizations through a \( |S| \times |O| \) matrix A. Each entry A[s, o] contains the list of actions that subject s can execute over object o. Table 3.1 illustrates an example of access matrix where, for example, user Alice can read and write Document1.

<table>
<thead>
<tr>
<th>User</th>
<th>Document1</th>
<th>Document2</th>
<th>Software1</th>
<th>Software2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>read, write</td>
<td>read</td>
<td>execute</td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td>read</td>
<td>read</td>
<td>read</td>
<td></td>
</tr>
<tr>
<td>John</td>
<td>read, write</td>
<td>read, execute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Laura</td>
<td>read, write</td>
<td>read, write, execute</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Privileges associated with each SQL object

Some disadvantages of the discretionary access control model are the vulnerability in high security settings to Trojan horses and the difficult administration in case of a environment with a high number of users. The reason for the Trojan horse vulnerability is that discretionary access control does no distinguish between users and subjects. It is then vulnerable from processes executing malicious programs that exploit the authorizations of the user invoking them.
2.1. ACCESS CONTROL MODELS

Mandatory Access Control

Mandatory security policies enforce access control on the basis of regulations mandated by a central authority. The most common form of mandatory policy is the multilevel security policy, based on the classifications of subjects and objects in the system. Each subject and object in the system is associated with an access class, usually composed of a security level and a set of categories. Security levels in the system are characterized by a total order relation, while categories form an unordered set used to give more granularity to the access control system. As a consequence, the set of access classes is characterized by a partial order relation, denoted \( \geq \) and called dominance. Given two access classes \( c_1 \) and \( c_2 \), \( c_1 \) dominates \( c_2 \), denoted \( c_1 \geq c_2 \), if the security level of \( c_1 \) is greater than or equal to the security level of \( c_2 \) and the set of categories of \( c_1 \) includes the set of categories of \( c_2 \). Access classes together with their dominance relationship form a lattice.

Mandatory policies can be classified as secrecy-based and integrity-based, operating in a dual manner.

Secrecy-Based Mandatory Policy \([9,13]\). The main goal of secrecy-based mandatory policies is to protect data confidentiality. The security level of the access class associated with an object defined the sensitivity of its content, the security level of the access class associated with a subject, reflects the degree of trust placed in the subject not to reveal sensitive information, that is called clearance. A user can connect to the system using any access class dominated by her clearance. A process generated by a user connected with a specific access class has the same access class as the user.

The access requests asked by a subject are evaluated by applying the following two principles;

- **No-Read-Up.** A subject \( s \) can read an object \( o \) if and only if the access class of the subject dominates the access class of the object.
- **No-Write-Down.** A subject \( s \) can write an object \( o \) if and only if the access class of the object dominates the access class of the subject.

The principles of the secrecy-based mandatory policy is to prevent information flows from high level subjects/objects to subjects/objects at lower (or incomparable) levels. However, these two principles may seems to be too restrictive. For instance, in a real scenario data may need to be downgraded. Considering these situations, the secrecy-based mandatory models can allow exceptions for processes that are trusted.

Integrity-Based Mandatory Policy \([4,9]\). The goal of integrity-based mandatory policies is to prevent subjects from indirectly modifying information that they cannot write. The integrity level associated with a user reflects then the degree of trust placed in the subject to insert and modify sensitive information. The integrity level associated with an object reflects the degree of trust placed on the information stored in the object and the potential damage that could result from unauthorized modifications of the information.

The access requests asked by a subject are evaluated applying the following two principles;

- **No-Read-Down.** A subject \( s \) can read an object \( o \) if and only if the integrity class of the object dominates the integrity class of the subject.
- **No-Write-Up.** A subject \( s \) can write an object \( o \) if and only if the integrity class of the subject dominates the integrity class of the object.
The integrity model prevents flows of information from low level objects to higher objects.

Note that secrecy-based and integrity-based models are not exclusive, it is usual to protect both the confidentiality and the integrity properties. Obviously, in this case, objects and subjects will be associated with both a security and an integrity class.

Let’s consider the following example of a security lattice in figure 2.1(a), where there are two security levels, Secret (S) and Unclassified (U) with S > U, and a set of categories \{Admin, Medical\}. Also considering the integrity example in the figure 2.1(b), where there are two security levels Crucial (C) and Important (I), where C > I, and the set of categories \{Admin, Medical\}.

From the secrecy-based point of view suppose in figure 2.1(a), lets say that user Alice has the clearance (S, \{Admin\}), and she connects to the system only with the clearance (S, \{\}). She will be able to read objects (S, \{\}) and (U, \{\}), because they are the only objects that her clearance dominates. And she will be able to write in (S, \{\}), (S, \{Admin\}), (S, \{Medical\}), and (S, \{Admin, Medical\}) access classes, because these are the object which objects clearance dominates Alices clearance.

From the integrity-based point of view, lets suppose that user Alice connects to the system with (C, Admin) clearance. She will be able to write objects with integrity class (C, \{Admin\}), (C, \{\}), (I, \{Admin\}) and (I, \{\}), because they are the only objects that her clearance dominates. In the same way, she will be able to read objects with integrity class (C, \{Admin\}) and (C,\{Admin, Medical\}), because these are the objects which object clearance dominate the user Alices clearance.

### Role-based Access Control

A third approach for access control is represented by Role-Based Access Control (RBAC) model [11]. A role is a set of privileges that a user can play, such user is associated with the role privileges. When accessing the system, each user has to specify the role he is going to play and, if he is granted to play that role, he can use the corresponding privileges. The access control policy is then defined through two different steps: first the administrator defines roles and the privileges related to each of them; second, each user is assigned with the set of roles he can play. Roles can be hierarchically organized in order to propagate access control privileges through the hierarchy. A user is allowed to play more than one role and more users are also
2.1. ACCESS CONTROL MODELS

available to play the same role at the same time.

It is important to know that roles and groups of users are two different concepts. A role is a collection of privileges, and a group is a collection of users. Furthermore, roles can be activated and deactivated by users but the membership in a group cannot be deactivated.

The main advantage of RBAC, respect to DAC and MAC, is that it better suits to commercial and close environments. Because in a company or a close environment, it is not important the identity of a person for his access to the system, however his responsibilities are important.

2.1.2 Credential-Based Access Control

In an open and dynamic scenario, parties may be unknown to each other and the traditional separation between authentication and access control cannot be applied anymore. Because in this scenario it is necessary to know which clients are allow to access the information as well as which servers are allowed to supply the data, that is the user wants to know if he can rely in the source who provide the information and the server wants to know if the user is allowed to access the information stored. Access control solutions should decide on the one hand which client is qualified to access the information, and, on the other hand, which server is qualified for providing the information. Trust management has been developed as a solution for supporting access control in open environments. Trust management models are possible thanks to the introduction of digital certificates. A digital certificate is an electronic paper credential like a driving license. A digital certificate is a digital document, certificated by a authority (trusted entity), that declare a set of properties to the certificate owner, as well as a public key.

Access control models, using digital certicates for granting or denying access to resources, make access decisions based of a set of properties that the requester should have. The final user can prove to have such properties by providing one or more digital certicates. Now we have that credential based access is not only useful for authenticating requester in order to access the information, it is also use to enhance access control restrictions based in the credentials of both side parties, when a requester/server does not provide the demanded properties the access will be denied.

As it was said before since parties are unknown to each other, the requester will not know about the necessary credentials for gaining access privileges. So during access control process the two parties must exchange information about the credentials needed to access. The access control system decision comes after a complex negotiation process, where parties exchange information not only related about itself also they exchange information related to restrictions imposed by the other side. This process is called trust negotiation, and the goal is to establish trust negotiation strategies between the parties in an automatic way. The characteristics of the principal trust negotiation strategies are the following:

1. The client first requests to access a resource.

2. The server then checks if the client provided the necessary credentials. In case of a positive answer, the server grants access to the resource; otherwise it communicates the client the policies that he has to fulfill.

3. The client selects the requested credentials, if possible, and sends them to the server.

4. If the credentials satisfy the request, the client is granted access to the resource.
2.1.3 Access Control through Encryption

Since amount of data organizations need to manage is increasing very quickly, data outsourcing is becoming more and more attractive to companies. Data outsourcing provides data storage allowing the data owner to concentrate only in the core business where data are managed by an external provider. Final clients are supposed to rely in how the data is stored in the server and how query’s are executed in it, but the main disadvantage is that service providers could not be fully trusted for accessing data information. To solve this problem solutions based on cryptography have been proposed to protect data privacy. The idea is to encrypt the data in the server side, and the problem is that server assume that any client has complete access to the data. In order to solve that, they should filter the out data to make it not accessible to all clients, but this will cause an excessive load to the owner, making the data outsourcing a slow process and making it not a good choice for the companies, because the disadvantages will be higher than the advantages. On the other hand, from the point of view of the data owners the remote server should not know the control access policies, so the service provider cannot implement the access control policy. In this situation neither the data owner nor the remote server can enforce the access control policy, for security and efficiency reasons, the data should implement themselves the selective access. This technique can be implemented with selective encryption, which consist in encrypt data using different keys and distributing the keys, in that way the users can decrypt only the data that they are authorized.

Let consider a system composed of a set $U$ of users and a set $R$ of resources. A resource may be a table, an attribute, a tuple, or even a cell, depending on the granularity of the policy. The access control policy defined by the data owner can be easily represented through a traditional access matrix $A$, where each cell $A[u,r]$ assume the value 1, if $u$ can access $r$, or the value 0, otherwise (currently only read privileges have been considered). Figure 2.2 represents an example of access matrix, where there are four users, namely A, B, C, and D, and four resources $r_1$, $r_2$, $r_3$, and $r_4$.

<table>
<thead>
<tr>
<th></th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>$r_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.2: An example of binary access matrix

The solution consist on using a different key for each resource, and communicating only the set of keys which protect the resources that are available to each user. This solution requires that each user should know a great number of secret keys. To solve the keys management in the user side key derivation methods can be adopted. A key derivation method allows the computation of an encryption key, by proving the knowledge of another secret key in the system.
2.2 Vendors Implementations

In order to have a better idea how the vendors implementation work, an example of an imaginary database is described here, the idea is to use this database along the document examples with the use of some queries over this database as well as explanations to understand better the concepts.

The medical database here presented has the patient table, which contains all the patient information, such as name, age, address, telephone and the disease name that they suffer. Disease table content information about a disease, each disease has an id, disease name and the danger level of such disease. The room table content information about each room in the sanitary system, that is, the room id, the number, the floor where it is situated, the patient that is living in such room, and the hospital where the room is located.

Table 2.3: patient table

<table>
<thead>
<tr>
<th>id1</th>
<th>name</th>
<th>age</th>
<th>address</th>
<th>telephone</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>21</td>
<td>first Av.</td>
<td>1111</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Luis</td>
<td>32</td>
<td>second Av.</td>
<td>2222</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Peter</td>
<td>13</td>
<td>third Av.</td>
<td>3333</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Sara</td>
<td>54</td>
<td>fourth Av.</td>
<td>4444</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Paul</td>
<td>65</td>
<td>fifth Av.</td>
<td>5555</td>
<td>5</td>
</tr>
</tbody>
</table>

(a) disease table

<table>
<thead>
<tr>
<th>id2</th>
<th>name</th>
<th>dangerousness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>allergy</td>
<td>high</td>
</tr>
<tr>
<td>2</td>
<td>cancer</td>
<td>very high</td>
</tr>
<tr>
<td>3</td>
<td>chickenpox</td>
<td>low</td>
</tr>
<tr>
<td>4</td>
<td>heart attack</td>
<td>very high</td>
</tr>
<tr>
<td>5</td>
<td>flu</td>
<td>normal</td>
</tr>
<tr>
<td>6</td>
<td>fever</td>
<td>very low</td>
</tr>
</tbody>
</table>

(b) room table

Table 2.4: result table

<table>
<thead>
<tr>
<th>name</th>
<th>floor</th>
<th>disease name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 John</td>
<td>first</td>
<td>fever</td>
</tr>
<tr>
<td>2 Luis</td>
<td>second</td>
<td>cancer</td>
</tr>
<tr>
<td>3 Peter</td>
<td>second</td>
<td>chickenpox</td>
</tr>
<tr>
<td>4 Sara</td>
<td>fourth</td>
<td>heart attack</td>
</tr>
<tr>
<td>5 Paul</td>
<td>fourth</td>
<td>flu</td>
</tr>
</tbody>
</table>

For instance, if the following query is executed in the medical database:

SELECT p.name, r.floor, d.name
FROM patient p, room r, disease d
WHERE p.id1 = r.patient AND p.disease = d.id3

the following result will be obtain:

Table 2.4: result table
2.2.1 MySQL

MySQL is popular for web applications and acts as the database component of the LAMP, BAMP, MAMP, and WAMP platforms. This added to an easy installation and good online documentation made MySQL one of the most used and well-known RDBMS. Nevertheless its access control model is based on discretionary access control (DAC) and does not implement any other model, the name given by MySQL authors to their DAC implementation is MySQL Access Privilege System.

The MySQL privilege system ensures that all users may perform only the operations allowed to them. As a user, when you connect to a MySQL server, your identity is determined by the host from which you connect and the username you specify. When you issue requests after connecting, the system grants privileges according to your identity and what you want to do. Therefore MySQL considers both hostname and username to identifying.

The server stores privilege information in the grant tables of the mysql database (that is, in the database named mysql). The MySQL server reads the contents of these tables into memory when it starts and re-reads them under some others circumstances. Access-control decisions are based on the in-memory copies of the grant tables. Here a short description of some the tables used to perform the access control is introduce.

<table>
<thead>
<tr>
<th>user</th>
<th>db</th>
<th>tables_priv</th>
<th>columns_priv</th>
<th>procs_priv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Host</td>
<td>Host</td>
<td>Host</td>
<td>Host</td>
</tr>
<tr>
<td>User</td>
<td>Db</td>
<td>Db</td>
<td>Db</td>
<td>Db</td>
</tr>
<tr>
<td>Password</td>
<td>User</td>
<td>User</td>
<td>User</td>
<td>User</td>
</tr>
<tr>
<td></td>
<td>Table_name</td>
<td>Table_name</td>
<td>Column_name</td>
<td>Runtime_name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Column_priv</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

There is one table more called host, which is similar to the db table but with an specialized use [15]. The user and db table have also a possible collection of privilege like Select, Update or Delete privilege, all of them are describe in MySQL documentation [15].

Each grant table contains the context in which the row applies, for instance, the tables_priv table contain the Host, Db, User and Table_name, that is, the user connected from a host, using database and accessing a table in such database will have the privileges describe by the columns table_privilege and column_privilege to such table. Therefore the server combines the information in the various grant tables to form a complete description of a user’s privileges.

Briefly, the server uses the grant tables in the following manner:

- The user table scope columns determine whether to reject or allow incoming connections. For allowed connections, any privileges granted in the user table indicate the user’s global (superuser) privileges. Any privilege granted in this table applies to all databases on the server.

- The db table scope columns determine which users can access which databases from which hosts. The privilege columns determine which operations are allowed. A privilege granted at the database level applies to the database and to all its tables.
• The host table is used in conjunction with the db table when it is wanted to give a user the access to a database from several hosts. For example, if a user should be able to use a database from several hosts, the Host value should be empty in the user’s db table row, then populate the host table with a row for each of those hosts.

• The tables_priv and columns_priv tables are similar to the db table, but are more fine-grained: They apply at the table and column levels rather than at the database level. A privilege granted at the table level applies to the table and to all its columns. A privilege granted at the column level applies only to a specific column.

• The procs_priv table applies to stored routines. A privilege granted at the routine level applies only to a single routine.

The way MySQL Access Privilege System works is the following:

1. Identity checking is performed using the three user table scope columns (Host, User, and Password). The server accepts the connection only if the Host and User columns in some user table row match the client hostname and username and the client supplies the password specified in that row.

2. The server looks in the db table for a match on the Host, Db, and User columns. The Host and User columns are matched to the connecting user’s hostname and MySQL username. The Db column is matched to the database that the user wants to access. If there is no row for the Host and User, access is denied.

3. If there is a matching db table row and its Host column is not blank, that row defines the user’s database-specific privileges.

4. If the matching db table row’s Host column is blank, it signifies that the host table enumerates which hosts should be allowed access to the database. In this case, a further lookup is done in the host table to find a match on the Host and Db columns. If no host table row matches, access is denied. If there is a match, the user’s database-specific privileges are computed as the intersection (not the union!) of the privileges in the db and host table entries; that is, the privileges that are ‘Y’ in both entries. (This way you can grant general privileges in the db table row and then selectively restrict them on a host-by-host basis using the host table entries.)

5. After determining the database-specific privileges granted by the db and host table entries, the server adds them to the global privileges granted by the user table. If the result allows the requested operation, access is granted. Otherwise, the server successively checks the user’s table and column privileges in the tables_priv and columns_priv tables, adds those to the user’s privileges, and allows or denies access based on the result. For stored routine operations, the server uses the procs_priv table rather than tables_priv and columns_priv.

Example

Now let explain how it work with the medical database example, lets say user Bob is a nurse that works in a hospital and try to connect from hospital.gov and execute a query. Bob as a nurse is allowed to execute select statement in tables room and disease, as well as select in columns patient.id1, patient.name and patient.age. The query that Bob executes is the same as in the medical database example:
SELECT p.name, r.floor, d.name
FROM patient p, room r, disease d
WHERE p.id1 = r.patient AND p.disease = d.id3

1. The server checks if user Bob from host hospital.gov and his password match any row in the user grant table. In that case global privileges from user grant table are added to user Bob permissions set, but Bob has no global privileges and right now cannot access any data.

2. The server checks if user Bob from host hospital.gov is allowed to connect to the medical database looking at db and host grant tables. User Bob has no specific database privilege in the medical database, therefore any privilege is added to user Bob privileges.

3. The server must checks if Bob has privilege in the three tables he want to access with the Select statement, and Bob can only perform Select statement in disease and room tables, but not in the patient table. The server should continue checking at column level.

4. The server checks if user Bob can perform select statement on columns id1, disease and name from patient table. Bob is not allowed to access the disease column from the patient table, therefore there will be a empty value as a result.

Analysis

MySQL granularity is restricted to objects, it is possible to protects tables, views but not specific rows or columns. The smallerst object that is possible to protect is a column inside a table using views. It could be possible to protect a subgroup of rows within a table creating a view with conditions. For instance if in the patient table is necessary to protect the patients that are younger than eighteen years old, then a new view with such condition should be created. In the same way a low content dependant access control with views could be implemented.

MySQL also does not support context dependant protection. Another problem of MySQL is that does not implement role-based access control and the user management in a big user community is impractical.
2.2. VENDORS IMPLEMENTATIONS

2.2.2 Informix

Informix is a family of relational database management system (RDBMS) products by IBM. It is positioned as IBM’s flagship data server for online transaction processing (OLTP) as well as integrated solutions.

Informix access control model implements discretionary, mandatory and role-based access control models.

Role-Based Access Control

The Informix role-based access control is the primary control mechanism that enables access to SQL objects using privileges and roles. The objects that can be protected are databases, tables, columns, views, routines and languages. By protecting and granting privileges to access these objects to authorized roles is the way that Informix implement role-based access control.

A role should be created with specific responsibilities, and the privileges granted to certain role should reflect its responsibilities. After creating roles the dba can grant to users these roles. A user can assume a role depending on the task he wants to perform in the database.

Role-based access control extends discretionary access control by grouping a set of privileges into roles and granting these roles to the users. For this reason there is no discretionary access control subsection here because explaining role-based access control also discretionary access control explained.

The following objects in the DBMS require users to have specific authorization or privileges in order to access them:

- Databases
- Tables
- Columns in tables
- Fragments
- Views
- User-defined types (UDT)
- Routines
- Languages

Most of the users query requests interact with one or more of these objects. The privileges granted to the user in these objects dictate whether the user can access or not the object. How Informix determines user privileges during the execution of a query is explained in figure 2.2 and is described here.

1. The system checks database privileges.

2. The system checks user privileges on the objects associated to the user query. User privileges are based on privileges granted to the user, privileges derived from users current role and privileges that belongs to public.
CHAPTER 2. STATE OF THE ART IN DATABASES ACCESS CONTROL

Figure 2.2: Privilege checking in SQL query, from [5].

Here an overview of the possible SQL object privileges for a given user or role:

<table>
<thead>
<tr>
<th>SQL object</th>
<th>Privileges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>CONNECT, RESOURCE, DBA</td>
</tr>
<tr>
<td>Table</td>
<td>SELECT, UPDATE, INSERT, DELETE, INDEX, ALTER, REFERENCES</td>
</tr>
<tr>
<td>Column</td>
<td>SELECT, UPDATE, REFERENCES</td>
</tr>
<tr>
<td>Fragment</td>
<td>INSERT, UPDATE, DELETE</td>
</tr>
<tr>
<td>View</td>
<td>SELECT, INSERT, UPDATE, DELETE</td>
</tr>
<tr>
<td>Sequence</td>
<td>SELECT, ALTER</td>
</tr>
<tr>
<td>UDT</td>
<td>USAGE, UNDER</td>
</tr>
<tr>
<td>Routine</td>
<td>EXECUTE</td>
</tr>
<tr>
<td>Language</td>
<td>USAGE</td>
</tr>
</tbody>
</table>

Table 2.5: Privileges associated with each SQL object

Informix implements with tables in the system the user/role privileges. Each object from the graphic above has a corresponding table in the system where the user privileges in such object are stored. Such table content a grantor and grantee column, the grantor column describe the user who has granted to the grantee user such privileges. Each object has a column with the set of authorizations available for that user. There are two SQL objects that have no table representation. These are View and Sequence SQL object but the authorization on this objects are calculated asking to the other object system tables. What the system does is as follows:

1. The system checks whether the user can connect to the database looking at the database table, and global privileges are from this table extracted.
2. The system checks the user object privileges.
2.2. VENDORS IMPLEMENTATIONS

(a) The system checks the table privileges if he is trying to access a table.
(b) The system checks the columns privileges in case that he has no access to the whole table.
(c) If the user is trying to access to another object such as a Routine, Language.. then the system will checks in the corresponding table the user privileges.

<table>
<thead>
<tr>
<th>SYSUSERS</th>
<th>SYSTAB</th>
<th>SYSCOL</th>
<th>SYSFRAG</th>
<th>SYSDTYPE</th>
<th>SYSPROC</th>
<th>SYSLANG</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>grantor</td>
<td>grantor</td>
<td>grantor</td>
<td>grantor</td>
<td>grantor</td>
<td>grantor</td>
</tr>
<tr>
<td>usertype</td>
<td>grantee</td>
<td>grantee</td>
<td>grantee</td>
<td>grantee</td>
<td>grantee</td>
<td>grantee</td>
</tr>
<tr>
<td>defrole</td>
<td>tabid</td>
<td>tabid</td>
<td>tabid</td>
<td>type</td>
<td>procid</td>
<td>langid</td>
</tr>
<tr>
<td></td>
<td>tabauth</td>
<td>colno</td>
<td>fragment</td>
<td>typeauth</td>
<td>procauth</td>
<td>langauth</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.6: System authorization tables

In the table 2.6 are described the grant tables above explained with the corresponding columns that each table content. What protect each table is already said but better to give a more specific description.

- The SYSUSERSAUTH table determines which type of user is connecting to the system, dba, normal user...
- The SYSTABAUTH table determines user/role privileges on tables, each privilege granted here is applied to the whole table and its columns. Here are also stored the privileges for the created sequences.
- The SYSCOLAUTH table helps to give more granularity to access control in tables. Each entry here determines specific privileges for a role in a specific column from a given table.
- The SYSFRAGAUTH table determines privileges on a fragment from a specific table.
- Informix supports user-defined data types (UDTs). The TYPE-level privileges are stored in the SYSTYPEAUTH table, and determines whether a user can use or not the type.
- Informix also support views. View privileges are checked on the corresponding SYSTABAUTH and/or SYSCOLAUTH tables.
- Sequences are created only by dba users and its privileges are similar to table-level privileges but only SELECT and ALTER privileges are allowed over sequences. Therefore its privileges are stored in SYSTABAUTH. When the user want to perform a select or alter on a sequence then the system checks the privileges on the SYSTABAUTH.
- SYSPROCAUTH table stores the privileges on routines.
- Languages are routines written in SPL or any other language such as C or JAVA. The SYSLANGAUTH table contents the privileges for each user for the languages routines.

**Mandatory Access Control**

While *discretionary access control* works at the level of database objects such as databases, tables, views, and so on, *label-based access control* LBAC (that is the name given by Informix developers to MAC) works at the level of data rows and columns. Tables are protected by security policies, and individual rows and columns are protected by security labels. The principle
with LBAC works, is that users whose security labels dominate the data labels protecting the rows (or columns) are able to read data. How is the dominance calculated? Any label is composed of individual components and elements within each component. A user label dominates a row label if individual components in the user label dominate individual components of a row label, for a given security policy. For instance, user label (HIGH, CEO) which briefly means that a user with a high access level belongs to the CEO of a company, will dominate the table label (LOW, Marketing) which briefly means that a table is protected with a LOW access level and belongs to the Marketing section of the company. The user will be able to read the table because each subelement of the label dominates the corresponding element of the table label, that is, HIGH level dominates LOW level and the CEO group dominates the Marketing group.

Some concepts will be introduce to explain how the label based access control works, however for a better understanding an example above the concepts will be also explain, and I recomend to read it parallel.

• Security label

The database administrator creates security labels and assigns them to individual users or roles. As a table is protected by a single security policy, only users with labels belonging to the tables security policy can access and modify the table. Other users, with no labels assigned, or with labels that belong to a different security policy, are not able to access the data in the table. A security label could be specific for read access, write access or both access.

A restriction when creating security labels is that there can be only one element specified as part of an ARRAY component. The components of a given security label are the following.

- ARRAY SECURITY COMPONENT TYPE

An ARRAY component type consists of an ordered set of elements, where an element occurring earlier in the ARRAY has a higher weight than the element occurring later.

- SET SECURITY COMPONENT TYPE

A SET component type constitutes an unordered set of elements. Unlike elements in an ARRAY component type, the elements in the SET component type do not have any weight relative to another.

- TREE SECURITY COMPONENT TYPE

A TREE component type consists of elements organized in a tree structure. An element can have any number of children, but only one parent.

• Security policy

A security policy is defined using the security components that have been created, that is, the array, set and tree security components. Based on the type of data in a table, a suitable security policy is defined that protects and provides granular data access to users. The security policy is used to protect the table. The rows and columns of a table
are protected by security labels, which are derived from the tables security policy.

A security policy contains one or more of the following possible rules:

- **IDSLBACREADARRAY**
  Each array component of the user security label must be greater than or equal to the array component of the data row security label. That is, only data at or below the level of the user can be read.

- **IDSLBACWRITEARRAY**
  Each array component of the user security label must be equal to the array component of the data row security label. That is, only data at the same level as the user can be written.

- **IDSLBACREADSET**
  Each set component of the user security label must include the set component of the data row security label.

- **IDSLBACWRITESET**
  Each set component of the user security label must include the set component of the data row security label.

- **IDSLBACREADTREE**
  Each tree component of the user security label must include at least one of the elements in the tree component of the data row security label (or the ancestor of one such element).

- **IDSLBACWRITETREE**
  Each tree component of the user security label must include at least one of the elements in the tree component of the data row security label (or the ancestor of one such element).

- **Label comparison**
  A label dominates another label if component elements in one label dominate the corresponding component elements in another label. How a component element in one label dominates the component element in another label depends on the component type. If all the components in label A dominate corresponding components in label B, then label A is said to dominate label B.

In the case of the ARRAY component type, the comparison depends on where the elements in the two labels are positioned within the array. If the ARRAY component element in label A is positioned earlier than the corresponding component element in the label B, then label A dominates with respect to the component that is being compared.

In the case of the SET component type, the comparison depends on whether the elements in one label are a super-set of elements in the other label. If the elements contained in label A’s SET component are a super-set of the elements contained in the corresponding component in label B, then label A dominates over label B with respect to the component that is being compared.

In the case of the TREE component type, the comparison depends on whether the element in one label is one of the elements in the second label or the ancestor of one such element. If each of the TREE component elements in label A is one of the elements in
the corresponding component in label B, or an ancestor of one such element in label B, 
label A dominates over label B with respect to the component that is being compared.

In totality, label A is said to dominate label B if each of the components in label A dom-
inates the corresponding components in label B.

• Protecting tables

Individual tables are protected by security policies. An unprotected table is one that has 
not any security policy attached. Users with no security labels granted to them can only 
access unprotected tables. In order to access protected tables, users should have READ 
and WRITE security labels that belong to the protected tables security policy. Rows and 
columns can be protected within a table:

1. When the row protection is enabled a extra column type IDSSECURITYLABEL 
is added to the table, this column determines the privileges for accessing such row. 
User labels get attached to rows when a user performs DML operations, storing the 
label in the extra column.

2. When the column protection is enable then the label associated with such column is 
stored in the SYSCOLAUTH table described in the role-based access control section.

3. Row and Column protection are not exclusive and can be use together.

Example

In the same context of the medical database, where in this case the patient table should be 
protected. The idea is that physicians and nurses cann access the table, of course all the physi-
cians may be access the whole patient information but not all the nurses should be able to the 
patient table. Let say that columns address and telephone are only able to some nurses.

Let create the security components, and the policy which owns the components,

CREATE SECURITY LABEL COMPONENT privacy_levels 
ARRAY {'confidential', 'unclassified'}

CREATE SECURITY LABEL COMPONENT classification 
TREE {'physician'ROOT, 'nurse' UNDER 'physician', 'other' UNDER 'nurse'}

CREATE SECURITY POLICY hospital_plcy COMPONENTS privacy_levels, classification 
let create the security label to assign to the employees,

CREATE SECURITY LABEL hospital_plcy.physician 
COMPONENT classification 'physician', 
COMPONENT privacy_level 'confidential' FOR ALL ACCESS;
CREATE SECURITY LABEL hospital_plcy.nurse 
COMPONENT classification 'nurse', 
COMPONENT privacy_level 'unclassified' FOR READ ACCESS;
CREATE SECURITY LABEL hospital_plcy.head_nurse 
COMPONENT classification 'nurse',
2.2. VENDORS IMPLEMENTATIONS

COMPONENT privacy_level 'confidential'
FOR READ ACCESS;

now let assign to the users such security labels, let say user Alice is a Physician, user Bob is a usual nurse and user Tom is the chief nurse.

GRANT SECURITY LABEL hospital_plcy.physician TO 'alice';
GRANT SECURITY LABEL hospital_plcy.nurse TO 'bob';
GRANT SECURITY LABEL hospital_plcy.head_nurse TO 'tom';

The last step is to protect the table with the already created policy.

CREATE TABLE client ( id1,
    name,
    age,
    address COLUMN SECURED WITH head_nurse,
    telephone COLUMN SECURED WITH head_nurse,
    disease COLUMN SECURED WITH physician
) SECURITY POLICY company_plcy;

Now the following query will be executed and the steps are as follow:

SELECT p.name, p.telephone, p.disease
FROM patient p
WHERE p.name = john

1. First the role-based access control is checked, let assume that users Alice, Bob and Tom have the authorization to access the client table. That is role-based access control first checks the database authorization and then the tables authorization in the SYSTABAUTH table, as all users can access the whole table information then the mandatory access control continues checking the users privileges.

2. patient table is protected with company_plcy, all users, Bob, Alice and Tom, own a label in such policy, so the LBAC will continuing checking the labels. In case of a user does not own a label in such policy then the user is not able to access the data in the table.

3. The column address is checked and users Alice and Tom are able to access the data because the security label that he owns dominates the security label assign to the address column. Nevertheless user Bob cannot access the column because his label does not dominates the address assigned label.

4. With the telephone column happen exactly the same as with the address column.

5. The column disease is checked. In this case only Alice’s label dominates the disease column label. Therefore this information will be not show to Bob and Tom.

The result is that only Alice can read the all the patient information, Tom is able to read all columns except the disease column and user Bob is able to read only the patient name in such query. Therefore any information will be show to users Tom and Bob however Alice will have the complete request information.

In case that the row protection is also enable the mechanism works in the same way, that is, the table will have an extra column with the security label that protect the row. For each requested row request the security labels from users and row will be checked, and the method will determine whether the row is available or not. When a user inserts a row in the table the new label added to such insertions will be the same as the security label that the user who performs the operation. For instance, user Bob with label security value 00997773333 insert the last two rows in the patient table.
Table 2.7: patient table with security label column

<table>
<thead>
<tr>
<th>id</th>
<th>IDSSECURITYLABEL</th>
<th>name</th>
<th>age</th>
<th>telephone</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1144777666</td>
<td>john</td>
<td>21</td>
<td>1111</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>1144777666</td>
<td>Luis</td>
<td>32</td>
<td>2222</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1144777666</td>
<td>peter</td>
<td>13</td>
<td>3333</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1144777666</td>
<td>Sara</td>
<td>54</td>
<td>4444</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1144777666</td>
<td>Paul</td>
<td>65</td>
<td>5555</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0099777333</td>
<td>Frank</td>
<td>20</td>
<td>6666</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0099777333</td>
<td>Carl</td>
<td>81</td>
<td>7777</td>
<td>1</td>
</tr>
</tbody>
</table>

Analysis

Informix mandatory access control enforcement gives a high granularity and very good user hierarchy management in order to access the data. The granularity level reach columns and rows in objects, but individual cells cannot be protected.

Nevertheless Informix has no context dependant access control. The content dependant as in MySQL could be implemented with views, but it does not offer an expressive mechanism as told before.
2.2.3 Oracle

The Oracle Database has become a major presence in database computing. In this section the Oracle Database 11g release 1 is going to be cover. Oracle access control model implements discretionary, mandatory and roled-based access control models as a classic access control models. Oracle support also a policy access control called by the authors Oracle Private Database Access Control.

**Roled-Based Access Control**

As said before, Oracle support also discretionairy access control where users can log in and access the data through a privilege system, either with his user privileges or owning a role and using role priviliges. Due to the privilege system works in the same way in both cases, here is only described the Roled-Based Access Control.

The following objects in the DBMS require users to have specific authorization or privileges in order to access them:

- Table
- View
- Sequences
- Procedure
- Functions and Packages
- Type

**Policy Access Control**

With this method the administrator can create policies for column and row level security writing PL/SQL code. Row and column level security is also available with the Mandatory Access Control without writing any PL/SQL code. The method is based on a query modification technique, that is called by the athors Oracle Virtual Private Database (OVPD). When a user access a table, view, or synonym with a OVPD policy, Oracle Database dynamically modifies the SQL statement of the user adding a WHERE condition.

There are two basic element in the OVPD, the function which generate the dynamic WHERE clause, and a policy to attach this function to the object to protect in the database. The function must have as argument the object to be protected (table, view, or synonym), and it must provide a return value to be use in the WHERE clause added predicate. On the other hand a policy will attach to a table, view or synonym the function and each time the object is accessed the WHERE clause will be added to the query.

The OVPD takes effect over all the users in the database including the object owner, of course the dba is not affected. In order to create the function, the oracle application context is available to retrieve user information, for more information about application context look at the chapter 6 in the Oracle Database Security Guide [7]. For instance, if we want to allow the access to the patient table only to such patients that are located in the first floor of the hospital, because all these patients have a mild disease:
CHAPTER 2. STATE OF THE ART IN DATABASES ACCESS CONTROL

In each function the return value must be a correct column type within the table to be protected. Let now attach a policy to the table:

```
SQL> BEGIN
  2    DBMS_RLS.ADD_POLICY (
  3        object_schema => 'medical',
  4        object_name => 'patient',
  5        policy_name => 'patient_policy',
  6        function_schema => 'sys',
  7        policy_function => 'patient_access',
  8        statement_types => 'select, insert, update, delete'
  9    );
 10  END;
 11 /
```

Right now, for instance the query `SELECT name, location FROM medical.patient` will be modified to `SELECT * FROM medical.patient WHERE location = 1`. There are more possibilities in the policies setup, such as return a complete row with a null value inside the cells columns that are not allowed to access, instead of not return the complete row. All possibilities are showed in the Oracle Database Security Guide chapter 7 [8].

This method takes effect after the role-based access control, modifying the query request, that is, each request must be checked at first by the discretionary access control and later the query will be modified. Basically this method offers the same granularity level as Mandatory Access, but also offers a better context protection.

### Mandatory Access Control

To be allowed access to a row, a user must first satisfy Oracle Database Discretionary Access Control (DAC) requirements, then the OVPD modifications take effect, and finally, satisfy Oracle Mandatory Access Control (MAC) requirements. Oracle MAC implementation is called Oracle Label Security (OLS). For instance, when a user execute a `SELECT` statement on a table and the DAC allow the access to the table, and the OVPD modifies the query, the OLS evaluates each row selected to determine whether the user can access it or not, and of course, this security checks can be configured to UPDATE, DELETE and INSERT statement as well. The principle with LBS works, is that users whose labels dominate the data labels protecting the rows are able to access the data. How is the dominance calculated? Any label is composed of individual component and elements within each component. A user label dominates a row
label if individual components in the user label dominate individual components of a row label, for a given policy.

- **Label components**
  - **Level component.** A level component type consist of an ordered set of elements, where an element occurring earlier in the level component has a higher weight than the element occurring later.
  - **Compartment component.** A compartment component type constitutes an unordered set of element. Here elements do not have any weight relative to another. This component goal is to provided a finer level of granularity.
  - **Group component.** A group component type consists of element organized in a tree structure. An element can have any number of children, but only one parent.

- **Label comparison**

A user can access data only within the range of his own label authorizations. A user has a maximum and minimum levels, a set of authorized compartments and a set of authorized groups (and, implicitly, authorization for any subgroups). Each user has a session label which consist in a a particular combination of level, compartments and groups which a user owns and can be changed by the user. Also the administrator assign to the user a default label when he connect.

A label dominates another label if component elements in one label dominate the corresponding component elements in another label. How a component in one label dominates the component element in another label depends on the component type. If all the components in label A (user label) dominate corresponding components in label B (row label), then label A is said to dominate label B.

In the case of the level component, if element in label A is positioned earlier than the corresponding component element in the label B, then level component from A dominates level component from B.

In the case of compartments, if the elements contained in label set A are a super-set of elements in the other label then component from label A dominates B component.

In the case of group compartment, if the element in label A is one of the elements on the label B or the ancestor of one of such elements, then the label A component dominates the label B component.

- **Policies**
  - **Read policy**

The following rules are used to determine a users read access to a row of data, in the order given:

1. The user’s level must be greater than or equal to the level of the data.
2. The user’s label must include at least one of the groups that belong to the data (or the parent group of one such subgroup).
3. The user’s label must include all the compartments that belong to the data.

– Write policy

To determine whether a user can write a particular row of data, OLS evaluates the following rules, in the order given:

1. The level in the data must be greater than or equal to the users minimum level and less than or equal to the users session level.

2. When groups are present, the users label must include at leas one of the groups with write access that appear in the data label (or the parent of one such subgroup). In addition, the users label must include all the compartments in the data label.

3. When no groups are present, the users label must have write access on all the compartments in the data label.

– Special Privileges

There are some special privileges which allow to the users special privileges like full read access or full read and write access. The are described in the Oracle Label Security Guide chapter 3 section 5 [8].

2.2.4 Summary

The table 2.8 has a summary with the vendor’s access controls supported. The table explain the main characteristic of each access control for a given vendor such as the level of granularity, and the content and context dependant possibilities to protect the data access.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Access Control</th>
<th>Granularity</th>
<th>content</th>
<th>context</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>Discretionary</td>
<td>view</td>
<td>based on views</td>
<td>without support</td>
</tr>
<tr>
<td>Informix</td>
<td>Discretionary</td>
<td>view</td>
<td>based on views</td>
<td>without support</td>
</tr>
<tr>
<td></td>
<td>Role based</td>
<td>view</td>
<td>based on views</td>
<td>without support</td>
</tr>
<tr>
<td></td>
<td>Mandatory</td>
<td>row and column</td>
<td>based on views</td>
<td>without support</td>
</tr>
<tr>
<td>Oracle</td>
<td>Discretionary</td>
<td>view</td>
<td>based on views</td>
<td>without support</td>
</tr>
<tr>
<td></td>
<td>Role based</td>
<td>view</td>
<td>based on views</td>
<td>without support</td>
</tr>
<tr>
<td></td>
<td>OVPD</td>
<td>row and column</td>
<td>based on views and oracle application context</td>
<td>without support</td>
</tr>
<tr>
<td></td>
<td>Mandatory</td>
<td>row and column</td>
<td>based on views</td>
<td>without support</td>
</tr>
</tbody>
</table>

Table 2.8: vendors main characteristics

As we can see in the table there is no context support for any vendor implementation. Another conclusion is that all vendor implementations for different access control methods offer the same granularity and content dependant protection. Also Oracle implements the Oracle Virtual Private Database which is able to give some extra conditions in order to protect data based on content dependant.
2.3 Research Approaches

The problem focused on this document is how to expand the expressiveness in nowadays access control relational databases systems. This topic has been already discussed not only in the RDBMS also in other databases types as RDF databases or semistructured databases, however there is not too much research about the access control in RDBMS in the last years. In this section I want to cover approaches extracted from some papers which maybe some useful ideas could be extracted for the future work of this document. Although some of this papers approaches are not based on RDBMS, they are based in the databases access control.

Let start with related work about RDBMS. There is not too much researches about hencing the conditions of access control possibilities, however about user management is easy to find papers. One possibility in the user management is the credential based access control, in this area about trust managament amount of papers have been writed, however not too much directly focused on Relational Databases. Trust Management Services in Relational Databases [10] presents a design that explains how the trust management can be implemented within a relational database. The authors exposed an extended SQL language in order to support all trust management concepts with SQL language, in that way each trust management concept is stored in a different table of the database. This technich offers a high expresiveness syntax and flexibility thanks to SQL and RDBMS power. One of the advantages of the credential based system is that not only solve the user management, they also offer a context access control based on the credentials that clients or servers prove to have. The concepts from the Trust Management that can be represented like an object in the datasets are the Authorities, the attributes to use in a policy and the policies itself.

Here is exposed how the research approach represents the trust principal trust management concepts:

- **Authority**

  The concept of authority defines the identities that are trusted to issues certificates. Here is the syntax and an example where DOH (department of health) authority is created with his X.509 certificate attributes:

  ```sql
  create authority AuthorityName
  [imported by FileName]
  [public_key = AttrValue
  {, AttrName = AttrValue}]

  create authority DOH
  public_key = '14:c9:ec...:4f:91:51',
  CN = 'Department of Health',
  O = 'Goverment',
  C = 'IT'
  ```

- **Certified Attributes (trust tables)**

  Trust tables represents all attributes that will be used in a policy. That is, the meaning of what it is possible to consider in the information provided through certificates in rules and queries.
create trusttable TrustTableName
[authoritative
  AuthorityClassOrName [with [no] delegation]
  {,AuthorityClassOrName [with [no] delegation]}]
[except AuthorityName {, AuthorityName}]

(AttrName AttrDomain [check (Condition)]
  {,AttrName AttrDomain [check (Condition)]}
  [check (Condition)])

Where TrustTableName represents a name associated with the set of attributes extracted from the certificate signed by a given authority.

The authoritative clause describes the authorities that are trusted as signers of certificates producing the specified set of attributes. When this clause is missing then an independent mechanism from the proposed is used for certificate verification service.

If no delegation is specified the software module responsible for verifying the integrity of the certificate will no allow certificate chains.

The except clause allow to the dba exclude specific authorities.

Here a syntax example:

create trusttable Physicians
  authoritative HealthGovAgency with delegation
  except HealthSchoolAuth
  (code char(9),
   name varchar(25),
   license_number int check (license_number is not null),
   speciality carchar(20)
  )

• Policy

A trust management policy regulates the access to resources based on the attributes stated by verified certificates. In this case the RDBMS should exploit the certified attributes to regulate access. This section explain how the certified attributes are used to regulate role activation in the RDBMS. Here is the syntax and an example where a the role Cardiologist for each user who has a certificate from a health government agency is activated when the user proves that he is a doctor specialized in cardiology:

create trustpolicy [PolicyName]
  [for Role [autoactive] | UserId]
  where Condition

create trustpolicy
  for Cardiologist autoactive
  where Physicians.speciality = 'Cardiology'
Condition is any predicate that can appear in the where clause of a SQL query and can refer to the trust table using its name and specifying the attributes contained in it.

The autoactivate clause activates a role automatically when this clause is forgotten then the user who satisfied the conditions is assigned with PUBLIC role privileges.

Basically with this design the dba stored the authorities in which the system rely, the attributes that can be use in the policy, and defines the policy rules. Later when a user presents his own credentials the system will check whether the corresponding policy is satisfied, and in that case a role will be activate for the specific user who will be able to perform role operations. After the role activation the role-based access control will continue checking the user operations permissions. The approach architecture could be describe as adding a top layer to the role-based access control in order to activate a role when a user connects to the database.

One of the most important advantages in this approach is the possibility to add context dependent access control, as well as in the already described vendor implementations with views is possible to have a content dependent access control, in this approach the context access control could be also implemented with the help of views, where access to data may depend on the properties of the user. Let see an example where the following view grant each physician access to the data of her patients:

```sql
create view PatientView as
    select Patients.*
    from Patients, Physicians
    where Physicians.code = Patients.doctor_code
```

On the other hand the main disadvantage here is the usability for the users, this always a fact in the cryptography implementations. The users should be related with digital signs and certificates. This also overload the dba configuration, he must configure all the protocols, add the trusted authorities and certified attributes.

Another research approach more related with the main theme of this thesis but focused in the RDF databases is the one proposed in Enabling Advanced and Context-Dependent Access Control in RDF Stores [1]. RDF stores usually protect the whole database information allowing or not the access to the whole data stored, without any kind of granularity. In this approach, a more fine grained access control as well as context dependent protection is proposed. The authors propose a layer on top of the RDF stores making the design store independent. This layer will checks by a set of defined policies whether the query request is allow to access or not the information it want to get. When a query arrive, the policy engine evaluates the contextual conditions such as time or possible credentials that the user should own. After contextual conditions checking they expand the queries adding some constraint before send it to the RDF database. Therefore the top layer has two different modules, the policy engine that evaluate context conditions, and the query expansion which add constraint to queries. Both modules evaluate each query based on the policy rules that the database administrator has previously defined.

The policies rules that restrict or allow the access to RDF should be able to specify graph patterns, that is, the information to be accessed, and the ability to check contextual conditions. A policy rule determines if the access to the RDF statement is allow or deny. Therefore the result of evaluate a policy rule will be always allow or deny the access. To solve the conflicts among rules that protect or allow the access to the same resources a simple policy
evaluation algorithm is used; if a deny policy is applicable then the resource is denied, else if an allow policy is applicable then the access is allowed, else the access to the resource is denied.

<table>
<thead>
<tr>
<th>No</th>
<th>Policy</th>
</tr>
</thead>
</table>
| pol₁ | deny access to triples \((X, \text{foaf:phone}, Z)\) IF  
\((X, \text{foaf:currentProject}, l3s:\text{rewerse}) \text{ AND}  
\text{Requester} = \text{RecommenderService}.\) |
| pol₂ | allow access to triples \((X, \text{foaf:phone}, Z)\) IF  
\text{Requester} = \text{Service} \text{ AND}  
\text{Service} \text{ is a trusted service} \text{ AND}  
\((l3s:\text{alice}, \text{foaf:knows}, X).\) |
| pol₃ | allow access to triples \((l3s:\text{alice}, \text{foaf:phone}, Z)\) |
| pol₄ | allow access to triples \((X, Y, Z)\) IF  
\text{Time is the current time} \text{ AND}  
\text{09:00 < Time AND Time < 17:00 AND}  
\text{Y} = \text{foaf:name AND X != l3s:tom.}\) |
| pol₅ | allow access to triples \((l3s:\text{alice}, \text{foaf:interest}, Z)\) IF  
\((Z, \text{rdf:type}, \text{foaf:Document}) \text{ AND}  
(X, \text{foaf:currentProject}, P) \text{ AND}  
(Z, \text{foaf:topic}, T) \text{ AND} (P, \text{foaf:topic}, T).\) |

Table 2.9: Example of high level policies controlling access to RDF statements

The table 2.9 describes some possible policies in a high level language, where the meaning of each policy is as follow:

1. RecommenderService is not allowed to access the telephone numbers of the REWERSE project members.
2. Recognized trusted requesters are allowed to access the telephone numbers from the people who Alice knows.
3. Anyone can access the Alice telephone number.
4. Anyone can access the name statement during working time with the exception tom name statement.
5. Only Alices interests related to her current project can be accessed.

After the contextual evaluation by the policy engine, which extract from each policy the contextual conditions and then checks that the query fulfil them. The query extension is performed, at this level some constraints are added to the final query in order to restrict the access to the protected data specified in the policy rules. Briefly how the query extension algorithm works is adding boolean expressions to the query as here is described:

1. The algorithm extracts path expressions found in the body of the policy rules, for each path a set of boolean expressions are extracted, as well as variables that could be use in more than one rule. This booleans expression will be the future constraints.
2. The set of the before boolean expressions extracted from all the allow policies that match with on of the FROM clause are connected with the OR boolean expression.
3. Policies constraints that match with on of the different FROM clauses are connected with the AND expression.

4. Finally is necessary to remove the triples affected with disallow policies, this is specified with the MINUS operator.

Thanks to the power offered by the policy engines this approach gives an expressive context and content dependant protection to the data.

Next and last type of databases to analyse are the XML databases, since the use of XML documents for data storage instead of relational databases, it is necessary to think about a security system that protect XML documents accesses. Damiani in his article called *A fine-grained access control system for XML documents* [6] was the first in introduce an access control system to the XML documents. The authors of the document propose an access control model based on a set of authorizations rules composed of subjects, objects, and the actions that users can perform over the objects. Each authorization rule determines if the access is allow or not to the described object. The idea is quite similar to the last one described with policy rules. Here each authoritazion rule is expressed as a five tuple of the form:

\[(\text{subject}, \text{object}, \text{action}, \text{sign}, \text{type})\]

where subjects are identify by his identity and his location, objects are represented with an URI and his granularity is determined with a path expression, that is, URI:path. The only possible action is READ but a briefly description to supports also WRITE operation is proposed. The sign positive indicates to allow the access, and the negative to deny the access. Finally the type determines how the authorization should be treated in case of propation. The type could be local (only apply to the specified elements attribute ) or recursive (apply to the subelements and their attributes). Type authorization can be applied at instance level (XML document) or at schema level (DTD document). To avoid conflict situation with type authorization between schema level and instance level, an authorization could be soft (only apply unless there is no schema level authorization) and hard (applied always ignoring instance level authorization). Table 2.10 describes the type possibilities.

<table>
<thead>
<tr>
<th>Level/Strength</th>
<th>Local</th>
<th>Recursive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>L</td>
<td>R</td>
</tr>
<tr>
<td>Instance (soft)</td>
<td>LS</td>
<td>RS</td>
</tr>
<tr>
<td>DTD</td>
<td>LD</td>
<td>RD</td>
</tr>
<tr>
<td>DTD (hard)</td>
<td>LDH</td>
<td>RDH</td>
</tr>
</tbody>
</table>

Table 2.10: Authorization Types

When a user makes a request on an object, it is necessary to evaluate which part of the document he is allow to access. The authors proposed a model where the system create a view of the document for each requester. The view generates by the system depends on the authorization rules above describe. The authors proposed algorithm to the creation of such a view is called tree labeling process which is follow by a transformation process in order to determine the subject authorizations on XML documents. Let describe briefly the how the access control process works:

1. Given an access request and the XML document to be accessed, the tree labeling process generates the tree corresponding to such document, for each of its nodes, the algorithm determines whether the requester is allowed or denied to access it.
2. For that purposed each node in the tree is associated with a vector \( n.\text{veclabel}[t] \), where type \( t \in \{L, LS, LD, LDH, R, RS, RD, RDH\} \). Both authorizations, the positive; \( n.\text{veclabel}[t].\text{Allowed} \) and the negative; \( n.\text{veclabel}[t].\text{Denied} \) depends on the sign + or -. Where the conflicts are resolved by applying a conflict resolution policy [12].

3. Next step is to propagate the labels where: authorizations on a node take precedence over those on its ancestors and authorizations at the instance level unless declared as soft, take precedence over authorizations at the schema level, unless declared as hard. The nodes that do not have any associated sign are associated with the negative type, that is, the closed policy is applied.

4. Right now all the nodes involve in the request have a label attached which determines the subject access. The view can be computed. In order to have a valid DTD structure in the new view a prune as well as a loosening transformation that consist on marking the required attributes of the DTD as optional are made to obtain the view.

A quite similar approach with some added improvements is described in *Specifying and enforcing access control policies for XML document sources* [3]. This new approach consider the case of documents not conforming/partially conforming to a DTD. Therefore the model provides to the Security Administrator for dealing with such documents. The second improvement to [6] is to provide access control modes specific to XML documents, because [6] only provides the read access mode. By the contrast [3] provide a number of specialized access modes for browsing and authoring, which allow the Security Administrator to authorize a user to read the information in an element and/or to navigate to navigate through its links, or to modify/delete one of the element links.
Chapter 3

Problem Solution: Concept
CHAPTER 3. PROBLEM SOLUTION: CONCEPT

This chapter is divided in three sections, the first one introduces the policy/rule concept and explains how to use these rules to protect the databases objects with examples. The second section explains and analyze with natural language how should the rules performance by a system which wants to add this advanced policy-based access control system, as well as explains the different possibilities and problems to implement it. The last section describes a solution in a formal way with definitions and an algorithm to implement the policy-based access control.

3.1 High level rules introduction

High level rules basic concepts

There is one rule format to use in the head of the rules, such format defines the object to be protected or allowed

\[
\text{db(database\_name, table\_name, column\_name, [column_1 = value_1,..,column_N = value_N])}
\]

Where database\_name refers to an specific database, table\_name refers to a specific table inside a database, column\_name and column\_i refer to an specific column inside the before described table, and value\_i refers to an specific value given by the user that can be or not in the content of the corresponding column\_i. With this rule is possible to refer to databases, tables, columns, rows and cells.

The other format is only used for querying the database and is used in the body of the rules, after the if from the head of the rules.

\[
\text{dbquery(database\_name, table\_name, [column_1 = value_1,..,column_N = value_N])}
\]

The main goal of the dbquery rule is to add more flexibility adding more conditions when the access to the object described in the head of the rule is requested.

The below example shows the two different rules used in this document, the rule says that the access to the column name in the patient table from the medical database is denied is the column position has the value first minister.

\[
\text{Rule example}
\begin{align*}
\text{Deny access to } & \text{db(medical, patient, name, _)} \text{ if} \\
& \text{dbquery(medical, patient, [position = first minister])}
\end{align*}
\]

Possible improvements to the rules format

Databases also allow protection to the operations, that is, operations such as SELECT, INSERT or other. A good idea is to add a new parameter operation at the beginning of the rules, for instance:

\[
\text{db(operation, database\_name, table\_name, column\_name, [column_1 = value_1,..,column_N = value_N])}
\]
3.1. HIGH LEVEL RULES INTRODUCTION

With the possibility of group the operations into groups like READ or WRITE access. In this thesis work only the READ privilege, that is, the SELECT statement from the SQL is going to be assume.

High level rules possibilities

Now let describe high level rules examples where the rules protect the information that vendors database are available to protect. Let suppose the following tables:

\[
\begin{align*}
\text{patient} & (\text{name, telephone, address, position, disease}) \\
\text{disease} & (\text{name, type, dangerousness})
\end{align*}
\]

Where patient.disease and disease.name have the same range of values.

1. Protecting a database.

\[
\begin{align*}
\text{deny access to db(medical, \_ , \_ , \_ ) } & \\
\text{allow access to db(medical, \_ , \_ , \_ ) if } & \\
\text{Requester = Employee}
\end{align*}
\]

The first rule protects the access to the database named medical to any user, and the second allows the access only to the Employees. Here the Requester statement allows the possibility to assign for instance roles to the rule constraints.

2. Protecting a table.

\[
\begin{align*}
\text{deny access to db(medical, patient, \_ , \_ ) } & \\
\text{allow access to db(medical, patient, \_ , \_ ) if } & \\
\text{Requester = Physician}
\end{align*}
\]

The first rule protects the access to the patient table inside the medical database, and the second one allows it to the Physicians.

3. Protecting a column.

\[
\text{deny access to db(medical, patient, telephone, \_ )}
\]

The rule protects the access to the patients telephone.

4. Protecting a row.

\[
\text{deny access to db(medical, patient, \_ , [position = First Minister])}
\]
CHAPTER 3. PROBLEM SOLUTION: CONCEPT

This rule protect the access to all the rows from the patient table if the patient’s position column matches with First Minister.

Let us expand the vendors possibilities to protect information such as cells or protection based on the content of other objects in the database:

5. Protecting a cell.

\[
\text{deny access to } \text{db(medical, patient, telephone, [position = First Minister])}
\]

Imagine that the First Minister’s telephone number is not allow to be accessed. This rule protect the access to the telephone number of the patient which patient’s position matches with First Minister.

6. Protecting a column based on the column information.

\[
\text{deny access to } \text{db(medical, patient, disease, _)} \text{ if } \\
\text{dbquery(medical, patient, [disease = cancer])}
\]

The patient’s disease column is not allowed to be accessed if exists any value in the column that matches with cancer. In a usual DBMS protecting a column is only possible with the MAC, but is not so expressive, also with OVPD is possible to protect columns with a little bit more expressiveness. See section 2.2.3.

7. Protecting a column based on other table information.

\[
\text{deny access to } \text{db(medical, patient, disease, _)} \text{ if } \\
\text{dbquery(medical, patient, [disease = X]) and } \\
\text{dbquery(medical, disease, [name = X, dangerousness = High])}
\]

Imagine that the patient’s disease column should be protected in case of existing any disease in the column consider with a high level of danger in the table disease. This rule protect such information.

8. Protecting a row based on other table information.

\[
\text{deny access to } \text{db(medical, patient, _, [disease = X])} \text{ if } \\
\text{dbquery(medical, disease, [name = X, dangerousness = High])}
\]

Imagine that all the patient’s row information whose disease column is considered with a high level of danger should not be accessed. This rule protect the rows whose disease columns are consider with a high level of danger in the table disease.

As well as the columns, rows or cells are able to be protected based on information based on other tables, the tables and databases are also possible to be protected based on other tables from other databases.
Solving rule conflicts

How to solve the policy conflicts whenever two or more different policies allow and deny the access to the same resource has been already studied. In this case a simple algorithm is going to be followed:

\[
\text{if a deny rule is applicable then access to the resource is denied}
\]
\[
\text{else if an allow rule is applicable then access to the resource is allowed}
\]
\[
\text{else access to the resource is denied (deny by default)}
\]
3.2 High level rules evaluation

Some high level rules possibilities have been shown in the last subsection, however these possibilities are only a sample and in this section all the possible combinations are going to be studied and evaluated.

The objects or elements that can be protected with the rules syntax are the following: databases, tables, columns, rows and cells, and each one of these objects can be also protected based on the following conditions: without conditions, based on data conditions from the same table where the object is, or based on data conditions from a different table where the object is, even based on a table information from other database. There are a couple of exceptions, for instance, a row can not be protected without conditions, in other case we are not protecting a row (it is not a good idea to protect the $7^{th}$ row from a table). In any case, there are a lot of possibilities to combine, this could made the analysis of all the rules annoying. Therefore protection based on context conditions such as time will be later analyze and here just concentrate in the content conditions.

There is another fact to take into account, after the query evaluation the actions that the rule performs are different for each query request. It means that the analysis for each rule should take into account the different requesting queries, for instance when a column is protected the query could request the protected column, could not request the protected column or could ask for the whole information with the * sign, as well as other possibilities. Of course sometimes the result and actions of the rule are the same for different queries requests, therefore they will be grouped in the analysis.

Denying access to databases

1. Protecting a database without conditions.

   deny access to db(medical, _, _, _)

   • Here the actions are the same for each query request.
   • The result is an empty set.
   • The request query is not modified and not executed.
   • How the rule is evaluated:

     1. Extract from the FROM clause the database to be accessed.
     2. Extract from the rule the database to be protected.
     3. If the database extracted from the query and the database extracted from the deny rule do not match, then the checks continues checking other rules.
     4. The user is not allowed to access the database and an empty value is returned.

2. Protecting a database with conditions.

   deny access to db(medical, _, _, _) if
   dbquery(system, database, [name = medical, access = denied])
Given the table 3.1 and the above database table which contains databases and some information about them. The rule protects the medical database if the column access from the database table has the *denied* value.

- Here the actions are the same for each query request.
- The result depends on the conditions, if the conditions are not satisfied the result is an *empty set*.
- The request query is *not modified* and *maybe not executed*.
- How the rule is evaluated:

  1. Extract from the FROM clause the database to be accessed.
  2. Extract from the rule the database to be protected.
  3. If the database extracted from the query and the database extracted from the deny rule do not match, then the check continuous with a different rule. If not, continue asking to the database.

- Asking the database:

  The question to the database is as follows
  
  ```sql
  SELECT * FROM system.database d
  WHERE d.name = medical AND d.access = denied
  ```
  
  If the result is not an empty set then the access to the database is denied and the query is not executed. If the result is an empty set then the system will continue checking the permissions.

- How the question query is generated

  1. A SELECT statement is created requesting the whole information with the * sign.
  2. In the FROM clause is added the table where the rule body conditions are based.
  3. A WHERE clause is added to the SELECT statement.
  4. The conditions extracted from the rule body are added to the new WHERE clause connected with a logical *and*.
  5. The query is executed with full access to the tables where the query access.
Denying access to tables

3. Protecting a table without conditions.

\texttt{deny \text{access to db(medical, patient, _, _)}}

- Here the actions are the same for each query request.
- The result will be an \textit{empty set}.
- The request query is \textit{not modified} and \textit{not executed}.
- How the rule is evaluated:
  1. Extract from the \texttt{FROM} clause the database and table to be accessed.
  2. Extract from the rule the database and table to be protected.
  3. If the table and the database extracted from the deny rule do not match with the table and database extracted from the \texttt{FROM} clause, the user is allow to access the requested data. If they match continue.
  4. The user is not allow to access and an empty value will be returned.

4. Protecting a table with conditions.

\texttt{deny \text{access to db(medical, patient, _, _) if}}
\texttt{dbquery(medical, patient, [name = unknown, disease = cancer])}

- Here the actions are the same for each query request.
- If the conditions are satisfied the result is an \textit{empty set}.
- The request query is \textit{not modified} and \textit{maybe not executed}.
- How the rule is evaluated:
  1. Extract from the \texttt{FROM} clause the database and table to be accessed.
  2. Extract from the rule the database and table to be protected.
  3. If the table and the database extracted from the deny rule do not match with the table and database extracted from the \texttt{FROM} clause, the user is allow to access the requested data. If they match continue asking the database.

- Asking the database:

  The question to the database is as follows
  \texttt{SELECT * FROM medical.patient p WHERE p.disease = cancer}

  If the result is not an empty set, then the access to the database is denied and the query is not executed. If the result is an empty set, then the system will continue checking the permissions.
3.2. HIGH LEVEL RULES EVALUATION

• How the question is generated

1. A SELECT statement is created requesting the whole information with the * sign.

2. In the FROM clause is added the table where the rule body conditions are based.

3. A WHERE clause is added to the SELECT statement.

4. The conditions extracted from the rule body are added to the new WHERE clause.

5. The query is executed with full access to the tables where the query access.

Denying access to columns

5. Protecting a column without conditions.

deny access to db(medical, patient, telephone, _)

In databases when access requests all the columns from the table with the * sign, and the access is not allowed to one or more columns, the result is an error that says that the user is not allowed to access to one or more columns. That is, the access is not allowed to the whole row. In case of a request that just ask for some columns and one or more of them are protected, the result is the same error.

However in the thesis are two different cases, when the user request ask for all the columns with the * sign, then all the query request be returned without the protected columns. The other case is when the user ask for some specific columns and one or more of them are protected, in such case an error is returned to the user.

(a) Requested query is the usual: SELECT * FROM medical.patient

- The result will be the requested query without the protected column.
- The final query will be:
  SELECT name, address, position, disease FROM medical.patient
- How the rule is evaluated:

1. Extract from the FROM clause the database and table to be accessed.

2. Extract from the SELECT clause the columns to be accessed.

3. Extract from the rule the database, table and column to be protected.

4. If the database and table extracted from the deny rule do not match with the database and table extracted from the FROM clause then the access is allowed.
• How the query modification is performed:

1. Replace the * sign for all the columns from the accessed table.

2. Delete the protected column from the SELECT statement.

(b) Requested query is `SELECT name, telephone FROM medical.patient`

• The result will an error message.

• The request query is *not modified* and *not executed*.

• How the rule is evaluated:

1. Extract from the FROM clause the database and table to be accessed.

2. Extract from the SELECT clause the columns to be accessed.

3. Extract from the rule the database, table and column to be protected.

4. If the database and table extracted from the deny rule do not match with the database and table extracted from the FROM clause, then the access is allowed. If they match continue.

5. Look at the columns to be accessed in the SELECT statement, if they contain the protected column from the deny rule, the access will not be allowed, and an error message will be returned. If the protected column is not there, then the original request result will be returned.

(c) Following the above steps with the query `SELECT name FROM medical.patient`, the result will be the requested because the column name does not match with the protected column telephone.

6. Protecting a column based on the column information.

```
deny access to db(medical, patient, disease, _) if
dbquery(medical, patient, [disease = cancer])
```

The protection of the column depends on the conditions that are in the body of the rule, if the conditions are true then the access to the column will be denied, if not, the access to the whole object is allowed. Therefore in this case after checking if the rule is applied, is needed to ask to the database whether the conditions are true or not.

As explained in the point 3 with the column protection, two different cases should be distinguish, when in the select clause the whole information is requested with the * sign, and when in the select clause only some columns are requested.

(a) Requested query is the usual `SELECT * FROM medical.patient`

• The result will be the original query if the rule’s body conditions are not satisfied, or the requested query without the protected column when the conditions from the rule’s body are satisfied.
• The final query in case of modification will be:
  SELECT name, address, position, telephone FROM medical.patient

• How the rule is evaluated:

  1. Extract from the FROM clause the database and table to be accessed.
  2. Extract from the database the columns to be accessed.
  3. Extract from the rule the database and table to be protected.
  4. If the database and table extracted from the deny rule do not match with the database and table extracted from the FROM clause then the access is allowed, if they match then continue asking the database.

• Asking the database.

  The query to the database is as follows
  SELECT * FROM medical.patient p WHERE p.disease = cancer
  when the result of the query is an empty result then the access to the column is allowed, if the result is not empty then continue with the query modification.

• How is the query constructed.

  1. A SELECT statement is created requesting the whole information with the * sign.
  2. In the FROM clause is added the table where the rule body conditions are based.
  3. A WHERE clause is added to the SELECT statement.
  4. The conditions extracted from the rule body are added to the WHERE clause.
  5. The query is executed with full access to the tables where the query access.

• How the query is modified.

  The * sign must be change for the list of allowed columns after the evaluation.

(b) Requested query is SELECT name, telephone FROM medical.patient

• If the rule’s body conditions are false then the result will be the original query result, if the rule’s body conditions are satisfied then the result will be an empty value for the requested access and the query is not executed.

• How the rule is evaluated:

  1. Extract from the FROM clause the database and table to be accessed.
2. Extract from the SELECT clause the columns to be accessed.

3. Extract from the rule the database, table and column to be protected.

4. If the database and table extracted from the deny rule do not match with the database and table extracted from the FROM clause then the access is allowed, if they match then continue.

5. Look at the columns to be accessed in the SELECT statement, if they content the protected column from the deny rule, then continue with asking the database. If the protected column is not there, then the original request result will be returned.

- Asking the database.

The query to the database is as follows

\[
\text{SELECT } * \text{ FROM medical.patient } p \text{ WHERE } p.\text{disease} = \text{cancer}
\]

when the result of the query is an empty result then the access to the column is allowed, if the result is not empty then an empty value is returned.

7. Protecting a column based on a different table information.

\[
\text{deny access to db(medical, patient, disease, _)} \text{ if dbquery(medical, patient, [disease = X]) and dbquery(medical, disease, [name = X, dangerousness = High])}
\]

The only difference to the above point 6 is the step of asking the database, in this case the query is the following one

\[
\text{SELECT } * \text{ FROM patient } p, \text{ disease } d
\]

\[
\text{WHERE } p.\text{disease} = d.\text{name AND } d.\text{dangerousness} = \text{High}
\]

To construct the question the FROM block should contains the both tables, the one to be accessed in the original query and the one which contains the protection conditions. The WHERE block must contains the join condition as well.

**Denying access to rows**

8. Protecting a row based on the table information.

\[
\text{deny access to db(medical, patient, _, [position = First Minister])}
\]

- Here the actions are the same for each query request.
- The result will be all the requested rows without the rows where the position column value match with First Minister.
- Supposing that the original request is \text{SELECT } * \text{ FROM patient}, in the final query there are two different possibilities:
SELECT patient.* FROM medical.patient
MINUS
SELECT patient.* FROM medical.patient WHERE position = First Minister
or
SELECT patient.* FROM medical.patient WHERE NOT(position <> First Minister)

- How the rule is evaluated:

  1. Extract from the FROM clause the database and table to be accessed.
  2. Extract from the rule the database and table to be protected.
  3. If the database and table extracted from the deny rule do not match with the database and table extracted from the deny rule, then the access is denied. If not continue with query modification.
  4. Extract from the requested access the rows where the position is equal to the protected value and return it to the user.

- How the query is modified for the first option:

  1. Extract from the rule the access condition.
  2. Add a MINUS clause to the original query.
  3. Add a SELECT statement to the right operator of the MINUS clause.
  4. Add the same accessed columns from the original SELECT clause to the new SELECT clause.
  5. Add the same accessed table from the original FROM clause to the new FROM clause.
  6. Add a WHERE clause to the new SELECT statement
  7. Add the condition to the WHERE clause from the new SELECT statement.

- How the query is modified for the second option:

  1. Extract from the rule the access condition.
  2. Add a WHERE clause to the original query (if necessary).
  3. Add the negation of the condition to the new WHERE clause, or concatenate it to the original WHERE clause with a logical and operator.

The second option will be used in this work because it just has one SELECT statement, therefore the database execution will be faster.
9. Protecting a row based on other table information.

\[
\text{deny access to } \text{db(medical, patient, _, [disease = X]) if dbquery(medical, disease, [name = X, dangerousness = High])}
\]

- Here the actions are the same for each query request.
- The result will be all the patient data without the rows where the patient disease is considered with high level of danger.
- Supposing that the original request is \text{SELECT * FROM patient}, in the final query there are three different possibilities:

\[
\begin{align*}
\text{SELECT * FROM patient} \\
\text{MINUS} \\
\text{SELECT * FROM patient WHERE disease IN} \\
 \quad (\text{SELECT name FROM disease WHERE dangerousness = HIGH})
\end{align*}
\]

or

\[
\begin{align*}
\text{SELECT * FROM patient} \\
\text{MINUS} \\
\text{SELECT patient.* FROM patient p, disease d} \\
\quad \text{WHERE p.disease = d.name AND d.dangerousness = HIGH}
\end{align*}
\]

or

\[
\begin{align*}
\text{SELECT patient.* FROM patient p, disease d} \\
\quad \text{WHERE p.disease = d.name AND NOT(d.dangerousness = HIGH)}
\end{align*}
\]

Of course for efficiency reasons the second and third option are the selected, in this work the third option is the used in the algorithm. We have to take in to account that in order to perform the MINUS, the columns accessed in both select statements must be the same and with the same order. Finally the last option is the implemented one in this work.

- How the rule is evaluated:

1. Extract from the FROM clause the database and table to be accessed.
2. Extract from the SELECT clause the columns to be accessed.
3. Extract from the rule the database and table to be protected.
4. If the database and table extracted from the SELECT statement do not match with the database and table extracted from the deny rule, then the access is allowed. If they match continue with the query modification.

- Query modification for the second option:
1. Extract from the rule body the access conditions and the new table(s) to be accessed.

2. Add a MINUS clause to the original query.

3. Add a SELECT statement to the right operator of the MINUS clause.

4. Add the same accessed columns from the original SELECT clause to the new SELECT clause.

5. Add to the FROM clause the new table(s) to be accessed, as well as the original table.

6. Add a WHERE clause to the new SELECT statement (if necessary).

7. Add the conditions separated with an AND operator to the WHERE clause from the new SELECT statement.

- Query modification for the third option:

1. Extract from the rule body the access conditions and the new table(s) to be accessed.

2. Add to the FROM clause the new table(s) to be accessed.

3. Add a WHERE clause to the query.

4. Add the conditions connected with an AND operator to the new WHERE clause.

5. Extract from the head rule the column with the X variable and from the body rule the column with the X variable and create a new condition where both extracted columns are equals "=".

6. Add a WHERE clause (if necessary).

7. Add the new condition to the WHERE block and concatenate it with a logical and in case of the WHERE contains conditions.

8. Add the negation of the rest of conditions extracted from the rule body with an logical and.

**Denying access to cells**

10. Protecting a cell based on the table information.

```sql
deny access to db(medical, patient, telephone, [position = First Minister])
```
In the cell protection are different possibilities to protect the access. The first is to follow the same policy that databases use to protect a column, that is, when the access to a one cell is not allowed then the access to whole row is not allowed. In such case the cell protection is the same as the row protection, and this is not the idea in this work. Other possibility is to use the OVPD assigning the corresponding policy to the column table where the protected cells are, in such case when the query is executed the user get a NULL value in the protected cells, unfortunately this solution is only available in Oracle and is quite complex to use. Other option is to create a new view of the table where the protected cells values are NULL, this option will be quite expensive if there are several rules that protect access to cells. And the last one is to modify the original query using databases flow functions, such the CASE function in the following way:

```sql
SELECT name,
    CASE WHEN position = First Minister THEN NULL ELSE telephone END,
    address,
    position,
    disease
FROM patient p
```

- Here the actions are the same for each query request.
- The result will be all the patient data with an empty value for the not allowed cells.
- How the rule is evaluated:

1. Extract from the FROM clause the database and table to be accessed.
2. Extract from the SELECT clause the columns to be accessed.
3. Extract from the rule the database, table and columns where the cells are.
4. If the database, table and column to be accessed in the select statement do not match with the database, table and columns where the protected cells are, then the access is allowed. If not continue with the query modification.

- Query modification:

1. Extract from the rule head the column where the protected cells are (telephone).
2. Extract from the rule body the columns where the conditions that protect the cells are (position).
3. Replace the column where the protected cell belongs from the SELECT statement with a CASE clause.
4. Add to the new CASE clause a WHEN clause with the protection conditions.
5. Add to the new WHEN clause inside the CASE clause the THEN NULL clause.
6. Add at the en of CASE clause a ELSE clause.
7. Add to the new ELSE the column name where the protected cells belong.

8. Add a END CLAUSE to the CASE clause.

11. Protecting a cell based on other table information.

\[
\text{deny access to } \text{db(medical, patient, disease, [disease = X]) if} \\
\text{dbquery(medical, disease, [name = X, dangerousness = High])}
\]

Following the same idea of the last case studied at above point 10, the final query looks like

\[
\text{SELECT name, telephone address, position,} \\
\text{CASE WHEN d.name = p.disease AND d.dangerousness = HIGH THEN ""} \\
\text{ELSE p.disease END,} \\
\text{FROM patient p, disease d} \\
\text{WHERE p.disease = d.name}
\]

- Here the actions are the same for each query request.
- The result will be all the patient data with an empty value for the not allowed cells.
- How the rule is evaluated:

  The query evaluation is the same as the previous case.

- Query modification:

  The query modification is the same as the previous case where moreover the new accessed table is added to the FROM clause, and the join condition is extracted as follows:
  1. Extract from the rule head and the rule body the columns with the X variable, and create a new condition where both extracted columns are equals ",=".
  2. Add a WHERE clause if necessary.
  3. Add the new condition to the WHERE clause connected with an AND with the other conditions.

**Allowing access to databases**

12. Allowing a database without conditions.

\[
\text{Allow access to } \text{db(medical, _, _, _)}
\]

- Here the actions are the same for each query request.
• The request query is *not modified* and if there are no more rules the query is executed.
• How the rule is evaluated:

1. Extract from the FROM clause the database to be accessed.

2. Extract from the rule the database to be protected.

3. If the database extracted from the query and the database extracted from the deny rule do match, then the access is allow. If not continue.

4. The user is not allowed to access the database and an empty value is returned.

13. Allowing a database with conditions.

\[
\text{Allow access to } \text{db(medical, \_\_\_, \_\_\_, \_\_) if } \\
\text{dbquery(system, database, [name = medical, access = allowed])}
\]

• Here the actions are the same for each query request.
• The request query is *not modified* and and if there are no more rules the query is executed.
• How the rule is evaluated:

1. Extract from the FROM clause the database to be accessed.

2. Extract from the rule the database to be protected.

3. If the database extracted from the query and the database extracted from the deny rule do match, then the access is allow. If not continue continue with asking the database.

• Asking the database.

The question to the database is as follows

\[
\text{SELECT * FROM system.database d WHERE d.name = medical AND d.access = allowed}
\]

If the result is not an empty set then the access to the database is allowed and the system will continue checking the permissions. If the result is an empty set then the access is denied.

• How the query question is generated

1. A SELECT statement is created requesting the whole information with the * sign.

2. In the FROM clause is added the table where the rule body conditions are based.
3.2. HIGH LEVEL RULES EVALUATION

3. A WHERE clause is added to the SELECT statement.

4. The conditions extracted from the rule body are added to the new WHERE clause.

5. The query is executed with full access to the tables where the query access.

Allowing access to tables

14. Allowing a table without conditions.

Allow access to db(medical, patient, _, _)  
- Here the actions are the same for each query request.
- The request query is not modified and and if there are no more rules the query is executed.
- How the rule is evaluated:
  1. Extract from the FROM clause the database and table to be accessed.
  2. Extract from the rule the database and table to be protected.
  3. If the database and table extracted from the query and the database and table extracted from the allow rule match, then the access is allow. If not continue.
  4. The user is not allowed to access such table in such database and an error message is returned to the user.

15. Allowing a table with conditions.

Allow access to db(medical, patient, _, _) if dbquery(medical, patient, [name = unknown, disease = cancer])  
- Here the actions are the same for each query request.
- The request query is not modified and and if there are no more rules the query is executed.
- How the rule is evaluated:
  1. Extract from the FROM clause the database and table to be accessed.
  2. Extract from the rule the database and table to be protected.
  3. If the database and table extracted from the query and the database and table extracted from the deny rule match, then the access is allow. If not continue asking the database.
• Asking the database:

The question to the database is as follows

```
SELECT * FROM system.patient p WHERE p.name = unknown AND d.disease = cancer
```

If the result is not an empty set then the access to the table is allowed and the system will continue checking the permissions. If the result is an empty set then the access is denied.

• How the query question is generated

1. A SELECT statement is created requesting the whole information with the * sign.

2. In the FROM clause is added the table where the rule body conditions are based.

3. A WHERE clause is added to the SELECT statement.

4. The conditions extracted from the rule body are added to the new WHERE clause.

5. The query is executed with full access to the tables where the query access.

**Allowing access to columns**

16. Allowing a column without conditions.

Allow access to db(medical, patient, disease, _)

(a) The request is `SELECT * FROM patient`.

- The result will be the allowed column.
- The query is modified and executed.
- The final query is:

  ```
  SELECT disease FROM medical.patient
  ```

- How the query modification works:

  1. Extract from the rule head the allowed column.

  2. Change from the SELECT statement the * sign for the allowed column.

(b) The access request the column affected by the rule as well as other columns, or instance `SELECT name, disease FROM patient`, where the name column is supposed to be allowed.

- The result is the requested query.
3.2. HIGH LEVEL RULES EVALUATION

- The query is not modified and executed, and an error is returned to the user.

17. Allowing a column based on the table information.

Allow access to db(medical, patient, disease, _) if
dbquery(medical, patient, [disease = cancer])

The access is allowed to the whole column if the conditions are fulfilled. Again in this situation it is necessary to ask to the database whether the conditions are fulfilled, in such case the access to the whole column is allowed, and denied in any other case.

(a) Case of all access is requested with SELECT * FROM patient

- Following the same policy as in protecting a column explained in point 5, the result will be as in the above case, an error message or the column information depending on the rule body conditions.
- The query is modified and executed in case the conditions are satisfied and not modified neither executed in any other case.
- If the conditions are fulfilled then the query is modified and looks as follows
SELECT disease FROM patient
- After the usual resource access checking the following query is executed in the database
SELECT * FROM patient p WHERE p.disease = cancer
If the result is not an empty set, the access will be allowed and the result will be the column information. If the result set is an empty set the access, the access is not allowed and the result is an empty set. If the result is not an empty set then continue with the query modification.

- Query modification

1. Extract the allowed column to access from the rule head.

2. Change the * from the select clause for the allowed column.

(b) Case of the query requests some columns where the protected column figures, and supposing that the access to the other columns are allowed.

- The result is the original request result or an error message, depending whether the conditions from the rule body are satisfied or not.
- The query is the same as the original, it is not modified and maybe not executed.
- After the usual resource access checking, the following query is executed in the database in order to ask if the conditions are fulfilled.
SELECT * FROM patient p WHERE p.disease = cancer
If the result is not an empty set, the access will be allowed and the result will be the column information. If the result set is an empty set the access, the access is not allowed and the result is an empty set.
• How is the query constructed.

1. A SELECT statement is created requesting the whole information with the * sign.

2. In the FROM clause is added the table where the rule body conditions are based.

3. A WHERE clause is added to the SELECT statement.

4. The conditions extracted from the rule body are added to the WHERE clause.

5. The query is executed with full access to the tables where the query access.

18. Allowing a column based on other table information.

allow access to db(medical, patient, disease, _) if
dbquery(medical, patient, [disease = X]) and
dbquery(medical, disease, [name = X, dangerousness = High])

The only different with the above point is the query that ask to the database whether the conditions are satisfied. Such query is constructed as explained in the before explained case 9.

SELECT * FROM patient p, disease d
WHERE p.disease = d.name AND d.dangerousness = High

Allowing access to rows

19. Allowing a row based on the table information.

allow access to db(medical, patient, _, [position = First Minister])

• The result will be all the patient data where the position is equal to the First Minister.
• The query is executed and modified.
• In this case after the usual resource access checking, the result query will be:
  SELECT patient.* FROM medical.patient WHERE position = First Minister
• Query modification:
  1. Extract from the rule the access condition.
  2. Add a WHERE clause to the original query.
  3. Add the condition to the new WHERE clause.
20. Allowing a row based on other table information.

allow access to db(medical, patient, _, [disease = X]) if
dbquery(medical, disease, [name = X, dangerousness = High])

- The result will be all the patients data which patient’s disease is considered with HIGH dangerousness.
- The query is executed and modified.
- The final query after the usual object access check is:

```
SELECT * FROM patient p, disease d
WHERE p.disease = d.name AND d.dangerousness = HIGH
```

- Query modification

1. Extract from the rule the access conditions and the new table where the conditions are based.

2. Add to the FROM clause the new table(s).

3. Add a WHERE clause.

4. Add to the new WHERE clause the conditions before extracted as well as the join condition.

To obtain the join condition, from the rule head and from the rule body the the column with the variable X are extracted and conected with the operator ”=”.

Allowing access to cells

21. Allowing a cell based on the table information.

allow access to db(medical, patient, telephone, [telephone = X]) if
dbquery(medical, patient, [telephone = X, position = First Minister])

- The result will be all the patient data where the column telephone has an empty value in all the rows where the position value is different from First Minister.
- The query is executed and modified.
- In this case after the usual resource access checking, the result query will be:

```
SELECT name,
    CASE WHEN position = First Minister THEN telephone ELSE NULL END,
    address,
    position,
    disease
FROM patient p
```
• Query modification:

1. Extract from the rule head the column where the protected cells are.

2. Extract from the rule body the columns where the conditions that protect the cells are.

3. Replace the column where the protected cell belongs from the SELECT statement with a CASE clause.

4. Add to the new CASE clause a WHEN clause with the allowing conditions.

5. Add to the new WHEN clause inside the CASE clause the THEN clause with the column name where the protected cells belong.

6. Add at the end of CASE clause a ELSE clause.

7. Add to the new ELSE an empty value "".

8. Add a END CLAUSE to the CASE clause.

22. Allowing a cell based on other table information.

allow access to db(medical, patient, disease, [disease = X]) if
dbquery(medical, disease, [name = X, dangerousness = Low])

Basically this case is as the last one above explained with some additions.

(a) In case of the query request only access to the telephone column, that is, select telephone from medical.patient.

• The result will be all the telephone numbers from the patients which position is equal to the First Minister.

• The final query after the usual match check will be:

SELECT disease FROM medical.patient p, medical.disease d
WHERE p.position = First Minister AND
p.disease = d.name AND d.dangerousness = High

• Query modification:

1. Extract from the rule the access conditions.

2. Extract from the rule the additional table to be accessed.

3. Add to the FROM clause the additional table.

4. Add a WHERE clause to the original query.

5. Add the conditions to the new WHERE clause.
6. Add the join condition as explain in case 9.

Conclusions

After analysing the before studied cases, four different patterns are extracted for each different modality, allow or deny. Let us concentrate in each requested column from the SELECT statement, these four different patterns of actions to execute when the access to a column is requester are the following:

1. Allow/Deny the access to a whole column without conditions.
2. Allow/Deny the access to a whole column based on conditions.
3. Allow/Deny the access to some rows of the column based on conditions.
4. Allow/Deny the access to some cells of the column based on conditions.

At first look patterns 3 and 4 maybe seem the same cases, however when for the same conditions the access is denied, the first one does not return a value to the user because the whole row is not allowed, and the second one returns a NULL in each denied cell from the column. The below table shows an example to clarify the difference, where the allowed accessed values are A, C and E.

<table>
<thead>
<tr>
<th>Column values</th>
<th>Returned value for case 3</th>
<th>Returned value for case 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>null</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: differences between cases 3 and 4

Pattern number one has been extracted from the following cases; when the access to a whole database, table or column is allowed/denied. Pattern number two has been extracted from the following cases; when the access to a whole database, table or column is allowed/denied based on some conditions. Pattern number three has been extracted from the following cases; when the access to some rows to the table where the column belongs are allowed/denied based on some conditions. Pattern number four has been extracted from the following cases; when the access to some column cell are allowed/denied based on some conditions.

Now let us describe the different actions to perform for each different extracted pattern. The actions will be different depending on whether the requested column comes from the list of columns of the SELECT block or whether it comes from the tipical start sign of the SELECT block.

1. Allow/Deny the access to a whole column without conditions.

   If the column comes from a list of columns and the column is denied or not allowed the query will fails. If the column comes from the start and the column is as well denied or not allowed then the column will be not accessed, that is should be deleted from the SELECT block. Otherwise the column is allowed and no actions are performed.
2. Allow/Deny the access to a whole column based on conditions.

These cases require a question to the database in order to check if the conditions are satisfied or not. After the execution of the question we will now if the access to the column is allowed or not. If the access is denied and the column comes from a list of columns the query will fail. If the column comes from the start then the column will be deleted again from the SELECT block.

3. Allow/Deny the access to some rows of the column based on conditions.

These cases add to the WHERE block of the original query some constraints. When the conditions come from an allow rule, they will be simply added to the where clause. If the conditions come from a deny rule, these conditions are at first denied and afterwards added to the WHERE clause.

4. Allow/Deny the access to some cells of the column based on conditions.

When a rule allow or deny the access to some cells of the requested column, the column is replaced from the SELECT block for a CASE expression.

The case expression from the fourth pattern is here studied in detail. There are three possibilities, when only allow rules affect the requested column, when only deny rules affect the requested column and when both, allow and deny rules affect the requested column. In the first case only the cells which values satisfied the conditions are returned to the user, for the rest of cells the user will read a null value, this corresponds with the following code

```sql
CASE column_name
WHEN conditions1 THEN column_name
.....
WHEN conditionsN THEN column_name
```

The second case corresponds with the next code where the user read a null value for the denied cells and for the rest he will read the correct value.

```sql
CASE column_name
WHEN conditions1 THEN NULL
.....
WHEN conditionsN THEN NULL
WHEN column_name THEN column_name
```

In the last case, where denied and allow rules are together, the denied conditions must be at the beginning of the CASE expression, in such way the denied values will take precedence over the allow values. The next code described the explained

```sql
CASE column_name
WHEN deny_conditions1 THEN NULL
.....
WHEN deny_conditionsN THEN NULL
WHEN allow_conditions1 THEN column_name
.....
WHEN allow_conditionsN THEN column_name
```

The CASE expression in the SQL language as a default action returns a null value if there is no default case, that is, when there is no ELSE condition a null value is returned.
Adding the WHERE clause

All the studied cases are simple queries without WHERE clause. Right now, the patterns have been extracted from the simple queries and we now the actions to perform in each different situation. Let us analyse when a WHERE clause is added to these studied queries.

The query WHERE block contains as well columns, although these columns are not returned to the user, they must be protected as well. If we take a look to what the databases do, when a user has no access to a certain column from the WHERE block an error is returned to the requester.

As with the columns from the SELECT block, the columns from the WHERE block have as well four different possible ways to be protected, the actions to perform for each different case are similar to the described ones for the column from the SELECT block, however there are some changes, let us described the different actions.

1. Allow/Deny the access to a whole column without conditions.

   If the whole column is allowed there are no actions to perform and the check continues. If the whole column is denied, the query fails.

2. Allow/Deny the access to a whole column based on conditions.

   These cases requires a question to the database in order to check if the conditions are satisfied or not. After the execution of the question we will now if the access to the column is allowed or not. If the access is denied the query fails if not the check continues.

3. Allow/Deny the access to some rows of the column based on conditions.

   These cases add to the WHERE block of the original query some constraints. When the conditions come from an allow rule, they will be simply added to the where clause. If the conditions come from a deny rule, these conditions are at first denied and afterwards added to the WHERE clause.

4. Allow/Deny the access to some cells of the column based on conditions.

   The actions to perform here are the same as for the third case, that is if a cell from WHERE block column is not allowed to be accessed then the whole row is protected.
3.3 Policy Based Query Expansion

This section explains an algorithm to perform the policy-based access control. All the rules evaluations before done in the last section are here take in to account. Given a SQL query, each column that appears on the SELECT or WHERE blocks from the query are accessed as well as each table and database that appear in the FROM block from the query. The approach consist of analyzing the set of elements accessed and restricting them according to the policies in force. Contextual conditions, for instance time constraints or conditions based on requester’s properties are evaluated by some policy engine, while other constraints are added to the original query and enforced during query processing.

Before starting with the algorithm and its description some terms and concepts are introduced in order to understand and be able to perform the algorithm.

SQL Queries

The first definition is the SQL query that the algorithm is allowed to process. As said during the analyze section, the only queries accepted are the ones based on the read permission, that is, the SELECT queries. This SELECT accepted queries are able to contain WHERE clauses with conditions, and of course the access to more than one table in the FROM block is allowed as well. Therefore in the modified queries the CASE control flow function is as well supported.

- SELECT expr1 FROM expr2 [WHERE BE_or]
  - expr1 = ([table.]column|case) A set of columns or case expressions
  - case = (CASE (WHEN BE_or THEN ([table.]column|value)) ELSE ([table.]column|value) END)
  - expr2 = ([database.]table) A set of tables with the possibility of the respective database.
  - BE_or = (expression [AND expression]* ) [OR BE_or], a set of logical disjunction expressions, where each element (expression) of this set is a set of logical conjunctions.
  - expression = ( ([table.]column|value) relation ([table.]column|value) ), this expression it is as well possible to be represented with the path expression definition (see below) in the way as follows
    - relation is a relation such as =, ≠, >, <, ≥, ≤, etc...

This SQL query description represents all the possible queries that the system accept as a request, in case of query modification, the modified query is also represented with the described syntax.

- Column Expression, col = (db, tb, col, source, BE_and, allowed), where
  - db represents a database, tb a table and col a column.
  - source is a a boolean, when it’s value is TRUE means that the column comes from a select expression with the * sign.
  - BE_and is a set of expressions connected with a logical conjunction.
  - allowed is a boolean, when it’s value is TRUE means that the access to the column col is allowed otherwise is denied.
The Column expression is going to be very useful in the algorithm, the idea is to represent each column that appears in the SELECT, CASE and WHERE blocks from the described SQL queries. The two last expressions $BE_{and}$ and allowed are used to represent the columns from the CASE block, therefore these values for the rest of columns have an empty value here.

Whenever the following representation db.tb.col appears during the rest of the documents has the meaning of a column expression where the three last fields of the column expression are not taken into account.

- Representing the Case Expression.

Is not quite usual to find a case expression on the select block of the requested queries, nevertheless when a query is modified in order to protect some cells, the query modification process does it with this control flow expression. Therefore the SQL request are not allowed to have a CASE expression inside the SELECT block, they are reserved for the modified queries, exactly only when the access to some cells from a columns is allowed. In such case the column access is replaced for a CASE expression in the SELECT block.

The following case expression inside the SELECT statement,

\[
\text{SELECT telephone, address,}
\begin{align*}
\text{ CASE WHEN disease = cancer \ AND NULL} \\
\text{ WHEN disease = AIDS \ AND NULL} \\
\text{ WHEN disease \ AND \ disease} \\
\text{ END}
\end{align*}
\text{FROM patient}
\]

is represented with a set of Column Expression as follows:

\[
\text{colSET} = \left( (m.p.address,FALSE,\emptyset,\emptyset) \right)
\left( (m.p.address,FALSE,\emptyset,\emptyset) \right)
\left( (m.p.disease,FALSE,(m.p.disease = cancer),FALSE) \right)
\left( (m.p.disease,FALSE,(m.p.disease = AIDS),FALSE) \right)
\left( (m.p.disease,FALSE,(m.p.disease),TRUE) \right)
\]

where there is no representation for the default case ELSE because is it not necessary during the query modification, see 3.2 conclusions section in page 62. Nevertheless this could be represented giving to the boolean expression $BE_{and}$ an empty value $BE_{and} = \emptyset$.

Another possible representation for the same CASE example from above is to order in a different way the set of elements inside the colSET because whenever the last record allowed of the column expression has the FALSE value, the conditions from the $BE_{and}$ must appear before the conditions from columns with last parameter allowed with TRUE value. The reason is due to the CASE control flow expression that says that the instructions of the first condition evaluated with a TRUE are executed.

- SQL query, $q = (colSel, colWhere, QBE_{and}|QBE_{and})$, where
  - ColSel and ColWhere are a set of columns col.
- $QBE_{or}$ is a set of logical disjunction expressions, where each element (expression) of this set is a set of logical conjunctions. This element has the same set of conditions as the set $BE_{or}$ from the original query.

- $QBE_{and}$ is as set of logical disjunction expression, where each expression element is a $QBE_{or}$.

- $\exists col \in expr_1 : \exists \notin \text{colSel}$

- $\exists col \in BE_{or} : \exists \notin \text{colWhere}$

- $\text{colWhere}$ is the set of different columns that appear in the $QBE_{or}$

This is the SQL representation that is going to be used in during the algorithm, as above described there is no column from the original SQL request no represented in the colSel or colWhere (columns that appear in the SELECT, WHERE or CASE blocks), therefore, all the accessed tables and databases are as well represented in the set $colSel \cup colWhere$.

During the query modification process the elements $QBE_{and}$ and $\text{colWhere}$ are expanded in order to restrict the query access to the resources, and the element $\text{colSel}$ could be as well expanded or reduced.

Each request query is as first represented with the definition $q = (\text{colSel}, \text{colWhere}, QBE_{or})$, that is, the boolean expressions from the WHERE block are represented with a set of logical disjunction expressions. However after the query modification process each query is represented with the definition $q = (\text{colSel}, \text{colWhere}, QBE_{and})$, that is, now the boolean expression from the WHERE block are now represented as a set of logical conjunctions.

**Example 1.** Supposing the table patient is represented with the tuple patient(id, name, age, telephone, position, disease) the following query try to retrieve all the patient information where patients are students and older than 18 years old or patients are retired with more than 65 years old.

```
SELECT * FROM patient p
WHERE (p.position = student AND p.age > 18) OR (p.position = retired AND p.age > 65)
```

has a query representation as follows (where letter ”m” represents ”medical”, letter ”p” represents ”patient”, letter ”F” represents the value FALSE and letter ”T” represents the value TRUE)

$q = (\text{colSel}, \text{colWhere}, QBE_{or})$ where

\[
\text{colSel} = ( (m, p, id, T, \emptyset, \emptyset),
(m, p, name, T, \emptyset, \emptyset),
(m, p, age, T, \emptyset, \emptyset),
(m, p, telephone, T, \emptyset, \emptyset),
(m, p, position, T, \emptyset, \emptyset),
(m, p, disease, T, \emptyset, \emptyset))
\]

\[
\text{colWhere} = ( (m, p, position, F, \emptyset, \emptyset),
(m, p, age, F, \emptyset, \emptyset))
\]

\[
QBE_{or} = ( (((m, p, position, F, \emptyset, \emptyset) = student), ((m, p, age, F, \emptyset, \emptyset) > 18))),
(((m, p, position, F, \emptyset, \emptyset) = retired), ((m, p, age, F, \emptyset, \emptyset) > 65)))
\]
3.3. POLICY BASED QUERY EXPANSION

Example 2. A possible query modification example for the above example is here introduced, where only the columns name, position and disease are accessed in the SELECT block. The column disease from the original SELECT block has been substituted for a CASE, and the columns position and age are accessed in the WHERE block:

```sql
SELECT name,
    CASE WHEN p.disease = cancer THEN NULL
         WHEN p.disease = AIDS THEN NULL
         WHEN p.disease THEN disease END,
    position
FROM patient p
WHERE (p.position = student AND p.age > 18) OR (p.position = retired AND p.age > 65)
```

and his query representation is as follows (where letter "m" represents "medical", letter "p" represents "patient", letter "F" represents the value FALSE and letter "T" represents the value TRUE)

\[ q = (\text{colSel}, \text{colWhere}, \text{QBE}_{\text{and}}) \]

where

- \( \text{colSel} = ( (m, p, name, F, \emptyset, \emptyset), \)
  \( (m, p, disease, F, (\{(m, p, disease, FALSE, \emptyset, \emptyset) = cancer\}, F)) \)
  \( (m, p, disease, F, (\{(m, p, disease, FALSE, \emptyset, \emptyset) = AIDS\}, F)) \)
  \( (m, p, disease, F, (m, p, disease, FALSE, \emptyset, \emptyset), T)) \)
  \( (m, p, position, F, \emptyset, \emptyset) ) \)

- \( \text{colWhere} = ( (m, p, position, F, \emptyset, \emptyset), \)
  \( (m, p, age, F, \emptyset, \emptyset) ) \)

- \( \text{QBE}_{\text{and}} = ( ( ((m, p, position, F, \emptyset, \emptyset) = student), ((m, p, age, F, \emptyset, \emptyset) > 18))) , \)
  \( ((m, p, position, F, \emptyset, \emptyset) = retired), ((m, p, age, F, \emptyset, \emptyset) > 65) )) \)

where the element \( \text{QBE}_{\text{and}} \) has a unique element inside, the \( \text{QBE}_{\text{or}} \) from example 1.

### Policies over Relational Databases

As well as the before new format described for representing the SQL query request, here is described a format to represent the policies over rules that are going to protect or allow the access to the data.

- \( db = (db, tb, col, BE_{\text{and}}) \), where
  - \( BE_{\text{and}} = (expression[\text{AND} expression]^*]) \),
    that is, \( BE_{\text{and}} \) is a set of expressions connected with a logical and.

- \( dbquery = (db, tb, col, BE_{\text{and}}) \)

A policy rule \( pol \) is a rule of the form:

- \( modality/db \leftarrow cp_1, \ldots, cp_n, dbquery_1, \ldots, dbquery_p \), where
  - \( modality \in \{\text{allow, deny}\} \)
are contextual predicates, for instance related to time, location, possession of credential, etc . . .

In the following the expression $H(\text{pol})$ will refer to the head of $\text{pol}$ and $B(\text{pol})$ to the body of $\text{pol}$.

**Example 3.** The following policy rule table have contextual conditions as well as conditions based on the data. First rule allow access to the whole patient rows to Physicians and Cardiologist whenever the position value of the rows are different of First Minister. The second rule allow the access to the telephone column only for a range of time. The third rule denied the access to the disease cells whenever the cells values are cancer. Rule number four denied the access to the diseases cells whenever such values are considered with a High level of dangerousness. The last policy allow the access to the whole id column whenever exists in such column a value higher than five.

<table>
<thead>
<tr>
<th>No</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1$</td>
<td>allow access to $db(\text{medical}, \text{patient}, _, [\text{pos} \neq \text{FirstMinister}])$ if Requester = Physician OR Requester = Cardiologist</td>
</tr>
<tr>
<td>$r_2$</td>
<td>deny access to $db(\text{medical}, \text{patient}, \text{telephone}, _)$. if\ Requester = Physician AND\ $09:00 &lt; \text{Time AND Time} &lt; 12:00$</td>
</tr>
<tr>
<td>$r_3$</td>
<td>allow access to $db(\text{medical}, \text{patient}, \text{disease}, [\text{disease} \neq \text{cancer}])$ if Requester = Physician</td>
</tr>
<tr>
<td>$r_4$</td>
<td>deny access to $db(\text{medical}, \text{patient}, \text{disease}, [\text{disease} = \text{X}])$ if\ $\text{dbquery(\text{medical}, \text{disease}, [\text{name} = \text{X}, \text{dangerousness} = \text{High}])$ AND Requester = Physician</td>
</tr>
<tr>
<td>$r_5$</td>
<td>allow access to $db(\text{medical}, \text{patient}, \text{id}, _)$. if\ Requester = Physician AND\ $\text{dbquery(\text{medical}, \text{patient}, [\text{id} &gt; 5]}$</td>
</tr>
</tbody>
</table>

and his policy representation is as follows (where letter "$m$" represents "$\text{medical}$", letter "$p$" represents "$\text{patient}$", letter "$F$" represents the value FALSE and letter "$T$" represents the value TRUE)

\[
\begin{align*}
\text{allow}(m, p, _, [\text{position} \neq \text{FirstMinister}]) & \leftarrow (\text{Requester} = \text{Physician}) \\
\text{allow}(m, p, _, [\text{position} \neq \text{FirstMinister}]) & \leftarrow (\text{Requester} = \text{Cardiologist}) \\
\text{deny}(m, p, \text{telephone}, _) & \leftarrow (\text{Time} > 09:00), (\text{Time} < 12:00), (\text{Requester} = \text{Physician}) \\
\text{allow}(m, p, \text{disease}, [\text{disease} \neq \text{cancer}]) & \leftarrow (\text{Requester} = \text{Physician}) \\
\text{deny}(m, p, \text{disease}, [\text{disease} = \text{X}]) & \leftarrow \\
& (\text{Requester} = \text{Physician}), \text{dbquery}(m, \text{disease}, [\text{name} = \text{X}, \text{dangerousness} = \text{High}]) \\
\text{allow}(m, p, \text{id}, _) & \leftarrow (\text{Requester} = \text{Physician}), \text{dbquery}(m, p, [\text{id} > 5])
\end{align*}
\]

**Policy Evaluation**

During the algorithm a policy evaluation is performed, this policy evaluation basically is a check whether a policy is applicable to the SQL request or not. To understand better how it works here are some definitions, the first one says when a policy is applicable to a column. The second one introduce an important data structure used along the algorithm, and the last definition says when a requester query fails returning an empty value to the user.
3.3. POLICY BASED QUERY EXPANSION

Definition 1 (Policy Applicability). Given a column expression \( \text{col} \), a set of policies \( P \) and a time-dependent state \( \Sigma \) (which in or case determines at each instant the extension of contextual predicates), a policy \( \text{pol} \in P \) is applicable to \( e \) (denoted by \( \hat{\text{pol}}(\text{col}) \)) iff

- \( \sigma' = \text{mgu(}\text{col}, H^p(\text{pol})) \), where mgu is the most general unifier
- \( \exists \sigma, \sigma'' : \sigma = \sigma' \sigma'' \land \forall cp_i \in B(\text{pol}), P \cup \Sigma \models \sigma cp_i \)
- and its application is a function \( \text{col, pol} P, \Sigma \rightarrow (BE_{\text{and}}, \text{action}, BE_{\text{and}}^J) \), where
  - \( \text{action} = (\text{col}\mid \text{row}\mid \text{cell}) \)
  - \( BE_{\text{and}} \) and \( BE_{\text{and}}^J \) are the set of constraints for accessing the given column. \( BE_{\text{and}} \) contains the constraints without the join condition and \( BE_{\text{and}}^J \) contains the join conditions if the conditions are based on other table as the original column is. (e.g. in the rule 4, \( BE_{\text{and}} = (\text{medical.disease.dangerousness} = \text{High}) \) and \( BE_{\text{and}}^J = (\text{medical.patient.disease} = \text{medical.disease.name}) \))

Intuitively, a policy \( \text{pol} \) is applicable to \( \text{col} \) if the object the policy is protecting unifies with the path expression \( \text{col} \) and all the contextual predicates and bound boolean expressions are satisfied.

Definition 2 (Policy Application). The Policy Application variable is a variable used in the algorithm to store the result of all the policy applications that are applicable to a given path expression \( \text{pa} \) and a set of policies \( P \), it has the following syntax

\[
\text{pa} = (\text{col}, BE_{\text{or, col}}, BE_{\text{or, row}}, BE_{\text{or, cell}})
\]

- \( \text{col} \) is a column expression.
- \( BE_{\text{or, col}}, BE_{\text{or, row}}, BE_{\text{or, cell}} \), are a set of conjunction expressions, where each set connects with the following set with a disjunction. Each boolean expression set represents the four different classified groups explained in 3.2 at conclusions section in page 62.

Definition 3 Given a query \( q = (\text{colSel}, \text{colWhere}, \text{QBE}_{\text{or}}) \) and defining \( \text{colSet} \) as the union between \( \text{colSel} \) and \( \text{colWhere} \), a set of policy rules \( P \) and a state \( \Sigma \), let say that \( q \) fails if either of the following two conditions hold:

- \( \exists \text{col} \in \text{colSet} : \exists \text{pol} \in P, H(\text{pol}) = \text{allow(db)} \land \text{pol(}\text{col}) \)
- \( \exists \text{col} \in \text{colSet} : \exists \text{pol} \in P, H(\text{pol}) = \text{deny(db)} \land \text{pol(}\text{col}) \land \text{pol} P, \Sigma \rightarrow (\emptyset, \emptyset) \)

Definition 4 (Well Formed Policy) Given a set of policies \( P \), let say that \( P \) is a well formed policy if the following condition is hold:

\[ \forall p_1 \in P : H(p_1) = \text{modality(db, tb, col, \emptyset)}, \exists p_2 \in P : H(p_2) = \text{modality(db, tb, col, X)} \]

where \( p_1 \) and \( p_2 \) are different policies inside the set of policies \( P \).

Lets try to explain the meaning of this definition with an example. Giving the two following policies:

1. Allow access to (medical, patient, telephone, _)
2. Allow access to (medical, patient, telephone, [telephone != 0000])

the first rule is considered as a more general rule than the second one because of the "_" at the last field. Therefore if in the set of policies exists the first rule, the second one is not allowed to be in the set of policies.
Algorithm: Policy Evaluation and Query Expansion

The algorithm is divided in three steps, each step is described with a different function. The Policy Application (PA) data structure above described is the most important variable used in the algorithm, in fact this structure is used twice, one for manage the deny policies and the other to manage the allow policies. In the first step, the policy prefiltering function fulfill such variables with information and may delete some columns from the field colSel from the original query. The second step, the database question function use the field BEor_col from such variables to check if full column access conditions are satisfied. The last step, the query modification function use the fields BEor_row and BEor_cell from the PA variables to modify the query.

Above the definition of such functions are introduced, afterward comes a description of such functions explaining how they work in order to a better understanding as well as some examples with the input and output of such functions.

Input:
a query \( q = (colSel, colWhere, QBE_{or}) \), a set of policy rules \( P \) and a state \( \Sigma \)

Output:
\( PA_{allow} \equiv \) a set of Policy Application (pa) derived from allow rules
\( PA_{deny} \equiv \) a set of Policy Application (pa) derived from deny rules
a query \( q' = (colSel, colWhere, QBE_{and}) \)

\[\text{policy_prefiltering}(q, P, \Sigma): \]

1:  //variables
2:  \( pa = (\emptyset, \emptyset, \emptyset, \emptyset) \), \( BE = \emptyset \), \( PA_{allow} = PA_{deny} = \emptyset \), \( P_{app} = \emptyset \)
3:  \( q'.colSel = q.colSel \)
4:  \( q'.colWhere = q.colWhere \)
5:  \( q'.QBE_{and} \cup = q.QBE_{or} \)
6:  //algorithm
7:  \( colSet = q.colSel \cup q.colWhere \)
8:  for all \( col \in colSet \) do
9:      //check allow policies
10:     \( P_{app} = \{pol | pol \in P \land H(pol) = allow(bd) \land \overline{pol(col)}\} \)
11:     if \( P_{app} = \emptyset \land \neg \text{col.source} \) then
12:        //no allow rule for such column that does not come from * (also columns
13:         //from the WHERE block included)
14:        return query failure
15:     else if \( P_{app} = \emptyset \land \neg \text{col.source} \) then
16:        //the column is no available in select *
17:        \( q'.col = \emptyset \)
18:        if \( q'.col = \emptyset \) then
19:           return query failure
20:     end if
21:    else if \( \exists pol \in P_{app} : col, pol \xrightarrow{P, \Sigma} (\emptyset, \emptyset) \) then
22:       //all objects matching col are allowed without conditions
23:       add\((PA_{allow}, (col, \emptyset, \emptyset, \emptyset))\)
24:     else
25:        \( pa = (col, \emptyset, \emptyset, \emptyset) \)
26:    end if

3.3. POLICY BASED QUERY EXPANSION

Algorithm 1: pre-filtering algorithm
Input:
PA_{allow} \equiv \text{a set of Policy Application (pa) derived from allow rules}
PA_{deny} \equiv \text{a set of Policy Application (pa) derived from deny rules}
a query \, q = (\text{colSel, colWhere, QBE}_{\text{and}})

Output:
PA_{allow} \equiv \text{a set of Policy Application (pa) derived from allow rules}
PA_{deny} \equiv \text{a set of Policy Application (pa) derived from deny rules}
a query \, q = (\text{colSel, colWhere, QBE}_{\text{and}})

database\_questions(PA_{allow}, PA_{deny}, q):

1: \text{BE}_{\text{and}} = \emptyset //a boolean expresion
2: \text{q}' = \emptyset //a sql query
3: \text{result} = \emptyset //an sql result
4: //check allow policies
5: \text{for all} \, \text{pa} \in PA_{allow} : \text{pa.BE}_{or,\text{col}} \neq \emptyset \text{ do}
6: \quad \text{if} \neg \text{pa.col.source} \text{ then}
7: \quad \quad \text{BE}_{\text{and}} \cup = \text{pa.BE}_{or,\text{col}}
8: \quad \text{else}
9: \quad \quad \text{q}'.\text{colSel} = *
10: \quad \quad \text{q}'.QBE_{\text{and}} \cup = \text{pa.BE}_{or,\text{col}}
11: \quad \quad \text{result} = \text{execute} \, q'
12: \quad \quad \text{if} \, \text{result} = \emptyset \text{ then}
13: \quad \quad \quad \text{PA}_{allow} = \text{pa}
14: \quad \quad \quad \text{PA}_{deny} = \text{pa}
15: \quad \quad \quad \text{q.selCol} = \text{pa.col}
16: \quad \quad \quad \text{if} \, q.selCol = \emptyset \text{ then}
17: \quad \quad \quad \quad \text{return} \, \text{query failure}
18: \quad \quad \text{end if}
19: \quad \quad \text{end if}
20: \quad \text{end if}
21: \text{end for}
22: \text{if} \, \text{BE}_{\text{and}} \neq \emptyset \text{ then}
23: \quad \text{q}'.\text{colSel} = *
24: \quad \text{q}'.QBE_{\text{and}} \cup = \text{BE}_{\text{and}}
25: \quad \text{result} = \text{execute} \, q'
26: \quad \text{if} \, \text{result} = \emptyset \text{ then}
27: \quad \quad \text{return} \, \text{query failure}
28: \quad \text{end if}
29: \text{end if}
3.3. POLICY BASED QUERY EXPANSION

30: // check deny policies
31: for all \( pa \in PA_{\text{deny}} \) : \( pa.BE_{or\_col} \neq \emptyset \) do
32:     if \( \neg pa.col.source \) then
33:         \( BE_{and\_\cup} = pa.BE_{or\_col} \)
34:     else
35:         \( q'.colSel = * \)
36:         \( q'.QBE_{and\_\cup} = pa.BE_{or\_col} \)
37:         result = execute \( q' \)
38:     if result \( \neq \emptyset \) then
39:         \( PA_{\text{allow}} = pa \)
40:         \( PA_{\text{deny}} = pa \)
41:         \( q.selCol = pa.col \)
42:         if \( q.selCol = \emptyset \) then
43:             return query failure
44:         end if
45:     end if
46: end if
47: end for
48: if \( BE_{and} \neq \emptyset \) then
49:     \( q'.colSel = * \)
50:     \( q'.QBE_{and\_\cup} = BE_{and} \)
51:     result = execute \( q' \)
52: return query failure
53: end if

Algorithm 2: database questions
**Input:**

a query \( q = (\text{colSel}, \text{colWhere}, QBE_{\text{and}}) \)

\( PA_{\text{allow}} \equiv \) a set of Policy Application (pa) derived from allow rules

\( PA_{\text{deny}} \equiv \) a set of Policy Application (pa) derived from deny rules

**Output:**

a query \( q = (\text{colSel}, \text{colWhere}, QBE_{\text{and}}) \)

query\_modification\((q, PA_{\text{allow}}, PA_{\text{deny}})\):

1: //variables declaration
2: \( \text{col} = \emptyset \) //an empty column
3: //allowed rows modifications
4: for all \( pa \in PA_{\text{allow}} : \) pa.BE\(_{or\_row}\) \( \neq \emptyset \) do
5: \( q.QBE_{\text{and}}\cup = \text{pa.BE}_{or\_row} \)
6: end for
7: //denied rows modifications
8: for all \( pa \in PA_{\text{deny}} : \) pa.BE\(_{or\_row}\) \( \neq \emptyset \) do
9: \( q.QBE_{\text{and}}\cup = \neg \text{pa.BE}_{or\_row} \)
10: end for
11: //allowed cells modifications
12: for all \( pa \in PA_{\text{allow}} : \) pa.BE\(_{or\_cell}\) \( \neq \emptyset \) do
13: \( \text{col} = (\text{pa.col.db}, \text{pa.col.tb}, \text{pa.col.col}, \text{FALSE}, \text{pa.BE}_{or\_cell}, \text{TRUE}) \)
14: \( q.selCol\_− = (\text{col.db}, \text{col.tb}, \text{col.col}, \_\_\_, \_\_) \)
15: \( q.selCol\cup = \text{col} \)
16: end for
17: //denied cells modifications
18: for all \( pa \in PA_{\text{deny}} : \) pa.BE\(_{or\_cell}\) \( \neq \emptyset \) do
19: \( \text{col} = (\text{pa.col.db}, \text{pa.col.tb}, \text{pa.col.col}, \text{FALSE}, \text{pa.BE}_{or\_cell}, \text{FALSE}) \)
20: \( q.selCol\_− = (\text{col.db}, \text{col.tb}, \text{col.col}, \_\_\_, \_\_) \)
21: \( q.selCol\cup = \text{col} \)
22: end for

**Algorithm 3:** query modification
3.3. POLICY BASED QUERY EXPANSION

policy_prefiltering

For each accessed column in the original query, all the applicable allow policies to such column are evaluated with 4 different possibilities; If there is no allow policy and the original request has no * symbol inside the SELECT block the query fails (line 11), see 3.2 conclusions section in page 62. If there is no allow policy and the original request has the * sign within the SELECT block the column will be deleted and not accessed in the modified query (of course if the query has no columns to be accessed fails) (line 15), see 3.2 conclusions section. If exists an allow policy that allow the access to such columns without restrictions, then the access to such column is allowed (line 18). And the last possibility, if there is at least one allow policy with some kind of conditions, each policy conditions are added to different fields from the PA data structure (line 24).

The different type of conditions are as follows; If the access to the whole column is protected with conditions, such conditions are stored in the $BE_{or, col}$ record (line 31). If the conditions refer to a row column inside a database, or the conditions refer to some cells of a column and such column appear only inside the WHERE block, such conditions are added to $BE_{or, row}$ record (line 33), see 3.2 adding the where section at page 62. If only some cells are protected (from the SELECT block columns), the conditions are added to the record $BE_{or, cell}$ (line 35). In case of the before conditions are based on other table where the protected column is, the join conditions are added to the $QBE_{and}$ (line 29).

With the deny rules the same process is followed taking into account that when there is no deny rule then there is no query failure returned (with the allow it returns a query failure). As said in section 3.1 Solving rule conflicts, deny by default is the policy to follow. This is performed in the allow section, whenever there is no allow rule for a given column the query fails (line 11).

Example 4. The below example shows how the function works, the above Example 1 is used as a SQL query input $q$, as well as the Example 3 as a set of policies $P$ and $\Sigma = (10 : 00, Requester = Physician)$ as a time dependant state.

Input

$$q = (colSel, colWhere, QBE_{or})$$  
where

$$colSel = ( (m,p, id, T, \emptyset, \emptyset), \quad (m,p, name, T, \emptyset, \emptyset), \quad (m,p, age, T, \emptyset, \emptyset), \quad (m,p, telephone, T, \emptyset, \emptyset), \quad (m,p, position, T, \emptyset, \emptyset), \quad (m,p, disease, T, \emptyset, \emptyset) )$$

$$colWhere = ( (m,p, position, F, \emptyset, \emptyset), \quad (m,p, age, F, \emptyset, \emptyset) )$$

$$QBE_{or} = ( (((m,p, position, F, \emptyset, \emptyset) = student), ((m,p, age, F, \emptyset, \emptyset) > 18))), \quad (((m,p, position, F, \emptyset, \emptyset) = retired), ((m,p, age, F, \emptyset, \emptyset) > 65)))$$

$$P = allow(m,p, \cdot, [position \neq FirstMinister]) \leftarrow (Requester = Physician)$$
allow(m, p, _ [position \neq \text{FirstMinister}]) \leftarrow (\text{Requester} = \text{Cardiologist})

deny(m, p, telephone, _) \leftarrow (\text{Time} > 09 : 00), (\text{Time} < 12 : 00), (\text{Requester} = \text{Physician})

allow(m, p, disease, [disease \neq \text{cancer}]) \leftarrow (\text{Requester} = \text{Physician})

deny(m, p, disease, [disease = X]) \leftarrow (\text{Requester} = \text{Physician}), \text{dbquery}(m, p, disease, \text{name} = X, \text{dangerousness} = \text{High}))

allow(m, p, id, _) \leftarrow (\text{Requester} = \text{Physician}), \text{dbquery}(m, p, [id > 5])

\Sigma = (10 : 00, \text{Requester} = \text{Physician})

Output

\[ q' = (\text{colSel}', \emptyset, QBE'_{\text{and}}) \]

\begin{align*}
\text{colSel}' &= ( (m, p, id, T, \emptyset, \emptyset), \\
& (m, p, name, T, \emptyset, \emptyset)) \\
& (m, p, age, T, \emptyset, \emptyset) \\
& (m, p, position, T, \emptyset, \emptyset) \\
& (m, p, disease, T, \emptyset, \emptyset))
\end{align*}

\begin{align*}
QBE'_{\text{and}} &= ( (((m, p, position, F, \emptyset, \emptyset) = \text{student}), ((m, p, age, F, \emptyset, \emptyset) > 18)), \\
& (((m, p, position, F, \emptyset, \emptyset) = \text{retired}), ((m, p, age, F, \emptyset, \emptyset) > 65)))) )
\end{align*}

\begin{align*}
P_{A}^{\text{allow}} &= ( (((m, p, id, T, \emptyset, \emptyset), (m, p, id, T, \emptyset, \emptyset) > 5), ((m, p, position, T, \emptyset, \emptyset) \neq \text{FirstMinister}), \emptyset), \\
& (((m, p, name, T, \emptyset, \emptyset), \emptyset, ((m, p, position, T, \emptyset, \emptyset) \neq \text{FirstMinister}), \emptyset), \\
& (((m, p, age, T, \emptyset, \emptyset), \emptyset, ((m, p, position, T, \emptyset, \emptyset) \neq \text{FirstMinister}), \emptyset), \\
& ((m, p, disease, T, \emptyset, \emptyset), \emptyset, ((m, p, position, T, \emptyset, \emptyset) \neq \text{FirstMinister}), \\
& ((m, p, disease, T, \emptyset, \emptyset) \neq \text{cancer}))))
\end{align*}

\begin{align*}
P_{A}^{\text{deny}} &= ( ((m, p, disease, T, \emptyset, \emptyset), \emptyset, ((m, p, disease, T, \emptyset, \emptyset) = (m, disease, name, F, \emptyset, \emptyset)), \\
& ((m, disease, dangerousness, F, \emptyset, \emptyset) = \text{High})))) )
\end{align*}

In the output element \text{colSel} from the Example 4 there is no \text{telephone} column because of the deny rule without data based conditions for such column. The two elements from the input \text{QBE}_{or} are converted into a one element in the output \text{QBE}_{and}. The \text{PA}_{allow} and \text{PA}_{deny} have the result of applying the set of policies \text{P} over the column set formed for the input elements \text{colSel} and \text{colWhere}.

database\_questions

The database question function checks the conditions that protect or allow the access to a whole column, see 3.2 conclusions section at page 62. Such conditions are stored in the field \text{BE}_{or,\text{col}} from the variables \text{PA}_{allow} and \text{PA}_{deny} by the above described function policy prefiltering.

The function is divided in two parts, the first part checks the conditions from the allow rules stored in \text{PA}_{allow}, and the second from the deny rules stored in \text{PA}_{deny}. In both parts are at first checked whether exists some conditions for a given column (lines 5 and 31), in such
case, if the column comes from a SELECT block without the * sign, the conditions are stored as a set element in a set of logical boolean disjunction expressions (lines 7 and 33) and such conditions are checked at the end executing a single query (lines 11 and 37). If such executed query has an empty result set, the query will fail when allow conditions are checked (line 17), in the case of the deny conditions check, will fail if the query has a not empty result set (line 43). If the column comes from a SELECT block with the * sign, for each column with whole column conditions protection a query is executed checking the conditions, however in this case the query does not fail (allow from lines 22 to 28 and deny from 48 to 53), the column is removed from the SELECT block and will is not accessed as explained in 3.2 at the conclusion section.

Example 5. The below example shows how the function works and used as an input parameter the output from the above Example 4.

In/Output

\[
q = (\text{colSel}, \emptyset, QBE_{\text{and}}) \text{ where }
\]
\[
\text{colSel} = ( (m, p, id, T, \emptyset, \emptyset),
(m, p, name, T, \emptyset, \emptyset),
(m, p, age, T, \emptyset, \emptyset),
(m, p, position, T, \emptyset, \emptyset),
(m, p, disease, T, \emptyset, \emptyset))
\]
\[
QBE_{\text{or}} = ( ((m, p, position, F, \emptyset, \emptyset) = \text{student}), ((m, p, age, F, \emptyset, \emptyset) > 18)),
((m, p, position, F, \emptyset, \emptyset) = \text{retired}), ((m, p, age, F, \emptyset, \emptyset) > 65)) )
\]
\[
PA_{\text{allow}} = ( ((m, p, id, T, \emptyset, \emptyset), ((m, p, id, T, \emptyset, \emptyset) > 5), ((m, p, position, T, \emptyset, \emptyset) \neq \text{FirstMinister}), \emptyset),
((m, p, name, T, \emptyset, \emptyset), \emptyset), ((m, p, position, T, \emptyset, \emptyset) \neq \text{FirstMinister}), \emptyset),
((m, p, age, T, \emptyset, \emptyset), \emptyset), ((m, p, position, T, \emptyset, \emptyset) \neq \text{FirstMinister}), \emptyset),
((m, p, disease, T, \emptyset, \emptyset), \emptyset), ((m, p, position, T, \emptyset, \emptyset) \neq \text{FirstMinister}),
((m, p, disease, T, \emptyset, \emptyset) \neq \text{cancer}) )
\]
\[
PA_{\text{deny}} = ( ((m, p, disease, T, \emptyset, \emptyset), \emptyset, \emptyset), ((m, p, disease, T, \emptyset, \emptyset) = (m, disease, name, F, \emptyset, \emptyset)),
((m, disease, dangerousness, F, \emptyset, \emptyset) = \text{High}))) )
\]

In the Example 5 the function only check the condition in bold from the input (if exists an element in the id column with value > 5 the access is allowed, otherwise denied), supposing such condition is true, the output is the same as the input, otherwise the column id is removed from the \(PA_{\text{allow}}\) variable.

query modification

The query modification function is divided in four parts. The first one add restrictions to the WHERE block of the original request in order to allow the access only to the allowed rows from the accessed tables (line 4). It is performed adding the row conditions stored by the \(\text{policyprefiltering}\) function in the \(BE_{\text{or}}\_\text{row}\) field from the \(PA_{\text{allow}}\) variable.
The second part do exactly the same as the first one, this time using the set of $BE_{or\_row}$ from the $PA_{deny}$ instead of $PA_{allow}$ with the exception that when the conditions are going to be added, at first are negated in order to allow the access to not allow the access to such denied rows (line 8).

In the third and fourth parts the cases conditions are added (lines 12 and 18). When the allow/deny cell conditions are added for a single column, the accessed column from the SELECT block is at first deleted in order to not access all elements of the column only the values provided by the CASE block (lines 14 and 22). See section 3.2 conclusions for more information.

**Example 5.** The below examples show how the query modification function works. As an input for the function the output from Example 4 is used.

**Input**

$$q = (colSel, \emptyset, QBE_{and})$$

$$colSel = ( (m, p, id, T, \emptyset, \emptyset),
(m, p, name, T, \emptyset, \emptyset),
(m, p, age, T, \emptyset, \emptyset),
(m, p, position, T, \emptyset, \emptyset),
(m, p, disease, T, \emptyset, \emptyset))$$

$$QBE_{and} = ( (((m, p, position, F, \emptyset, \emptyset) = student), ((m, p, age, F, \emptyset, \emptyset) > 18))),
(((m, p, position, F, \emptyset, \emptyset) = retired), ((m, p, age, F, \emptyset, \emptyset) > 65)))) )$$

$$PA_{allow} = ( ((m, p, id, T, \emptyset, \emptyset), ((m, p, id, T, \emptyset, \emptyset) > 5), ((m, p, position, T, \emptyset, \emptyset) \neq FirstMinister), \emptyset),
((m, p, name, T, \emptyset, \emptyset), \emptyset, ((m, p, position, T, \emptyset, \emptyset) \neq FirstMinister), \emptyset),
((m, p, age, T, \emptyset, \emptyset), \emptyset, ((m, p, position, T, \emptyset, \emptyset) \neq FirstMinister), \emptyset),
((m, p, position, T, \emptyset, \emptyset), \emptyset, ((m, p, position, T, \emptyset, \emptyset) \neq FirstMinister), \emptyset),
((m, p, disease, T, \emptyset, \emptyset), \emptyset, ((m, p, position, T, \emptyset, \emptyset) \neq FirstMinister),
((m, p, disease, T, \emptyset, \emptyset) \neq cancer) )$$

$$PA_{deny} = ( ((m, p, disease, T, \emptyset, \emptyset), \emptyset, \emptyset, ((m, p, disease, T, \emptyset, \emptyset) = (m, disease, name, F, \emptyset, \emptyset)),
((m, disease, dangerousness, F, \emptyset, \emptyset) = High)))) )$$
3.3. POLICY BASED QUERY EXPANSION

Output

\[ q = (colSel, colWhere, QBE_{and}) \] where

\[ colSel = ( (m, p, id, T, \emptyset, \emptyset), \]
\[ (m, p, name, T, \emptyset, \emptyset)), \]
\[ (m, p, age, T, \emptyset, \emptyset)) \]
\[ (m, p, position, T, \emptyset, \emptyset)) \]
\[ (m, p, disease, F, ((m, p, disease, F, \emptyset, \emptyset) = cancer), F)) \]
\[ (m, p, disease, F, (((m, p, disease, F, \emptyset, \emptyset) = (m, disease, name, F, \emptyset, \emptyset)), \]
\[ ((m, disease, name, F, \emptyset, \emptyset) = High)), F)) \]
\[ QBE_{and} = ( (((((m, p, position, F, \emptyset, \emptyset) = student), ((m, p, age, F, \emptyset, \emptyset) > 18)), \]
\[ (((m, p, position, F, \emptyset, \emptyset) = retired), ((m, p, age, F, \emptyset, \emptyset) > 65))), \]
\[ ((m, p, position, T, \emptyset, \emptyset) \neq FirstMinister) ) \]

In the Example 5 output, the disease column have been replace for a CASE statement in order to deny the access to the denied cells from the unique element of the input variable QA\text{deny}. To the variable QBE\text{and} the row conditions from the PA\text{allow} elements have been added, these condition are always the same for the different elements, therefore only appears once in the QBE\text{and} variable from the output.

**main algorithm**

The main algorithm shows how the introduced functions are use in order to implement the policy based query expansion.

**Input:**
a query \( q = (colSel, colWhere, QBE_{or}) \), a set of policy rules \( P \) and a state \( \Sigma \)

**Output:**
a SQL query result

**Main algorithm**(q, P, \( \Sigma \)):

```
//variables
PA\text{allow} = PA\text{deny} = \emptyset
q' = \emptyset // a new SQL query
//algorithm
q = policy\_prefiltering(q, P, \( \Sigma \))
database\_questions(PA\text{allow}, PA\text{deny})
q' = query\_modification(q', PA\text{allow}, PA\text{deny})
//return a value
return q'
```

Algorithm 4: main algorithm
Example 1
The query request all the columns with the star, but only the allowed columns are accessed and returned to the user.

Input
\[ q = (colSel, \emptyset, \emptyset) \] where
\[
\begin{align*}
\text{colSel} &= ( (m, p, id, T, \emptyset, \emptyset), \\
& (m, p, name, T, \emptyset, \emptyset) \\
& (m, p, age, T, \emptyset, \emptyset) \\
& (m, p, telephone, T, \emptyset, \emptyset) \\
& (m, p, position, T, \emptyset, \emptyset) \\
& (m, p, disease, T, \emptyset, \emptyset) ) \\
\Rightarrow & \text{SELECT * FROM patient}
\end{align*}
\]

\[ P = \]

\begin{tabular}{|c|l|}
\hline
No & Rule \\
\hline
\text{rule}_1 & allow access to db(medical, patient, name, \_)
\hline
\text{rule}_2 & allow access to db(medical, patient, age, \_)
\hline
\text{rule}_3 & allow access to db(medical, patient, telephone, \_)
\hline
\text{rule}_4 & allow access to db(medical, patient, disease, \_)
\hline
\text{rule}_5 & deny access to db(medical, patient, telephone, \_)
\hline
\text{rule}_6 & deny access to db(medical, patient, disease, \_)
\hline
\end{tabular}

\[ \Sigma = \emptyset \]

Output
\[ q = (colSel, \emptyset, \emptyset) \] where
\[
\begin{align*}
\text{colSel} &= ( (m, p, name, T, \emptyset, \emptyset), \\
& (m, p, age, T, \emptyset, \emptyset) ) \\
\Rightarrow & \text{SELECT name, age FROM patient}
\end{align*}
\]

Example 2
The query request all the columns with the star, and only the allowed columns depending on whether the conditions are satisfied or not are returned to the user.

Input
\[ q = (colSel, \emptyset, \emptyset) \] where
\[
\begin{align*}
\text{colSel} &= ( (m, p, id, T, \emptyset, \emptyset), \\
& (m, p, name, T, \emptyset, \emptyset) \\
& (m, p, age, T, \emptyset, \emptyset) \\
& (m, p, telephone, T, \emptyset, \emptyset) \\
& (m, p, position, T, \emptyset, \emptyset) \\
\Rightarrow & \text{SELECT * FROM patient}
\end{align*}
\]
3.3. POLICY BASED QUERY EXPANSION

\[(m, p, \text{disease}, T, \emptyset, \emptyset)\]

\[P =\]

\begin{tabular}{|c|l|}
\hline
No & Rule \\
\hline
\hline
rule\textsubscript{1} & allow access to db(medical, patient, name, \_)
\hline
rule\textsubscript{2} & allow access to db(medical, patient, age, \_)
\hline
rule\textsubscript{3} & allow access to db(medical, patient, position, \_) if \\
& \text{dbquery}(\text{medical, patient}, [\text{position \neq FirstMinister}])
\hline
rule\textsubscript{4} & allow access to db(medical, patient, disease, \_)
\hline
rule\textsubscript{5} & deny access to db(medical, patient, disease, \_) if \\
& \text{dbquery}(\text{medical, disease}, [\text{name} = X, \text{dangerousness} = \text{High}])
\hline
\end{tabular}

\[\Sigma = \emptyset\]

**Output**

\[q = (\text{colSel}, \emptyset, \emptyset)\text{ where}\]

If dbquery from rule 3 has a not empty set result and dbquery from rule 5 has an empty result set:

\[\text{colSel} = ( (m, p, \text{name}, T, \emptyset, \emptyset), \\
(m, p, \text{age}, T, \emptyset, \emptyset), \\
(m, p, \text{position}, T, \emptyset, \emptyset), \\
(m, p, \text{disease}, T, \emptyset, \emptyset)) \Rightarrow \text{SELECT name, age, position, disease FROM patient}\]

If dbqueries from rule 3 and rule 5 have a not empty set result:

\[q = (\text{colSel}, \emptyset, \emptyset)\text{ where}\]

\[\text{colSel} = ( (m, p, \text{name}, T, \emptyset, \emptyset), \\
(m, p, \text{age}, T, \emptyset, \emptyset), \\
(m, p, \text{position}, T, \emptyset, \emptyset)) \Rightarrow \text{SELECT name, age, position FROM patient}\]

If dbqueries from rule 3 and rule 5 have an empty set result:

\[\text{colSel} = ( (m, p, \text{name}, T, \emptyset, \emptyset), \\
(m, p, \text{age}, T, \emptyset, \emptyset), \\
(m, p, \text{disease}, T, \emptyset, \emptyset)) \Rightarrow \text{SELECT name, age, disease FROM patient}\]

If dbquery from rule 3 has an empty set result and dbquery from rule 5 has a not empty result set:

\[\text{colSel} = ( (m, p, \text{name}, T, \emptyset, \emptyset), \\
(m, p, \text{age}, T, \emptyset, \emptyset)) \Rightarrow \text{SELECT name, age FROM patient}\]
Example 3

The query accesses all the columns with the star and all the columns are accessed and returned to the user without the denied or not allowed rows.

Input

\[ \mathbf{q} = (\mathbf{colSel}, \emptyset, \emptyset) \text{ where} \]

\[
\begin{aligned}
\mathbf{colSel} = & ( (m, p, id, T, \emptyset, \emptyset), \\
& (m, p, name, T, \emptyset, \emptyset), \\
& (m, p, age, T, \emptyset, \emptyset), \\
& (m, p, \text{telephone}, T, \emptyset, \emptyset), \\
& (m, p, \text{position}, T, \emptyset, \emptyset), \\
& (m, p, \text{disease}, T, \emptyset, \emptyset) )
\end{aligned}
\]

\[ \Rightarrow \text{SELECT * FROM patient} \]

\[ P = \]

\begin{tabular}{|c|l|}
\hline
\text{No} & \text{Rule} \\
\hline
rule_1 & allow access to db(medical, patient, \_), [position \neq \text{FirstMinister}] \\
rule_2 & deny access to db(medical, patient, \_), [disease = Cancer] \\
\hline
\end{tabular}

\[ \Sigma = \emptyset \]

Output

\[ \mathbf{q} = (\mathbf{colSel}, \emptyset, QBE_{or}) \text{ where} \]

\[
\begin{aligned}
\mathbf{colSel} = & ( (m, p, id, T, \emptyset, \emptyset), \\
& (m, p, name, T, \emptyset, \emptyset), \\
& (m, p, age, T, \emptyset, \emptyset), \\
& (m, p, \text{telephone}, T, \emptyset, \emptyset), \\
& (m, p, \text{position}, T, \emptyset, \emptyset), \\
& (m, p, \text{disease}, T, \emptyset, \emptyset) )
\end{aligned}
\]

\[ \Rightarrow \text{SELECT id, name, telephone, position, disease FROM patient} \]

\[
\begin{aligned}
\mathbf{colWhere} = & ( (m, p, \text{position}, F, \emptyset, \emptyset), \\
& (m, p, \text{disease}, T, \emptyset, \emptyset) )
\end{aligned}
\]

\[ QBE_{and} = ( ((m, p, \text{position}, F, \emptyset, \emptyset) \neq \text{FirstMinister}), \\
((m, p, \text{disease}, F, \emptyset, \emptyset) \neq \text{cancer}) ) \\
\]

\[ \Rightarrow \text{WHERE (position \neq \text{FirstMinister}) AND NOT(disease = cancer)} \]
3.3. POLICY BASED QUERY EXPANSION

Example 4

The query request all the columns with the star with some conditions in the WHERE clause, the modified query maintain these conditions and add extra conditions to avoid the access to denied or not allowed rows.

Input

\[ q = (colSel, colWhere, QBE_{or}) \]

\[ colSel = ( (m, p, id, T, \emptyset, \emptyset), \\
(m, p, name, T, \emptyset, \emptyset), \\
(m, p, age, T, \emptyset, \emptyset), \\
(m, p, telephone, T, \emptyset, \emptyset), \\
(m, p, position, T, \emptyset, \emptyset), \\
(m, p, disease, T, \emptyset, \emptyset) ) \]

\[ colWhere = ( (m, p, position, T, \emptyset, \emptyset), \\
(m, p, age, T, \emptyset, \emptyset) ) \]

\[ QBE_{or} = ( \(((m, p, position, F, \emptyset, \emptyset) = \text{student}), ((m, p, age, F, \emptyset, \emptyset) > 18))) \\
((((m, p, position, F, \emptyset, \emptyset) = \text{retired}), ((m, p, age, F, \emptyset, \emptyset) > 65))) ) \]

\[ \Rightarrow \text{SELECT } * \text{ FROM patient } p \\
\text{WHERE (p.position = student AND p.age > 18)} \text{ OR} \\
\text{ (p.position = retired AND p.age > 65)} \]

\[ P = \]

<table>
<thead>
<tr>
<th>No</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule1</td>
<td>allow access to db(medical, patient, _, [position \neq FirstMinister])</td>
</tr>
<tr>
<td>rule2</td>
<td>deny access to db(medical, patient, _, [disease = Cancer])</td>
</tr>
</tbody>
</table>

\[ \Sigma = \emptyset \]

Output

\[ q = (colSel, colWhere, QBE_{or}) \]

\[ colSel = ( (m, p, id, T, \emptyset, \emptyset), \\
(m, p, name, T, \emptyset, \emptyset), \\
(m, p, age, T, \emptyset, \emptyset), \\
(m, p, telephone, T, \emptyset, \emptyset), \\
(m, p, position, T, \emptyset, \emptyset), \\
(m, p, disease, T, \emptyset, \emptyset) ) \]

\[ \Rightarrow \text{SELECT } \text{id, name, telephone, position, disease FROM patient} \]

\[ colWhere = ( (m, p, position, T, \emptyset, \emptyset), \\
(m, p, age, T, \emptyset, \emptyset), \\
(m, p, disease, T, \emptyset, \emptyset) ) \]
$QBE_{\text{and}} = ( [ ((m, p, \text{position}, F, \emptyset, \emptyset) = \text{student}), ((m, p, \text{age}, F, \emptyset, \emptyset) > 18)),
((m, p, \text{position}, F, \emptyset, \emptyset) = \text{retired}), ((m, p, \text{age}, F, \emptyset, \emptyset) > 65)) ],$

$[(m, p, \text{position}, F, \emptyset, \emptyset) \neq \text{First Minister}],
[(m, p, \text{disease}, F, \emptyset, \emptyset) \neq \text{cancer}) ]$

$\Rightarrow \text{WHERE } ((p.\text{position} = \text{student} \text{ AND } p.\text{age} > 18) \text{ OR }
(p.\text{position} = \text{retired} \text{ AND } p.\text{age} > 65)) \text{ AND }
(p.\text{position} \neq \text{First Minister}) \text{ AND }
\neg (p.\text{disease} = \text{cancer})$

Example 5

The query request all the columns with the star, and the modified query add extra conditions to avoid the access to denied or not allowed rows.

Input

$q = (colSel, \emptyset, \emptyset) \text{ where }$

$colSel = ( (m, p, id, T, \emptyset, \emptyset),
(m, p, \text{name}, T, \emptyset, \emptyset),
(m, p, \text{age}, T, \emptyset, \emptyset),
(m, p, \text{telephone}, T, \emptyset, \emptyset),
(m, p, \text{position}, T, \emptyset, \emptyset),
(m, p, \text{disease}, T, \emptyset, \emptyset) ) \Rightarrow \text{SELECT * FROM patient}$

$P =$

<table>
<thead>
<tr>
<th>No</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule1</td>
<td>allow access to db(medical, patient, _, [position \neq FirstMinister])</td>
</tr>
<tr>
<td>rule2</td>
<td>deny access to db(medical, patient, _, [disease = X]) if dbquery(medical, disease, [name = X, dangerousness = High])</td>
</tr>
</tbody>
</table>

$\Sigma = \emptyset$

Output

$q = (colSel, \emptyset, QBE_{\text{or}}) \text{ where }$

$colSel = ( (m, p, id, T, \emptyset, \emptyset),
(m, p, \text{name}, T, \emptyset, \emptyset),
(m, p, \text{age}, T, \emptyset, \emptyset),
(m, p, \text{telephone}, T, \emptyset, \emptyset),
(m, p, \text{position}, T, \emptyset, \emptyset),
(m, p, \text{disease}, T, \emptyset, \emptyset) )$

$\Rightarrow \text{SELECT id, name, telephone, position, disease FROM patient}$
3.3. POLICY BASED QUERY EXPANSION

\[
\text{colWhere} = (m, p, \text{position}, T, \emptyset, \emptyset), \\
(m, p, \text{disease}, F, \emptyset, \emptyset), \\
(m, d, \text{name}, F, \emptyset, \emptyset), \\
(m, d, \text{dangerousness}, F, \emptyset, \emptyset)
\]

\[
QBE_{\text{and}} = ([ (m, p, \text{position}, F, \emptyset, \emptyset) \neq \text{FirstMinister}], \\
[(m, p, \text{disease}, F, \emptyset, \emptyset) = (m, \text{disease}, name, F, \emptyset, \emptyset)], \\
[(m, \text{disease}, \text{dangerousness}, F, \emptyset, \emptyset) \neq \text{High}])
\]

\[\Rightarrow \text{SELECT} \ \text{id, name, age, telephone, position, disease} \\
\text{FROM} \ \text{patient p, disease d} \\
\text{WHERE} \ (p.\text{position} \neq \text{First Minister}) \ \text{AND} \\
(p.\text{dangerousness} = d.\text{name}) \ \text{AND} \\
\text{NOT}(d.\text{dangerousness} = \text{High})
\]

Example 6

The query access all the columns with the star, and in the modified query all the columns are returned to the user without some cell values which are not allowed or denied by the policy rules.

Input

\[q = (\text{colSel}, \emptyset, \emptyset) \text{ where} \]

\[
\text{colSel} = ( \ (m, p, \text{id}, T, \emptyset, \emptyset), \\
(m, p, \text{name}, T, \emptyset, \emptyset) \\
(m, p, \text{age}, T, \emptyset, \emptyset) \\
(m, p, \text{telephone}, T, \emptyset, \emptyset) \\
(m, p, \text{position}, T, \emptyset, \emptyset) \\
(m, p, \text{disease}, T, \emptyset, \emptyset) \ )
\]

\[\Rightarrow \text{SELECT * FROM} \ \text{patient} \]

\[P =
\begin{array}{|c|}
\hline
\text{No} & \text{Rule} \\
\hline
\text{rule}_1 & \text{allow access to} \ db(\text{medical, patient, disease, } [\text{disease} \neq \text{cancer}]) \\
\text{rule}_2 & \text{deny access to} \ db(\text{medical, patient, disease, } [\text{disease} = \text{AIDS}]) \\
\text{rule}_3 & \text{deny access to} \ db(\text{medical, patient, position, } [\text{position} = \text{FirstMinister}]) \\
\hline
\end{array}
\]

\[\Sigma = \emptyset\]

Output

\[q = (\text{colSel}, \emptyset, \emptyset) \text{ where} \]

\[
\text{colSel} = ( \ (m, p, \text{position}, T, ((m, p, \text{position}, F, \emptyset, \emptyset) = \text{FirstMinister}), F), \\
(m, p, \text{disease}, T, ((m, p, \text{disease}, F, \emptyset, \emptyset) = \text{AIDS}), F), \\
(m, p, \text{disease}, T, ((m, p, \text{disease}, F, \emptyset, \emptyset) \neq \text{cancer}), T) \ )
\]

\[\Rightarrow \text{SELECT CASE WHEN} \ \text{position} = \text{FirstMinister} \ \text{THEN} \ \text{NULL} \ \text{END}, \]


CASE WHEN disease = AIDS THEN NULL,
    WHEN disease ≠ cancer THEN disease END,
FROM patient p

Example 7
The query request all the columns with the star and the modified query only returns the allowed columns by the policy rules.

Input
\[ q = (colSel, \emptyset, \emptyset) \text{ where } \]
\[ colSel = ( (m, p, id, T, \emptyset, \emptyset), (m, p, name, T, \emptyset, \emptyset), (m, p, age, T, \emptyset, \emptyset), (m, p, telephone, T, \emptyset, \emptyset), (m, p, position, T, \emptyset, \emptyset), (m, p, disease, T, \emptyset, \emptyset) ) \Rightarrow \text{SELECT * FROM patient} \]

\[ P = \]

<table>
<thead>
<tr>
<th>No</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule_1</td>
<td>allow access to db(\text{(medical, patient, disease, [disease ≠ cancer]}))</td>
</tr>
<tr>
<td>rule_2</td>
<td>deny access to db(\text{(medical, patient, disease, [disease = AIDS]}))</td>
</tr>
<tr>
<td>rule_3</td>
<td>deny access to db(\text{(medical, patient, position, [position = FirstMinister]}))</td>
</tr>
<tr>
<td>rule_4</td>
<td>deny access to db(\text{(medical, patient, disease, _}))</td>
</tr>
<tr>
<td>rule_5</td>
<td>allow access to db(\text{(medical, patient, position, _}))</td>
</tr>
<tr>
<td>rule_6</td>
<td>allow access to db(\text{(medical, patient, name, _}))</td>
</tr>
</tbody>
</table>

\[ \Sigma = \emptyset \]

Output
\[ q = (colSel, \emptyset, \emptyset) \text{ where } \]
\[ colSel = ( (m, p, name, T, \emptyset, \emptyset), (m, p, position, T, \emptyset, \emptyset) ) \Rightarrow \text{SELECT name, position} \]

Example 8
The query access all the columns with the star, and in the modified query all the columns are returned to the user without some cell values which are not allowed or denied by the policy rules.

Input
\[ q = (colSel, \emptyset, \emptyset) \text{ where } \]
3.3. POLICY BASED QUERY EXPANSION

\[\text{colSel} = ( (m,p,id,T,\emptyset,\emptyset),
\quad (m,p,name,T,\emptyset,\emptyset),
\quad (m,p,age,T,\emptyset,\emptyset),
\quad (m,p,telephone,T,\emptyset,\emptyset),
\quad (m,p,position,T,\emptyset,\emptyset),
\quad (m,p,disease,T,\emptyset,\emptyset) \) \implies \text{SELECT} \ast \text{FROM} \text{patient} \]

\[\Sigma = \emptyset \]

\textbf{Output}

\[q = (\text{colSel}, \emptyset, QBE_{or}) \text{ where} \]

\[\text{colSel} = ( (m,p,id,T,\emptyset,\emptyset),
\quad (m,p,name,T,( (m,p,disease,F,\emptyset,\emptyset) = \text{AIDS}), F),
\quad (m,p,name,T,( (m,p,disease,F,\emptyset,\emptyset) \neq \text{Cancer}), T),
\quad (m,p,age,T,\emptyset,\emptyset),
\quad (m,p,disease,T,( (m,p,position,F,\emptyset,\emptyset) \neq \text{FirstMinister}), T) \) \implies \text{SELECT} \text{id},
\quad \text{CASE WHEN} \ p.disease = \text{AIDS} \ \text{THEN} \ \text{NULL}
\quad \text{WHEN} \ p.disease \neq \text{cancer} \ \text{THEN} \ \text{name} \ \text{END},
\quad \text{age},
\quad \text{CASE WHEN} \ p.position \neq \text{FirstMinister} \ \text{THEN} \ \text{disease} \ \text{END}
\quad \text{FROM} \ \text{patient} \ p \]

\textbf{Example 9}

The query access all the columns with the star, and in the modified query all the columns are returned to the user without some cell values which are not allowed or denied by the policy rules.

\textbf{Input}

\[q = (\text{colSel}, \emptyset, \emptyset) \text{ where} \]

\[\text{colSel} = ( (m,p,id,T,\emptyset,\emptyset),
\quad (m,p,name,T,\emptyset,\emptyset) \]

\[P = \]

\begin{array}{|c|l|}
\hline
\text{No} & \text{Rule} \\
\hline
\text{rule}_1 & \text{allow access to} \ db(\text{medical, patient, disease,} [id = Y, disease = X]) \ \text{if} \\
& \text{dbquery(medical, patient,} [id = Y, position \neq \text{FirstMinister}] \\
\hline
\text{rule}_2 & \text{allow access to} \ db(\text{medical, patient, position,} [id = Y, name = X]) \\
& \text{dbquery(medical, patient,} [id = Y, disease \neq \text{cancer}] \\
\hline
\text{rule}_3 & \text{deny access to} \ db(\text{medical, patient, position,} [id = Y, name = X]) \\
& \text{dbquery(medical, patient,} [id = Y, disease = \text{AIDS}] \\
\hline
\text{rule}_4 & \text{allow access to} \ db(\text{medical, patient, id, } _) \\
\hline
\text{rule}_5 & \text{allow access to} \ db(\text{medical, patient, age, } _) \\
\hline
\end{array}
(m, p, age, T, ∅, ∅) ⇒ SELECT * FROM patient
(m, p, telephone, T, ∅, ∅)
(m, p, position, T, ∅, ∅)
(m, p, disease, T, ∅, ∅)

Σ = ∅

Output

q = (colSel, ∅, QBE_or) where

colSel = ( (m, p, disease, F, ((m, p, disease, F, ∅, ∅) = (m, disease, name, F, ∅, ∅)),
        ((m, disease, name, F, ∅, ∅) = High), F)),
(m, p, disease, T, ((m, p, position, F, ∅, ∅) ≠ FirstMinister), T),
(m, p, disease, T, ((m, p, disease, F, ∅, ∅) ≠ cancer), T)

⇒ SELECT CASE WHEN p.disease = d.name AND
    d.dangerousness = High THEN NULL
    WHEN p.disease ≠ cancer THEN name END,
    WHEN p.position ≠ First Minister THEN disease END
FROM patient p, disease d

Example 10

The query request all the columns with the star, and the modified query add extra conditions
to avoid the access to denied or not allowed rows.

Input

q = (colSel, ∅, ∅) where

colSel = ( (m, p, id, T, ∅, ∅),
        (m, p, name, T, ∅, ∅)
        (m, p, age, T, ∅, ∅)
(m, p, telephone, T, ∅, ∅)
(m, p, position, T, ∅, ∅)
(m, p, disease, T, ∅, ∅) ) ⇒ SELECT * FROM patient

P =

No | Rule
---|---
rule\textsubscript{1} | allow access to db(medical, patient, disease, [id = Y, disease = X]) if
dbquery(medical, patient, [id = Y, position ≠ FirstMinister])
rule\textsubscript{2} | allow access to db(medical, patient, disease, [disease ≠ cancer])
rule\textsubscript{3} | deny access to db(medical, patient, disease, [disease = X]) if
dbquery(medical, disease, [name = X, dangerousness = High])
### 3.3. POLICY BASED QUERY EXPANSION

<table>
<thead>
<tr>
<th>No</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{rule}_1$</td>
<td>allow access to db(medical, patient, disease, $\neg$ cancer))</td>
</tr>
<tr>
<td>$\text{rule}_2$</td>
<td>allow access to db(medical, patient, position, $\neg$ FirstMinister)</td>
</tr>
</tbody>
</table>

$\Sigma = \emptyset$

**Output**

$q = (\text{colSel}, \text{colWhere}, QBE_{\text{and}}) \text{ where}$

$\text{colSel} = ( (m, p, id, T, \emptyset, \emptyset), (m, p, name, T, \emptyset, \emptyset), (m, p, age, T, \emptyset, \emptyset), (m, p, telephone, T, \emptyset, \emptyset), (m, p, position, T, \emptyset, \emptyset), (m, p, disease, T, \emptyset, \emptyset))$

$\text{colWhere} = (m, p, position, F, \emptyset, \emptyset)$

$QBE_{\text{and}} = (\neg (m, p, position, F, \emptyset, \emptyset) \neq \text{FirstMinister})$

$\Rightarrow \text{SELECT id, name, age, telephone, position, disease}$

$\text{FROM patient p, disease d}$

$\text{WHERE (p.position} \neq \text{First Minister)}$

**Example 11**

The query access all the columns with the star, and in the modified query all the columns are returned to the user without some cell values which are not allowed or denied by the policy rules, the modified query has as well as some extra conditions in the WHERE clause to avoid the access to not allowed rows.

**Input**

$q = (\text{colSel}, \emptyset, \emptyset) \text{ where}$

$\text{colSel} = ( (m, p, id, T, \emptyset, \emptyset), (m, p, name, T, \emptyset, \emptyset), (m, p, age, T, \emptyset, \emptyset), (m, p, telephone, T, \emptyset, \emptyset), (m, p, position, T, \emptyset, \emptyset), (m, p, disease, T, \emptyset, \emptyset))$

$\Rightarrow \text{SELECT * FROM patient}$

$P =$

<table>
<thead>
<tr>
<th>No</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{rule}_1$</td>
<td>allow access to db(medical, patient, disease, $\neg$ cancer))</td>
</tr>
<tr>
<td>$\text{rule}_2$</td>
<td>allow access to db(medical, patient, position, $\neg$ FirstMinister)</td>
</tr>
<tr>
<td>$\text{rule}_3$</td>
<td>allow access to db(medical, patient, position, $\neg$ FirstMinister)</td>
</tr>
</tbody>
</table>
\[ \Sigma = \emptyset \]

**Output**

\[ q = (colSel, colWhere, QBE_{and}) \text{ where} \]

\[ colSel = ( (m, p, id, T, \emptyset, \emptyset),
\quad (m, p, name, T, \emptyset, \emptyset),
\quad (m, p, age, T, \emptyset, \emptyset),
\quad (m, p, telephone, T, \emptyset, \emptyset),
\quad (m, p, position, T, ((m, p, position, F, \emptyset, \emptyset) \neq \text{FirstMinister}), T),
\quad (m, p, disease, T, ((m, p, disease, F, \emptyset, \emptyset) \neq \text{cancer}), T) ) \]

\[ colWhere = (m, p, position, F, \emptyset, \emptyset) \]

\[ QBE_{and} = (((m, p, position, F, \emptyset, \emptyset) \neq \text{FirstMinister})) \]

\[ \Rightarrow \text{SELECT id, name, age, telephone,}
\quad \text{CASE WHEN } p.\text{disease} \neq \text{cancer} \text{ THEN NULL}
\quad \text{WHEN } p.\text{disease} \neq \text{cancer} \text{ THEN name END,}
\quad \text{FROM patient } p \text{ WHERE (position } \neq \text{ FirstMinister)} \]

**Example 12**

The query request a column and the policy application returns a query failure because the user has no access to the column.

**Input**

\[ q = (colSel, \emptyset, \emptyset) \text{ where} \]

\[ colSel = ( (m, p, name, F, \emptyset, \emptyset)) \quad \Rightarrow \text{SELECT name FROM patient} \]

\[ P = \Sigma = \emptyset \]

**Output**

query failure

**Example 13**

The query request some columns and the policy application returns a query failure because the user has no access some of the requested columns.
3.3. POLICY BASED QUERY EXPANSION

Input

\[ q = (\text{colSel}, \emptyset, \emptyset) \] where

\[ \text{colSel} = ( (m, p, \text{name}, F, \emptyset, \emptyset), (m, p, \text{age}, F, \emptyset, \emptyset), \Rightarrow \text{SELECT name, age, telephone FROM patient} ) \]

\[ P = \]

<table>
<thead>
<tr>
<th>Rule</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow access to db(medical, patient, name, _)</td>
<td>(rule_1)</td>
</tr>
<tr>
<td>allow access to db(medical, patient, age, [age &gt; 18])</td>
<td>(rule_2)</td>
</tr>
</tbody>
</table>

\[ \Sigma = \emptyset \]

Output

query failure

Example 14

The query request some columns and the policy application returns a query failure because the user has no access some of the requested columns.

Input

\[ q = (\text{colSel}, \emptyset, \emptyset) \] where

\[ \text{colSel} = ( (m, p, \text{name}, F, \emptyset, \emptyset), (m, p, \text{age}, F, \emptyset, \emptyset), \Rightarrow \text{SELECT name, age, telephone FROM patient} ) \]

\[ P = \]

<table>
<thead>
<tr>
<th>Rule</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow access to db(medical, patient, _, _)</td>
<td>(rule_1)</td>
</tr>
<tr>
<td>deny access to db(medical, patient, telephone, _)</td>
<td>(rule_2)</td>
</tr>
</tbody>
</table>

\[ \Sigma = \emptyset \]

Output

query failure
Example 15

The query request all the columns with the star and with some conditions in the WHERE clause. The modified deletes the denied columns and add some extra conditions to the WHERE clause to avoid the access to the not allowed rows. **Input**

\[ q = (colSel, colWhere, QBE_{or}) \] where

\[ colSel = ( (m, p, id, T, \emptyset, \emptyset), \]
\[ (m, p, name, T, \emptyset, \emptyset), \]
\[ (m, p, age, T, \emptyset, \emptyset), \]
\[ (m, p, telephone, T, \emptyset, \emptyset), \]
\[ (m, p, position, T, \emptyset, \emptyset), \]
\[ (m, p, disease, T, \emptyset, \emptyset) ) \]

\[ colWhere = (m, p, position, T, \emptyset, \emptyset) \]

\[ QBE_{or} = ((m, p, position, F, \emptyset, \emptyset) = FirstMinister) \]

\[ \Rightarrow SELECT * \text{ FROM patient WHERE position = First Minister} \]

**P =**

<table>
<thead>
<tr>
<th>No</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule_1</td>
<td>allow access to db(medical, patient, ____, [position ______FirstMinister])</td>
</tr>
<tr>
<td>rule_2</td>
<td>deny access to db(medical, patient, telephone, ____)</td>
</tr>
</tbody>
</table>

\[ \Sigma = \emptyset \]

**Output**

\[ q = (colSel, colWhere, QBE_{and}) \] where

\[ colSel = ( (m, p, id, T, \emptyset, \emptyset), \]
\[ (m, p, name, T, \emptyset, \emptyset), \]
\[ (m, p, age, T, \emptyset, \emptyset), \]
\[ (m, p, position, T, \emptyset, \emptyset), \]
\[ (m, p, disease, T, \emptyset, \emptyset) ) \]

\[ colWhere = (m, p, position, F, \emptyset, \emptyset) \]

\[ QBE_{and} = ([ (m, p, position, F, \emptyset, \emptyset) \_\_\_\_\_\_FirstMinister], \]
\[ ([ (m, p, position, F, \emptyset, \emptyset) \_\_\_\_\_\_FirstMinister]) \]

\[ \Rightarrow SELECT id, name, age, position, disease \]
\[ \text{FROM patient p} \]
\[ \text{WHERE (position \_\_\_\_\_\_FirstMinister) AND (position \_\_\_\_\_\_FirstMinister)} \]
Chapter 4

Problem Solution: Implementation
This section provides details about the implementation of the described algorithm in the chapter 3, Problem Solution: Concept. At first a general architecture and the description of each component with the technology used are described. Afterwards a flowchart diagram describing the algorithm implementation is introduced and finally some execution results are presented in order to ensure the proper working of the system.

4.1 Architecture

The implementation adds an additional layer on top of an arbitrary RDBMS. The incoming request queries are first processed and modified according to access control policies established by the administrator before they are directed to the RDBMS.

The architecture of the implementation, illustrated in figure 4.1, has three main modules: Query Modification, Policy Engine and RDBMS Access.

**Query Modification.** The main task of this core module is to rewrite a given query in a way that only the allowed databases, tables, columns, rows and cells are accessed and returned. The Query Modification module queries the Policy Engine module for each different requested or/and accessed column from the SELECT and WHERE statement from the original request in order, and modifies the original query subtracting or replacing columns from the SELECT statement and adding constraints to the WHERE statement. The initial implementation support a subset of the standard SQL-2003 language [16] for the incoming and outgoing queries.

**Policy Engine.** This module is the responsible for a partial policy evaluation. Query context information such as the requester or credentials may be used as well. In the actual implementation a fake policy implementation is used however any policy engine [2] such as Protune, Cassandra, PeerTrust . . . may be used as well.

**RDBMS Access.** After modifying a query the modified query can be passed to a RDBMS. Since the solution is repository independent, the MySQL RDMS is integrated in the actual implementation. The result set returned contains only the allowed objects and can be directly returned to the user.

The implementation was developed in Java SE 6. The Java API, Java Database Connectivity (JDBC) was used to implement the RDBMS Access module. The fake Policy Engine module is as well a Java class which implements the policy rules described in the next section 4.2 Results. Actually in the current implementation the User Interface with which users can interact with the system, is a text file, which should contain in each line a SQL query request to the RDBMS. The result set for each incoming query is as well showed to the user in a text file and the administrator can check as well another text file where the result of each query modification is written.
4.1. ARCHITECTURE

Figure 4.1: Policy Based Query Expansion

Query Modification Module

The core module Query Modification, illustrated in figure 4.2, is divided in four different process: the parser, the Policy Prefiltering, the Database Questions and the Query Modification. The parser is the responsible for reading the incoming queries and storing them into the format used during the query modification process as described in Chapter 3.3 Policy Based Query Expansion. If the incoming query is accepted for the parser, the new query format is received for the policy prefiltering process and it checks whether the policies and the accessed columns are compatible, otherwise an error is returned to the requester. With the information recopiled for the policy prefiltering process the Query Modification process modify the query adding conditions to the WHERE block or replacing columns from the SELECT block for CASE expressions and send it to the RDBMS. The process between the Policy Prefiltering and the Query Modification is the Database Questions which checks in the database some conditions in order to remove the access to some columns or give to the user a query failure.
CHAPTER 4. PROBLEM SOLUTION: IMPLEMENTATION

Figure 4.2: Policy Based Query Expansion

The **Parser** process is implemented in the `tools.Parser` Java class inside the tools package. The Parser function is to transform the SQL query into a new format, for this purpose the SQL4J library [14] has been used to read the SQL queries.

The **Policy Prefiltering** process is implemented as a method inside the `algorithm.PolicyBasedQueryExpansion` Java class, it is divided in two parts, the first one checks the allow policies and the second one the deny policies. The allow policies check is represented as a flowchart in the figure [4.3] the deny check is basically the same process considering the negation. In the case of the allow check, the method checks for all the columns from the SELECT and WHERE
statements whether exists an allow rule, otherwise it deletes the column from the original query or returns a query failure when the not allowed column does not come from the SELECT *. If an allow policy exists it save the conditions/constraints in a different variable, depending if the conditions restrict the access to a whole column, a row or some cells of the column, see section 3.2 Conclusions.

Figure 4.3: Policy Prefiltering
The **Database Questions** process is as well divided in two parts, the first one asks the database whether the conditions for allowing the access to a whole column are satisfied or not, and the second part check the equivalent with the deny conditions. The above flowchart figure 4.4 illustrates how the allow conditions check is done. If the original query requests all the columns from a table with the SELECT * expression, a query is created and executed for each column conditions, if the conditions are satisfied the check continues, if not the column is deleted. When the column comes from a list of accessed columns, the conditions are bound and only one query is created and executed after binding all the conditions. If the result set is empty then the conditions are not satisfied and the query fails.

![Figure 4.4: Database questions](image)

The **Query Modification** process, illustrated in figure 4.5, as the other processes is divided in two parts, the first one modify the query with the conditions from the allowed and denied rows, and the second part modifies the query with the conditions from the allowed and denied cells. Both parts are represented in figure 4.5. At first the row conditions are checked and the conditions which allow the access only to the allowed rows are added to the WHERE clause. Afterwards the conditions which deny the access to the denied rows are added as well to the WHERE clause. After adding the row conditions, for each column is checked if exists cell restrictions, in such case the column is removed from the SELECT block and replaced for a CASE expression which allows the accessed only to the allowed and not denied cells. Of course the conditions are added in order, at first the CASE expression is exactly where corresponding the deleted column was and afterwards the deny conditions are writed before the allow conditions, see 3.2 Conclusions.
Figure 4.5: Query Modification
CHAPTER 4. PROBLEM SOLUTION: IMPLEMENTATION

User Interface

As commented before, the user interface is based on text files. The file sql.in.txt must contain the incoming queries, each line from the file has a SQL query which is individually treated. The modified queries are written in the sql.out.txt file and the result set from the RDBMS to the modified queries is written in the result.txt file. The three files must be in the root of the java project implementation.

Class Diagram

The class diagram, illustrated in figure [4.6] has four different packages: tools, database, algorithm and utils. The tools package contains the Parser class which main goal is to read a given query and transform it to the algorithm format. The Parser constructor receives the input query and the connection to the database and it stores the new query format in the "q" class attribute. The database package has the DB class and it implements the RDBMS access module from the figure [4.1] (the database access), it gives a connection to the MySQL database which is the one supported in the implementation, here should be implemented the access to future databases.

The algorithm package has the implementation of the algorithm, the PolicyBasedQueryExpansion contains the implementation of the Policy Prefiltering process [4.3], Database Questions [4.4] and Query Modification process [4.5], and the modified query (after each method execution as well as the final modified query) is stored in the "q" class attribute. The PolicyEngine class implements the faked policy engine, the methods allowQuestion and denyQuestion are used by the method policyPrefiltering from the PolicyBasedQueryExpansion class in order to ask for the conditions for allowing/denying the access to a column, which result is stored respectively in the paAllow or paDeny attributes from the PolicyBasedQueryExpansion class. The Main class implements the main algorithm from section 3.3, it implements the input/output for reading the incoming queries and write the result in files, and call the PolicyBasedQueryExpansion methods to obtain the modified query, therefore a future user should modify only this class if he needs a new source to obtain the incoming queries or a new source to store the result.

The utils package contain some classes which are needed as data structures during the algorithm, for instance the new Query format used during the algorithm which implements the method to add and remove columns from the SELECT and WHERE blocks as well as a method to add conditions to the WHERE block called addToBE. Other classes are PolicyApplication which is used to store the result of asking the Policy Engine for the columns access, the ColumnExpression class which represent the structure column expression (a usual column from the SELECT, WHERE or CASE block) described in the chapter 3.3.
4.1. Architecture

Figure 4.6: Class Diagram
4.2 Results

In order to test the correct work of the implementation of the algorithm, some policies over three different tables patient, disease and location have been created. The policies are described in the below tables. The patient table describes a patient with his personal data and two different foreign keys, id_disease and id_location. The first one depends on the primary key of the disease table and the pretended meaning is to link the patient with a disease, that is the patient owns such disease. The second foreign key works as the first one, in this case with a location, that is a link to know where the patient is located (e.g. a hospital).

The policies contains four diferents requesters; nurse, physician, specialist and administrator. Where the nurse is the requester with less privileges and the administrator the requester with more privileges.

**patient table** (id, name1, name2, surname1, surname2, age, borndate, nationality, address1, address2, telephone1, telephone2, telephone3, profession, email, id_location, id_disease)

**disease table** (id, name, type, classification, dangerousness, contagious, death, begin, end, treatment, comment)
The column name has the name of the disease, the column dangerousness says how dangerousness is a disease (e.g. low, high...), the column contagious says wheter the disease is contagious or not, the column death says if the patient who owns such disease caused a casualty or not because of the disease, the columns begin and end mean the date of the begininning and end of the disease.

**location table** (id, name, floor, room, section, state, region, city, address, telephone1, telephone2, telephone3, web)

**patient table policies**

<table>
<thead>
<tr>
<th>No</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rule</strong>&lt;sub&gt;1&lt;/sub&gt;</td>
<td>allow access to db(medical, patient, _, _) if requester = Administrator</td>
</tr>
<tr>
<td><strong>rule</strong>&lt;sub&gt;2&lt;/sub&gt;</td>
<td>allow access to db(medical, patient, name1, _) if requester = Nurse</td>
</tr>
<tr>
<td><strong>rule</strong>&lt;sub&gt;3&lt;/sub&gt;</td>
<td>allow access to db(medical, patient, name2, _) if requester = Nurse</td>
</tr>
<tr>
<td><strong>rule</strong>&lt;sub&gt;4&lt;/sub&gt;</td>
<td>allow access to db(medical, patient, surname1, _) if requester = Nurse</td>
</tr>
<tr>
<td><strong>rule</strong>&lt;sub&gt;5&lt;/sub&gt;</td>
<td>allow access to db(medical, patient, surname2, _) if requester = Nurse</td>
</tr>
<tr>
<td><strong>rule</strong>&lt;sub&gt;6&lt;/sub&gt;</td>
<td>allow access to db(medical, patient, age, _) if requester = Nurse</td>
</tr>
<tr>
<td><strong>rule</strong>&lt;sub&gt;7&lt;/sub&gt;</td>
<td>allow access to db(medical, patient, borndate, _) if requester = Nurse</td>
</tr>
<tr>
<td><strong>rule</strong>&lt;sub&gt;8&lt;/sub&gt;</td>
<td>allow access to db(medical, patient, nationality, _) if requester = Nurse</td>
</tr>
<tr>
<td>No</td>
<td>Rule</td>
</tr>
<tr>
<td>-----</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>rule_9</td>
<td>allow access to db(medical, patient, email, _) if requester = Nurse</td>
</tr>
<tr>
<td>rule_10</td>
<td>allow access to db(medical, patient, id_location, _) if requester = Nurse</td>
</tr>
<tr>
<td>rule_11</td>
<td>deny access to db(medical, patient, telephone1, _) if requester = Nurse</td>
</tr>
<tr>
<td>rule_12</td>
<td>deny access to db(medical, patient, telephone2, _) if requester = Nurse</td>
</tr>
<tr>
<td>rule_13</td>
<td>deny access to db(medical, patient, telephone3, _) if requester = Nurse</td>
</tr>
<tr>
<td>rule_14</td>
<td>deny access to db(medical, patient, address1, _) if requester = Nurse</td>
</tr>
<tr>
<td>rule_15</td>
<td>deny access to db(medical, patient, address2, _) if requester = Nurse</td>
</tr>
<tr>
<td>rule_16</td>
<td>deny access to db(medical, patient, id_disease, _) if requester = Nurse</td>
</tr>
<tr>
<td>rule_17</td>
<td>deny access to db(medical, patient, _, [profession = First Minister]) if requester = Nurse</td>
</tr>
<tr>
<td>rule_18</td>
<td>deny access to db(medical, patient, _, [profession = famous]) if requester = Nurse</td>
</tr>
<tr>
<td>rule_19</td>
<td>allow access to db(medical, patient, _, [profession ≠ First Minister]) if requester = Physician</td>
</tr>
<tr>
<td>rule_20</td>
<td>deny access to db(medical, patient, id_disease, [id_disease = X]) if requester = Physician AND dbquery(medical, disease, [id = X, classification != A])</td>
</tr>
<tr>
<td>rule_21</td>
<td>deny access to db(medical, patient, id_disease, [id_disease = X]) if requester = Physician AND dbquery(medical, disease, [id = X, classification = A])</td>
</tr>
<tr>
<td>rule_22</td>
<td>deny access to db(medical, patient, id_disease, [id_disease = X]) if requester = Physician AND dbquery(medical, disease, [id = X, contagious = TRUE])</td>
</tr>
<tr>
<td>rule_23</td>
<td>allow access to db(medical, patient, _, _) if requester = Specialist</td>
</tr>
<tr>
<td>rule_24</td>
<td>deny access to db(medical, patient, id_disease, [id_disease = X]) if requester = Specialist AND dbquery(medical, disease, [id_disease = X, dangerousness = High, contagious = TRUE, classification = A])</td>
</tr>
</tbody>
</table>

Table 4.1: Applicable policies to the patient table
The administrator has whole access to the patient table, the specialist is allowed to access the whole patient table except come values of the id_disease column (when the disease is considered with a high level of dangerousness is contagious and classified with A). The Physician is as well allowed to access the whole table but not to the some values of the column id_disease (when the disease is considered as contagious or classified with A). The nurse requester is allowed to access some columns (name1, name2, surname1, surname2, age, borndate, nationality, email, id_location) and has the access denied to the columns telephone1, telephone2, telephone3, address1, address2, id_disease as well as the access denied to the rows where the profession is first minister or famous.

disease table policies

<table>
<thead>
<tr>
<th>No</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule25</td>
<td>allow access to db(medical, disease, _, _) if requester = Administrator</td>
</tr>
<tr>
<td>rule26</td>
<td>allow access to db(medical, disease, _, _) if requester = Nurse</td>
</tr>
<tr>
<td>rule27</td>
<td>deny access to db(medical, disease, name, _) if requester = Nurse</td>
</tr>
<tr>
<td>rule28</td>
<td>deny access to db(medical, disease, name, _) if requester = Nurse</td>
</tr>
<tr>
<td>rule29</td>
<td>deny access to db(medical, disease, contagious, _) if requester = Nurse</td>
</tr>
<tr>
<td>rule30</td>
<td>deny access to db(medical, disease, dangerousness, _) if requester = Nurse</td>
</tr>
<tr>
<td>rule31</td>
<td>deny access to db(medical, disease, death, _) if requester = Nurse</td>
</tr>
<tr>
<td>rule32</td>
<td>allow access to db(medical, disease, _, _) if requester = Physician</td>
</tr>
<tr>
<td>rule33</td>
<td>deny access to db(medical, disease, death, _) if requester = Physician</td>
</tr>
<tr>
<td>rule34</td>
<td>allow access to db(medical, disease, _, _) if requester = Specialist</td>
</tr>
<tr>
<td>rule35</td>
<td>deny access to db(medical, disease, death, _) if requester = Specialist AND dbquery(medical, disease, [death = TRUE])</td>
</tr>
</tbody>
</table>

Table 4.2: Applicable policies to the disease table

The administrator has again the full access to the table, the specialist has the full access except to the column death in case it contains the value TRUE. The physician has the access to the whole table except to the column death. The nurse requester has the access to the columns contagious, dangerousness and death forbidden, the column name is forbidden if the disease is consider as contagious or has caused a casualty.
4.2. RESULTS

location table policies

<table>
<thead>
<tr>
<th>No</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule36</td>
<td>allow access to db(medical, location, _, _) if requester = Administrator</td>
</tr>
<tr>
<td>rule37</td>
<td>allow access to db(medical, location, _, _) if requester = Nurse</td>
</tr>
<tr>
<td>rule38</td>
<td>allow access to db(medical, location, _, _) if requester = physician</td>
</tr>
<tr>
<td>rule39</td>
<td>allow access to db(medical, location, _, _) if requester = Specialist</td>
</tr>
</tbody>
</table>

Table 4.3: Applicable policies to the location table

The location table columns are accessible to everyone.

queries and applicable policies

The next eleven queries are executed by the four different requesters before described.

1. SELECT * FROM patient
2. SELECT * FROM disease
3. SELECT * FROM location
4. SELECT patient.name1, patient.address1, patient.telephone1, patient.profession FROM patient
5. SELECT patient.name1, patient.address1, patient.telephone1, patient.profession FROM patient WHERE patient.profession = 'second_minister'
6. SELECT * FROM patient WHERE patient.profession = 'second_minister'
7. SELECT patient.name1, disease.name, disease.dangerousness FROM patient, disease WHERE patient.id = disease.id
8. SELECT patient.name1, disease.name, disease.death FROM patient, disease WHERE patient.id = disease.id
9. SELECT patient.name1, disease.name, disease.dangerousness, disease.contagious, disease.classification, disease.death FROM patient, disease WHERE patient.id = disease.id
10. SELECT * FROM patient, disease WHERE patient.id = disease.id
11. SELECT * FROM patient, location WHERE patient.id = location.id

For each original query and requester usually the query modification process gives a different modified query. Here, are described for each query and requester the original query, the list of applicable policies, the modified query given an implementation of the concept chapter, as well as a short comment explaining why is the query modified.
1. SELECT * FROM patient

- Set of applicable policies
  - Administrator: rule1
  - Nurse: rule2...rule18
  - Physician: rule19...rule22
  - Specialist: rule23 and rule24

- Query modification
  - Administrator:
    ```sql
    SELECT patient.id, patient.name1, patient.name2, patient.surname1,
    patient.surname2, patient.age, patient.borndate,
    patient.nationality, patient.address1, patient.address2,
    patient.telephone1, patient.telephone2, patient.telephone3,
    patient.profession, patient.email, patient.id_location,
    patient.id_disease
    FROM medical.patient
    ```
  - Nurse:
    ```sql
    SELECT patient.name2, patient.surname1, patient.surname2, patient.age,
    patient.borndate, patient.nationality, patient.email,
    patient.id_location
    FROM medical.patient
    WHERE NOT((patient.profession = 'first_minister') OR
    (patient.profession = 'famous'))
    ```
  - Physician:
    ```sql
    SELECT patient.id, patient.name1, patient.name2, patient.surname1,
    patient.surname2, patient.age, patient.borndate,
    patient.nationality, patient.address1, patient.address2,
    patient.telephone1, patient.telephone2, patient.telephone3,
    patient.profession, patient.email, patient.id_location,
    CASE WHEN disease.contagious = 'true' THEN NULL
    WHEN disease.classification = 'A' THEN NULL
    WHEN disease.dangerousness <> 'high' THEN
    patient.id_disease END
    FROM medical.disease, medical.patient
    WHERE ((patient.profession <> 'first_minister'))
    ((patient.id_disease = disease.id))
    ```
  - Specialist:
    ```sql
    SELECT patient.id, patient.name1, patient.name2, patient.surname1,
    patient.surname2, patient.age, patient.borndate,
    patient.nationality, patient.address1, patient.address2,
    patient.telephone1, patient.telephone2, patient.telephone3,
    patient.profession, patient.email, patient.id_location,
    CASE WHEN disease.contagious = 'true' AND
    disease.dangerousness = 'high' AND
    disease.classification = 'A' THEN NULL
    ```
4.2. RESULTS

WHEN patient.id_disease THEN patient.id_disease END
FROM medical.disease, medical.patient
WHERE ((patient.id_disease = disease.id))

• Comment
The result query for the Nurse requester access only to the allowed columns from rules 2 to 10 and the allowed rows from the rules 17 and 18. The result query from the Physician requester access all the columns from the patient, with some constraints to the column id_disease (rules 20, 21 and 22), the rows where the patient profession equals to first_minister are as well denied. The Specialist requester is allowed to access all the columns from the patient table with some restrictions to the id_disease column (rule 24). Both the Physician and the Specialist requesters have in the where clause the join between the patient and disease table because both access to the disease table in order to restrict the access to the id_disease column.

2. SELECT * FROM disease

• Set of applicable policies
  – Administrator: rule25
  – Nurse: rule26...rule31
  – Physician: rule32 and rule33
  – Specialist: rule34 and rule35

• Query modification
  – Administrator:
    SELECT disease.id, disease.name, disease.type,
    disease.classification, disease.dangerousness,
    disease.contagious, disease.death, disease.begin,
    disease.end, disease.treatment, disease.comment
    FROM medical.disease

  – Nurse:
    SELECT disease.id, disease.type, disease.classification,
    disease.begin, disease.end, disease.treatment,
    disease.comment
    FROM medical.disease

  – Physician and Specialist:
    SELECT disease.id, disease.name, disease.type,
    disease.classification, disease.dangerousness,
    disease.contagious, disease.begin, disease.end,
    disease.treatment, disease.comment
    FROM medical.disease

• Comment
The result query from the Nurse requester has no access to the columns denied in rules 27 to 31. The result query from the Physician request has no access to the denied column death (rule 33). The result from the Specialist requester has as well the column death denied (rule 35) because there is at least one value in the death column with TRUE value.
3. SELECT * FROM location
   
   • Set of applicable policies
     – Administrator: rule\textsubscript{36}
     – Nurse: rule\textsubscript{37}
     – Physician: rule\textsubscript{38}
     – Specialist: rule\textsubscript{39}
   
   • Query modification
     – Administrator = Nurse = Physician = Specialist :
       SELECT location.id, location.name, location.floor, location.room,
       location.section, location.state, location.region,
       location.city, location.address, location.telephone1,
       location.telephone2, location.telephone3, location.web
       FROM medical.location
     
     – Comment
       The location table has no restrictions.

4. SELECT patient.name1, patient.address1, patient.telephone1, patient.profession
   FROM patient
   
   • Set of applicable policies
     – Administrator: rule\textsubscript{1}
     – Nurse: rule\textsubscript{2}, rule\textsubscript{11}, rule\textsubscript{14}, rule\textsubscript{17}
     – Physician: rule\textsubscript{19}
     – Specialist: rule\textsubscript{23}
   
   • Modified query
     – Administrator, Specialist
       SELECT patient.name1, patient.address1, patient.telephone1,
       patient.profession
       FROM medical.patient
     
     – Nurse: query failure.
     – Physician:
       SELECT patient.name1, patient.address1,
       patient.telephone1, patient.profession
       FROM medical.patient WHERE ((patient.profession <> 'first_minister'))
     
     • Comment
       The Specialist has no restrictions to the selected columns from the patient table. The Nurse requester obtains a query failure because has no access to the column profession (there is no rule which allow the access). The result query for the Physician requester has access to all the selected columns excepts to the rows where the profession value is first_minister (rule 19).

5. SELECT patient.name1, patient.address1, patient.telephone1, patient.profession
   FROM patient
   WHERE patient.profession = 'second_minister'
• Set of applicable policies
  - Administrator: rule1
  - Nurse: rule2, rule11, rule14, rule17
  - Physician: rule19
  - Specialist: rule23

• Modified Query
  - Administrator, Specialist
    SELECT patient.name1, patient.address1, patient.telephone1, patient.profession
    FROM medical.patient
    WHERE ((patient.profession = 'second_minister'))
  - Nurse: query failure.
  - Physician:
    SELECT patient.name1, patient.address1, patient.telephone1, patient.profession
    FROM medical.patient
    WHERE NOT((patient.profession = 'first_minister')) AND
    ((patient.profession <> 'first_minister')) AND
    ((patient.profession = 'second_minister'))

• Comment
  The Specialist requester has the full access because there is no restrictions to the selected queries. The Nurse requester has a query failure because the profession column is not allowed (there is no rule which allow the access). The result query for the Physician requester has access to all the selected columns excepts to the rows where the profession value is first_minister (rule19).

6. SELECT * FROM patient WHERE patient.profession = 'second_minister'

• Set of applicable policies
  - Administrator: rule1
  - Nurse: rule2...rule18
  - Physician: rule19...rule22
  - Specialist: rule23 and rule24

• Modified query
  - Administrator:
    SELECT patient.id, patient.name1, patient.name2, patient.surname1, patient.surname2, patient.age, patient.borndate, patient.nationality, patient.address1, patient.address2, patient.telephone1, patient.telephone2, patient.telephone3, patient.profession, patient.email, patient.id_location, patient.id_disease
    FROM medical.patient
    WHERE ((patient.profession = 'second_minister'))
Nurse: query failure

Physician

\[
\begin{align*}
&\text{SELECT patient.id, patient.name1, patient.name2, patient.surname1,} \\
&\quad \text{patient.surname2, patient.age, patient.borndate, patient.nationality,} \\
&\quad \text{patient.address1, patient.address2, patient.telephone1,} \\
&\quad \text{patient.telephone2, patient.telephone3, patient.profession,} \\
&\quad \text{patient.email, patient.id_location,} \\
&\quad \text{CASE WHEN disease.contagious = 'true' THEN NULL} \\
&\quad \text{WHEN disease.classification = 'A' THEN NULL} \\
&\quad \text{WHEN disease.dangerousness <> 'high' THEN patient.id_disease END} \\
\end{align*}
\]

FROM medical.disease, medical.patient
WHERE ((patient.profession <> 'first_minister')) AND
((patient.profession = 'second_minister')) AND
((patient.id_disease = disease.id))

Specialist:

\[
\begin{align*}
&\text{SELECT patient.id, patient.name1, patient.name2, patient.surname1,} \\
&\quad \text{patient.surname2, patient.age, patient.borndate,} \\
&\quad \text{patient.nationality, patient.address1, patient.address2,} \\
&\quad \text{patient.telephone1, patient.telephone2, patient.telephone3,} \\
&\quad \text{patient.profession, patient.email, patient.id_location,} \\
&\quad \text{CASE WHEN disease.contagious = 'true' AND} \\
&\quad \text{disease.dangerousness = 'high' AND} \\
&\quad \text{disease.classification = 'A' THEN NULL} \\
&\quad \text{WHEN patient.id_disease THEN patient.id_disease END} \\
\end{align*}
\]

FROM medical.disease, medical.patient
WHERE ((patient.profession = 'second_minister')) AND
((patient.id_disease = disease.id))

- Comment

The Nurse requester result is a query failure due to the access of the column \textit{profession} in the where clause (there is no rule which allow the access). The Physician requester has the access to the whole table without the rows where the profession values are \textit{first_minister} and to the column \textit{id_disease} are as well some restrictions (rules 20 to 22). The Specialist request is modified and some conditions added to the access of the column \textit{id_disease} (rule 24). Both the Physician and the Specialist requesters have in the where clause the join between the \textit{patient} and \textit{disease} table because both access to the disease table in order to restrict the access to the \textit{id_disease} column.

7. SELECT patient.name1, disease.name, disease.dangerousness
FROM patient, disease
WHERE patient.id_disease = disease.id

- Set of applicable policies
  - Administrator: \textit{rule}_{1}, \textit{rule}_{25}
  - Nurse: \textit{rule}_{2}, \textit{rule}_{16}, \textit{rule}_{17}, \textit{rule}_{18}, \textit{rule}_{26}, \textit{rule}_{27}, \textit{rule}_{30}
  - Physician: \textit{rule}_{19} \ldots \textit{rule}_{22}, \textit{rule}_{32}
  - Specialist: \textit{rule}_{23}, \textit{rule}_{24}, \textit{rule}_{34}
4.2. RESULTS

- Modified query
  - Administrator:
    
    ```
    SELECT patient.name1, disease.name, disease.dangerousness
    FROM medical.disease, medical.patient
    WHERE ((patient.id_disease = 'disease.id'))
    ```
  
  - Nurse: query failure.
  
  - Physician:
    
    ```
    SELECT patient.name1, disease.name, disease.dangerousness
    FROM medical.disease, medical.patient
    WHERE ((patient.profession <> 'first_minister'
    AND disease.dangerousness <> 'high')) AND
    ((patient.id_disease = 'disease.id')) AND
    ((patient.profession <> 'first_minister')) AND
    NOT((disease.contagious = 'true') OR (disease.classification = 'A'))
    ```

  - Specialist:
    
    ```
    SELECT patient.name1, disease.name, disease.dangerousness
    FROM medical.disease, medical.patient
    WHERE ((patient.id_disease = 'disease.id')) AND
    NOT((disease.contagious = 'true' AND
    disease.dangerousness = 'high' AND
    disease.classification = 'A')) AND
    ```

- Comment
  The Nurse requester has the query failure because has no access to the dangerousness column (rule 30). The Physician has access to the selected columns except to some restricted rows extracted from rules 19 to 22. The Specialist has as well all the selected columns allowed except to some restricted rows extracted from rule 24.

8. SELECT patient.name1, disease.name, disease.death
   FROM patient, disease
   WHERE patient.id_disease = disease.id

- Set of applicable policies
  - Administrator: `rule1, rule25`
  - Nurse: `rule2, rule16, rule17, rule18, rule26, rule27, rule28`
  - Physician: `rule19 ... rule22, rule32, rule33`
  - Specialist: `rule23, rule24, rule34, rule35`

- Result
  - Administrator:
    
    ```
    SELECT patient.name1, disease.name, disease.death
    FROM medical.disease, medical.patient
    WHERE ((patient.id_disease = 'disease.id'))
    ```
  
  - Nurse: QUERY FAILURE!!
  
  - Physician: QUERY FAILURE!!
  
  - Specialist: QUERY FAILURE!!
• comments
The Nurse and Physician have the access to the death column denied therefore they have a query failure (rules 27 and 31). The Specialist as well has the access denied (rule 35) to the column death because it is exist a TRUE value in such column.

9. SELECT patient.name1, disease.name, disease.dangerousness, disease.contagious, disease.classification, disease.death
FROM patient, disease
WHERE patient.id_disease = disease.id

• Set of applicable policies
  – Administrator: rule₁, rule₂₅
  – Nurse: rule₂, rule₁₈, rule₂₆, rule₃₁
  – Physician: rule₁₉, rule₂₂, rule₃₂ and rule₃₃
  – Specialist: rule₂₃, rule₂₄, rule₃₄, rule₃₅

• Modified query
  – Administrator:
    SELECT patient.name1, disease.name, disease.dangerousness, disease.contagious, disease.classification, disease.death FROM medical.disease, medical.patient
    WHERE ((patient.id_disease = 'disease.id'))
  – Nurse: query failure
  – Physician: query failure
  – Specialist: query failure

10. SELECT * FROM patient, disease WHERE patient.id_disease = disease.id

• Set of applicable policies
  – Administrator: rule₁, rule₂₇
  – Nurse: rule₂, rule₁₆, rule₁₇, rule₁₈, rule₂₆, rule₃₁,
  – Physician: rule₁₉, rule₂₂, rule₃₃, rule₃₄
  – Specialist: rule₂₃, rule₂₄, rule₃₄, rule₃₅

• Modified query
  – Administrator:
    SELECT patient.id, patient.name1, patient.name2, patient.surname1, patient.surname2, patient.age, patient.borndate, patient.nationality, patient.address1, patient.address2, patient.telephone1, patient.telephone2, patient.telephone3, patient.proffesion, patient.email, patient.id_location, patient.id_disease, disease.id, disease.name, disease.type, disease.classification, disease.dangerousness, disease.contagious, disease.death, disease.begin, disease.end, disease.treatment, disease.comment
    FROM medical.disease, medical.patient
    WHERE ((patient.id_disease = 'disease.id'))
  – Nurse: query failure
  – Physician:
SELECT patient.id, patient.name1, patient.name2, patient.surname1, 
patient.surname2, patient.age, patient.borndate, patient.nationality, 
patient.address1, patient.address2, patient.telephone1, 
patient.telephone2, patient.telephone3, patient.profession, 
patient.email, patient.id_location, 
CASE WHEN disease.contagious = 'true' THEN NULL 
WHEN disease.classification = 'A' THEN NULL 
WHEN disease.dangerousness <> 'high' THEN patient.id_disease END, 
disease.id, disease.name, disease.type, disease.classification, 
disease.dangerousness, disease.contagious, disease.begin, 
disease.end, disease.treatment, disease.comment 
FROM medical.disease, medical.patient 
WHERE ((patient.profession <> 'first_minister' AND 
    disease.dangerousness <> 'high')) AND 
    ((patient.id_disease = 'disease.id')) AND 
    ((patient.profession <> 'first_minister')) AND 
NOT((disease.contagious = 'true') OR (disease.classification = 'A'))

Specialist:
SELECT patient.id, patient.name1, patient.name2, patient.surname1, 
patient.surname2, patient.age, patient.borndate, patient.nationality, 
patient.address1, patient.address2, patient.telephone1, 
patient.telephone2, patient.telephone3, patient.profession, 
patient.email, patient.id_location, 
CASE WHEN disease.contagious = 'true' AND 
    disease.dangerousness = 'high' AND 
    disease.classification = 'A' THEN NULL 
WHEN patient.id_disease THEN patient.id_disease END, 
location.id, location.name, location.floor, location.room, 
location.section, location.state, location.region, location.city, 
location.address, location.telephone1, location.telephone2, 
location.telephone3, location.web 
FROM medical.location, medical.disease, medical.patient 
WHERE ((patient.id_location = 'location.id'))

• Comment
The Nurse requester has no access to the \textit{id\_disease} column therefore the query 
failure is the result of the modification. The Physician has no access to the \textit{death} 
column from the disease column, and the rows from the conditions from rule 20 to 
23 are as well denied. The Specialist as well has the death column denied because 
exists a true value in such column and some rows from rule 24 are as well denied.

11. SELECT * FROM patient, location WHERE patient.id_location = location.id

• Set of applicable policies
  - Administrator: \textit{rule\_1, rule\_36}
  - Nurse: \textit{rule\_2 ... rule\_18, rule\_37}
  - Physician: \textit{rule\_19 ... rule\_22, rule\_38}
  - Specialist: \textit{rule\_23, rule\_24 and rule\_39}
• Result

  - Administrator:
    ```sql
    SELECT patient.id, patient.name1, patient.name2, patient.surname1,
           patient.surname2, patient.age, patient.borndate, patient.nationality,
           patient.address1, patient.address2, patient.telephone1,
           patient.telephone2, patient.telephone3, patient.profession,
           patient.email, patient.id_location, patient.id_disease, location.id,
           location.name, location.floor, location.room, location.section,
           location.state, location.region, location.city, location.address,
           location.telephone1, location.telephone2, location.telephone3,
           location.web
    FROM medical.location, medical.patient
    WHERE ((patient.id_location = 'location.id'))
    ```

  - Nurse:
    ```sql
    SELECT patient.name2, patient.surname1, patient.surname2, patient.age,
           patient.borndate, patient.nationality, patient.email,
           patient.id_location, location.id, location.name, location.floor,
           location.room, location.section, location.state, location.region,
           location.city, location.address, location.telephone1,
           location.telephone2, location.telephone3, location.web
    FROM medical.location, medical.patient
    WHERE NOT((patient.profession = 'first_minister')
              OR (patient.profession = 'famous')) AND
    ((patient.id_location = 'location.id'))
    ```

  - Physician:
    ```sql
    SELECT patient.id, patient.name1, patient.name2, patient.surname1,
           patient.surname2, patient.age, patient.borndate, patient.nationality,
           patient.address1, patient.address2, patient.telephone1,
           patient.telephone2, patient.telephone3, patient.profession,
           patient.email, patient.id_location,
           CASE WHEN disease.contagious = 'true' THEN NULL
           WHEN disease.classification = 'A' THEN NULL
           WHEN disease.dangerousness <> 'high' THEN patient.id_disease END,
           location.id, location.name, location.floor, location.room,
           location.section, location.state, location.region, location.city,
           location.address, location.telephone1, location.telephone2,
           location.telephone3, location.web
    FROM medical.location, medical.disease, medical.patient
    WHERE ((patient.profession <> 'first_minister')) AND
    ((patient.id_location = 'location.id'))
    ```

  - Specialist:
    ```sql
    SELECT patient.id, patient.name1, patient.name2, patient.surname1,
           patient.surname2, patient.age, patient.borndate, patient.nationality,
           patient.address1, patient.address2, patient.telephone1,
           patient.telephone2, patient.telephone3, patient.profession,
           patient.email, patient.id_location,
    ```
CASE WHEN disease.contagious = 'true' AND
disease.dangerousness = 'high' AND
disease.classification = 'A' THEN NULL
WHEN patient.id_disease THEN patient.id_disease END,
location.id, location.name, location.floor, location.room,
location.section, location.state, location.region, location.city,
location.address, location.telephone1, location.telephone2,
location.telephone3, location.web
FROM medical.location, medical.disease, medical.patient
WHERE ((patient.id_location = 'location.id'))

4.3 Changing policies and queries

Of course if the queries change, the modified queries will as well change. Therefore the result set will be a different one. The user must modify the sql.in.txt file in order to prove with different queries and check the sql.out.txt file to check how the new queries look like. The new result set for the new queries will be available in the result.txt file.

To change the policy rules is more difficult since the policy engine is faked, implemented as a Java class. Therefore, in order to change the rules the algorithm.PolicyEngine Java class should be modified. If the policies rules are changed, the query modification will work whenever the rules make sense with the database tables and columns, otherwise the RDBMS will return an error because probably a new introduced column or table will not exist in the database.

Therefore if in a future the current implementation is joined with a real policy engine, adding and/or removing policies to the policy engine as well as writing new queries as an input, it is supposed that the query modification module will work properly.
Chapter 5

Problem Solution: Experiments and Evaluation
This chapter evaluates the implementation of this work. At first the environment where the tests have been executed is described, afterward the expected results are introduced and finally the experiments results with some comments and a conclusion are presented.

5.1 Introduction

I set up a MySQL database with 10,000 records in each table in a Pentium M 1.4 GHz with 1GB memory and with a Linux operating system installed, and I sent the queries from the same computer. I checked the approach by setting an initial set of queries without a WHERE clause in order to return the maximal number of results, exactly 10,000 records. I automatically generated extra boolean expression extracted from the policies described in chapter 4, which were added to the original queries. Since I wanted to test the impact of adding conditions based on the same table which the original query access and conditions based on tables which the original query does not access, I evaluate the two possible options splitting the experiments in two different sets, one without adding extra tables to the FROM clause and the other adding extra tables to the FROM clause. The type of conditions added to the WHERE clause are as well divided in groups, the different groups are the following: conditions connected with AND, with OR and with AND NOT. In order to represent a not favorable situation, all the added conditions are based on columns without an index in the tables, in case of having an index in some columns the times will be even faster than the times shown here. For the same reason, the queries here evaluated do not obtain a query failure during the query modification algorithm, such situation will give as well a better performance.

The reason of having three different groups of conditions in the WHERE clause to test are due to how the query modification process modifies the query. Actually it connects the conditions with an AND or AND NOT, however each connected condition may have boolean expression inside connected with an OR. Therefore the worst situation is adding a single condition which inside has many OR connectors like in a) \( \text{WHERE} (A \lor \cdots \lor I) \), although this situation is not very probable because it means that all the policies rules affect only a single column, this other situation b) \( \text{WHERE} (A \lor \cdots \lor I) \land \cdots \land (J \lor \cdots \lor N) \) is more probably to happen, where the fact of having the AND connector will give a better performance as is commented in the experiment section. Other probably situations are c) \( \text{WHERE} \neg(A \lor \cdots \lor I) \land \cdots \land \neg(J \lor \cdots \lor N) \) and the situations b) and c) together.

5.2 Experiments

I expect that the fact of adding conditions to the WHERE clause will generally reduce the response time of a query execution in the database with one exception, when the conditions are only connected with an OR, in such case I expect the time will be higher than the original query. The reason is because when adding AND conditions the result set will be reduced, however when OR conditions are added the result set is not so strongly reduced as with the AND conditions. When other tables are added to the FROM clause I expect the time will increase considerably, in such situations if extra conditions are added to the WHERE clause, I expect it will work as before explained, the more conditions added to the WHERE clause the faster will be the query execution, although the new reduced execution time for the most of the cases will be slower than the original query execution time.
5.2. EXPERIMENTS

Conditions based on the same table where the query access

The graph 5.1 shows how adding conditions to the WHERE clause (extracted from the policies) usually decrease the evaluation time. The reason for this decrease is due to the extra conditions added to the WHERE clause, because it reduces the number of returned records to the user. In the graph 5.1 we can observe adding only OR conditions does not decrease the response time while AND or AND NOT (AND NOT(conditions)) decrease the response time. The time is severely reduced when a strong condition is added, for instance, in the graph after adding two conditions where one of them is a strong one, we can observe an steep slope. If we continue adding more conditions they do not reduce very much the number of returned records, therefore the slope keep it constant in the graph. With eight extra conditions the slope start again to go down due to the MySQL query optimizer, which start to find some contradictions in the conditions and some queries have a reponse time aroung zero miliseconds. With ten different conditions most of the queries have a contradiction in the WHERE clause and the average time tends to zero. Adding OR conditions does not improve the performance, however the good new is that it does not increase the reponse time. As said in the introduction, the case a) is not very probable although is here measured, the case b) will look likes a function between the OR and AND, that is the slope will be not so steep. And the situations b) and c) will have even a better performance due again to the query optimizer which using the Morgan’s Laws it will convert the following expression AND NOT(A ∨ ⋯ ∨ N) to the new expression ¬(A) ∧ ⋯ ∧ ¬(N), that is, the OR connector is change to an AND NOT connector which is faster than an OR.

![Figure 5.1: Response time when increasing the number of WHERE clauses for one table.](image)
Conditions based on tables where the query does not access

The graphs 5.2, 5.3 and 5.4 shows how adding conditions to the WHERE clause (extracted from the policies) usually decrease the evaluation time. All join conditions are made with the primary key of each table, that is, the join columns have an index in the table. Some experiments made with columns without an index have a disproportionately evaluation time, where the times are over a minute for a join between two tables. Therefore these situations must be avoided otherwise the cost of the fine access protection will be very high to be used. The red line of the graphs represents the original query execution time where no conditions are added to the WHERE clause.

In the graphs the same conclusion as for the graph 5.1 are here extracted. We can observe that adding one extra table to the FROM clause does not increase too much the response time with respect to the original query, in fact the response times are faster than the original query for the graphs 5.2 and 5.4 when some extra conditions are added to the WHERE clause. Adding two tables to the FROM clause give a response time better or close than the original query when some conditions are added to the WHERE clause. This is due to the query optimizer which at first performs the conditions from the WHERE clause and afterwards the join, in such way the tables will not have too many records when the join is performed.

![Graph showing response time vs. number of extra WHERE conditions](image)

**Figure 5.2:** Response time when increasing the number of FROM and WHERE clause, where the WHERE clause are connected with AND.
However when 4 tables are added to the FROM clause the times are close to, or a bit higher than the original query execution time, even adding eight extra tables to the FROM clause the response time is not very higher than the original query when some conditions are added to the WHERE clause (the time is around 50 percent higher). Of course if no extra where conditions are added to the WHERE clause, the response time is slower, around 50 percent slower for each extra added table.

Figure 5.3: Response time when increasing the number of FROM and WHERE clause, where the WHERE clause are connected with OR.

The graph 5.3 with OR conditions added to the WHERE clause, has of course, a worst performance with respect to the original query. As said in the introduction, the case a) is not very probable although is here measured, the case b) will look likes a function between the OR and AND, that is the slope will be not so steep. And the situations b) and c) will have even a better performance due again to the query optimizer which using the Morgan’s Laws it will convert the following expression AND NOT(A V ··· V N) to the new expression ¬(A) ∧ ··· ∧ ¬(N), that is, the OR connector is change to an AND NOT connector which is faster than an OR.
5.3 Adding indexes to the column tables

After some experiments with an index in one of the columns which appears in the WHERE clause from the modified queries, the result times are really fast. The experiments were made with only one indexed extra column, when the indexed column is in the WHERE clause. The response time in such case could be only a few milliseconds. When some extra tables are added as well to the FROM clause where each table has an extra indexed column, the response time when the conditions are favorable are close to the times of having no extra tables in the FROM table. The graph 5.5 shows the results where the times are faster for adding AND conditions to the WHERE clause in a query with six extra tables in the FROM clause. An exception occurs when adding eight extra conditions, probably the query optimizer does not work so good when some contradictions appear in the WHERE clause.
However when the conditions are connected only with OR or AND NOT the response times with an indexed column are like the response time with no index in the tables. There is no a better performance in such cases.

![Figure 5.5: Response time when increasing the number WHERE clause, where the WHERE clauses are connected with AND.](image)

### 5.4 Conclusions

The results confirm the expected hypothesis, the more conditions added to the WHERE clause, the faster is the query execution time or the more possibilities has the MySQL query optimizer to improve the execution time. Good news are the fact of adding conditions to the WHERE clause, because it does not increase the response time, it is sustained around the original execution time. As expected, the fact of adding tables in the FROM clause increase considerably the response time, but fortunately the more tables added to the FROM clause, the more possibilities to increase the number of conditions in the WHERE clause, that is, probably the fact of adding tables to the FROM clause does not cause an important fall of the performance, even with 2 extra added tables the times could be faster than the original execution time.

The response time will give a very good performance when a strong condition is added, for instance, a condition that is satisfied as maximum for the 10 percent of the records will decrease severely the response time, making even faster or at least not slower the fact of adding
tables to the FROM clause. On the other hand, adding very weak conditions to the WHERE clause does not improve significantly the performance, for instance, adding conditions that are satisfied for more than 95 percent of the records.

Adding indexes to columns improve considerably the performance when such columns appear in the WHERE clause. Therefore the rule designer should consider to add index to the columns which can get profit of this situation.

As a main conclusion I would highlight that fined-grained access control does not imply a significant time cost, even it could give a better performance. For instance, adding strong conditions to the original query, togethuer with adding extra tables to the FROM clause, could have a better or at least not worst performance as the original query. The proposal of adding extra tables to the FROM clause where the join conditions are based on columns without an index must be ruled out, because the execution time are excessive.

5.5 Algorithm times

The system read the incomming queries from a text file, therefore a hundred of queries in the input file have been written and the average time was around 3 ms for each query. Comparing this time with the query execution time, we can assure that it is negligible. However the less queries are in the input file the more average time is obtained. A deep analyse showed that for the first query the time is over 100 ms, this is due to the Parser process which connects to the database in order to get information from the database, such as the name of the database where the user is connected or collect all the columns names for the queries with the start (e.g. SELECT * FROM patient implies a connection to the database to collect all the columns from the patient table).

If the algorithm times are divided in four parts: Parser, Policy_Prefiltering, Database Questions and Query Modification, and the times are measured without measuring the query failures. The average times for the process Policy_Prefiltering and Query Modification are negligible around 3 ms for each one. The Parser average time is around 110 ms for the first query and 15 ms for the rest. The Database Questions process it is just executed when there are any question to ask to the database, that is, when a whole column is protected with conditions, this process needs to connect with the database, since the Parser process needs as well the connection the response time for this process is around 10 ms supposing that all the queries have questions to perform.

As a conclusions from these results we can say that the algorithm time reduce the performance when the input is a single query, because it has an significant execution time (133 ms). However this high execution time is due to the connection to the database, which takes the most of the time of the execution, around 109 ms. On the other hand, each query request need a connexion to the database in order to get an answer, therefore such time must be added sooner or later to the total response time, and without the connection time the algorithm time depends mainly on the Database Question process since the Policy_Prefiltering and Query Modification times are negligible.
Chapter 6

Conclusions and Future Work
This work describes the basic concepts of access control and investigated different issues concerning the access control system. In particular, the current Relational Databases Management System Access Control and the needs for providing support to access control in open environments, where the identities of the involved parties could be unknown. Such open environment with unknown parties and a vast number of known parties need as well support to a more fine grained access control to the objects of the RDBMS. However, current RDBMS do not provide access control mechanisms that suit this requirements, they are focused to close environments such as companies.

In this work an approach independent of the RDBMS used is presented for the integration of expressive policies in order to provide a fine grained access control mechanism for RDBMS. These policies conditions may fit the RDBMS nature and data content as well as other external conditions such contextual conditions. The process is divided in order to pre-evaluate conditions of the policy engine not depending on the RDBMS and leaving to the highly query optimizer of the RDBMS the query analyse and data access. An implementation with those ideas is here presented as well as an evaluation showing how the cost of this access control layer in time is acceptable for the gained fine grained access control over data.

Since the implementation presents a fake policy engine, which it is possible to manipulate in order to change the implemented policies, the integration with a real policy engine is a future task to be done, thus the possibilities of defining rules and contextual conditions will be hence as well as more usable for the rule designer. Another task of the rule designer is to extract from the rules columns which the data could be more dependant, thus giving an optimized access to such columns the performance of the new system will be highly improved (as explained in the evaluation section).

A farther research on the thema should focus at first on the writing queries, that is, the queries which insert data in the database. The current work is focused on the SELECT queries which read the information from the database. However queries which modify or insert data in the databases are not evaluate here, in the section 3.1 High level rules introduction is proposed a possible improvement in the rules format in order to support different operations: read, write, or other. If the new format rule look as follows

\[
db(operation, database_name, table_name, column_name, [column1 = value1, \ldots, columnN = valueN])
\]

where the \textit{operation} field may have the following values READ, WRITE or ALL (of course any other research focus would may consider other values in such field). The typical rule which protects the data rows where the column \textit{position} contains the \textit{First Minister} value

\[
deny \text{ access to } db(medical, patient, _, [position = \text{First Minister}])
\]

could be change for the following rule

\[
deny \text{ access to } db(READ, medical, patient, _, [position = \text{First Minister}])
\]

This rule deny the access to all the queries which try to retrieve the information from the database, however if the rule designer wants to deny to write in the rows where the column
value contains the *First Minister* value, the operation field must contains a WRITE value, in this case, since both operations, to read and to write are forbidden by the rule designer, the new rule looks as follows

deny access to db(ALL, medical, patient, _, [position = First Minister])

Considering the incoming accepted queries, an important improvement in the subset of the SQL accepted grammar will be in the column description, right now all columns figuring in the query must be written as follows "table_name.column_name" and must be changed only for the "column_name", in such way the queries will be not so long and the usability from the point of view of the user who writes the queries will be improved. For instance, the following accepted query

```sql
SELECT patient.id, patient.name1, patient.name2, patient.surname1,
       patient.surname2, patient.age, patient.borndate, patient.nationality,
FROM patient
WHERE ((patient.profession = 'second_minister'))
```

is not accepted when is written as follows

```sql
SELECT id, name1, name2, surname1, surname2, age, borndate, nationality,
FROM patient
WHERE ((profession = 'second_minister'))
```

due to the nonexistent table names of the columns. However when a query is modified the fact of adding the tables as a prefix to each column does not represent a problem for the user, furthermore it is neccesary when the query is modified for instance as follows.

```sql
SELECT id, patient.name1, disease.name,
       surname1, surname2, age, borndate, nationality,
FROM patient, disease
WHERE ((profession = 'second_minister') AND
       (patient.id_disease = medical.id_disease))
```

Regarding the algorithm execution time, there are some improvements to do. The main goal on a future work should focus on avoiding the connection to the database during the execution of the algorithm. The *Parser* and *Database Questions* processes connect to the database. The *Parser* process allways connects to get the name of the database where the requester wants to access and collect the columns when the query request ask for the columns with the start sign (e.g. SELECT * FROM patient). A possible solution to avoid the *Parser* process connection is reading in advance the databases and tables information, in such way only when a modification is done in the database the data must be updated reading again from the database. More difficult is to avoid the connection from the *Database Question* process which determines if a whole column is protected or not for a given requester, this process connects to the database only if the access to a column is protected/allowed based on conditions, querying for such conditions to the database. A possibility could be removing the option to deny/allow the access to a column based on conditions, which will decrease the access control expressiveness. Other possibility is to research a new solution based on query modification as done for the other options, it implies to study the SQL syntax possibilities and to analyse again the possible use cases and situations in order to modify the query correctly and to make sure it works in all the possible situations.
Bibliography


