Inline Views: Protecting against SQL Injection Attacks while Providing Access to Aggregate Values

BY

PAOLO BRUZZO
B.S, Politecnico di Milano, Milan, Italy, 2013

THESIS

Submitted as partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Chicago, 2016

Chicago, Illinois

Defense Committee:

Prasad Sistla, Chair and Advisor
V.N. Venkatakrishnan
Pier Luca Lanzi, Politecnico di Milano
ACKNOWLEDGMENTS

I want to thank my UIC advisor Prasad Sistla for believing in my capabilities and giving me the opportunity to work on this awesome project at UIC, and my Polimi advisor Stefano Zanero for his precious advice and for finding time to advise me in his busy schedule. I also want to thank all the members of my family, especially my parents, who have always pushed me to do my best, and without whom I wouldn’t have had the possibility of enjoying such an amazing experience in Chicago; they have been an essential mental, financial and educational back-up. Moreover, a special thanks goes to my girlfriend Jewel who has been a huge moral support during the development of this work, in particular during some tough moments; and to all the amazing people that I met during the last year and contributed to this work by sharing ideas, thoughts and critics. In particular I cannot avoid to mention those who I mostly shared the Chicago experience with (names sorted with the python random.shuffle() function): Dario, Sarah, Teo, Haihua and Gian.

PB
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Structure of the document</td>
<td>2</td>
</tr>
<tr>
<td>2 BACKGROUND</td>
<td>3</td>
</tr>
<tr>
<td>2.1 SQL Injections</td>
<td>3</td>
</tr>
<tr>
<td>2.1.1 The basic OR injection</td>
<td>4</td>
</tr>
<tr>
<td>2.1.2 The database is not read-only</td>
<td>5</td>
</tr>
<tr>
<td>2.1.3 Finding the names of the tables</td>
<td>6</td>
</tr>
<tr>
<td>2.1.4 The UNION injection</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Mitigations</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1 Sanitize the inputs</td>
<td>7</td>
</tr>
<tr>
<td>2.2.2 Quote saving</td>
<td>8</td>
</tr>
<tr>
<td>2.2.3 Bound parameters</td>
<td>9</td>
</tr>
<tr>
<td>2.2.4 Fine Grained Access Control</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Problem definition</td>
<td>10</td>
</tr>
<tr>
<td>2.3.1 State of the art solutions</td>
<td>11</td>
</tr>
<tr>
<td>2.3.2 Goal</td>
<td>12</td>
</tr>
<tr>
<td>3 APPROACH</td>
<td>13</td>
</tr>
<tr>
<td>3.1 Security policies</td>
<td>13</td>
</tr>
<tr>
<td>3.2 Query rewriting and modification technique</td>
<td>14</td>
</tr>
<tr>
<td>3.2.1 Common rules</td>
<td>15</td>
</tr>
<tr>
<td>3.2.2 Simple Policies</td>
<td>16</td>
</tr>
<tr>
<td>3.2.3 One-One aggregate policy</td>
<td>18</td>
</tr>
<tr>
<td>3.2.3.1 Avg</td>
<td>19</td>
</tr>
<tr>
<td>3.2.3.2 Sum - Max - Min</td>
<td>24</td>
</tr>
<tr>
<td>3.2.3.3 Count</td>
<td>26</td>
</tr>
<tr>
<td>3.2.3.4 Other variants</td>
<td>29</td>
</tr>
<tr>
<td>3.2.4 One-Many aggregate policy</td>
<td>31</td>
</tr>
<tr>
<td>3.2.4.1 Avg</td>
<td>31</td>
</tr>
<tr>
<td>3.2.4.2 Sum - Max - Min</td>
<td>34</td>
</tr>
<tr>
<td>3.2.4.3 Count</td>
<td>37</td>
</tr>
<tr>
<td>3.2.5 Many-One and Many-Many aggregate policies</td>
<td>39</td>
</tr>
<tr>
<td>4 IMPLEMENTATION</td>
<td>41</td>
</tr>
<tr>
<td>4.1 System overview</td>
<td>41</td>
</tr>
<tr>
<td>4.2 System front-end</td>
<td>43</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (continued)

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.1</td>
<td>Initial design and characteristics</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Implementation details</td>
</tr>
<tr>
<td>4.3</td>
<td>Policy file structure</td>
</tr>
<tr>
<td>4.4</td>
<td>System back-end</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Setup Requirements</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Query modification process</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Tables replacement</td>
</tr>
<tr>
<td>5</td>
<td>EVALUATION</td>
</tr>
<tr>
<td>5.1</td>
<td>Test machine</td>
</tr>
<tr>
<td>5.2</td>
<td>Structure of the tests</td>
</tr>
<tr>
<td>5.3</td>
<td>Performance impacts</td>
</tr>
<tr>
<td>5.3.1.1</td>
<td>Test 1: selection of 90% of the tuples</td>
</tr>
<tr>
<td>5.3.1.2</td>
<td>Test 2: selection of 5% of the tuples</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Aggregate queries</td>
</tr>
<tr>
<td>5.3.2.1</td>
<td>Test 3: avg, sum, min, max</td>
</tr>
<tr>
<td>5.3.2.2</td>
<td>Test 4: count</td>
</tr>
<tr>
<td>5.4</td>
<td>Performance improvements</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Oracle database</td>
</tr>
<tr>
<td>5.4.1.1</td>
<td>Test: inline view with high DT and low FT</td>
</tr>
<tr>
<td>6</td>
<td>CONCLUSIONS</td>
</tr>
<tr>
<td>7</td>
<td>FUTURE WORK</td>
</tr>
<tr>
<td>APPENDIX</td>
<td></td>
</tr>
<tr>
<td>CITED LITERATURE</td>
<td></td>
</tr>
<tr>
<td>VITA</td>
<td></td>
</tr>
<tr>
<td>TABLE</td>
<td>PAGE</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>I</td>
<td>15</td>
</tr>
<tr>
<td>II</td>
<td>18</td>
</tr>
<tr>
<td>III</td>
<td>20</td>
</tr>
<tr>
<td>IV</td>
<td>25</td>
</tr>
<tr>
<td>V</td>
<td>27</td>
</tr>
<tr>
<td>VI</td>
<td>28</td>
</tr>
<tr>
<td>VII</td>
<td>32</td>
</tr>
<tr>
<td>VIII</td>
<td>52</td>
</tr>
<tr>
<td>IX</td>
<td>52</td>
</tr>
<tr>
<td>X</td>
<td>61</td>
</tr>
<tr>
<td>XI</td>
<td>64</td>
</tr>
<tr>
<td>XII</td>
<td>67</td>
</tr>
<tr>
<td>XIII</td>
<td>68</td>
</tr>
<tr>
<td>XIV</td>
<td>75</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>DESCRIPTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>System overview</td>
<td>42</td>
</tr>
<tr>
<td>2</td>
<td>Front-end main panel</td>
<td>45</td>
</tr>
<tr>
<td>3</td>
<td>Front-end simple policy panel</td>
<td>46</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
<td></td>
</tr>
<tr>
<td>DBA</td>
<td>Database Administrator</td>
<td></td>
</tr>
</tbody>
</table>
SUMMARY

In this work we describe a novel technique applicable to already existing web applications, in order to protect them against classic SQL injection attacks. We focus on the definition of security policies from the point of view of the DBA, and their automatic transformation into correct SQL statements that will act as an inline view, and replace every occurrence of the table that needs to be protected; in this way each query will only target the data that can be accessed by the application user, instead of the original database table. The novelty of this work consist into the definition of certain rules that allow not only to transform simple policies into the actual temporary views, but also allow to retrieve aggregate results (such as sum or average of other data) in a correct way; a simple drop in replacement of a view that only makes available a subset of the original data, would in fact return wrong results in case of selection of aggregate values. With this work we want to show the feasibility of this approach, and the performance impact that it brings to the target applications.
CHAPTER 1

INTRODUCTION

Web applications security has been a huge topic of study in the last years and many techniques have been developed to protect the access to the data stored in databases that are queried by the applications users; this data is subject to unwanted disclosure through injections performed by an attacker if the application code hasn’t been carefully written. Many modern tools help a programmer to enforce some basic security checks and use common security patterns, anyway there is still a significant number of legacy web applications that have been written without particular concern about security, and need to be fixed without modifying the source code or the database schema, that would require an amount of resources not always available. We propose a solution to this issue that aims to add an intermediate filter in between the web application and the database, to modify at runtime the queries that are issued, such that they respect access control policies specified by the administrator. The filter will intercept every SQL query issued from the web application, and replace each occurrence of the tables that needs protection with an inline view that will only provide the data that the user is allowed to access. The proposed solution will aim to reduce at the minimum the impact on the original web application, both from the point on view of source code modification and the runtime performance.
1.1 Structure of the document

The thesis is structured like follows: in Chapter 2 we give a background overview of the problems caused by sql injections, some practical examples of how sensitive data can be disclosed, a brief overview of our approach and the comparison with already existing techniques. In Chapter 3 we explain in details the theoretical work done to come up with the proposed solution, in Chapter 4 we show the implementation details of our approach, and in Chapter 5 we show the performance impact and how it affects the user experience. Chapter 6 will discuss the conclusions, and Chapter 7 will report some future works and ideas to further improve our tool.
CHAPTER 2

BACKGROUND

2.1 SQL Injections

SQL injection is a technique used to attack web applications by injecting SQL commands that can compromise the security of the application. The first discussion about this kind of attack started appearing in 1998 \[1\], when Jeff Forristal explained in a video interview how he came across SQL injections, and even after so many years, he is not surprised that they are still a widely spread concern, and in 2013 it’s been classified the number one on the OWASP classification \[2\]. In order to be a successful attack, SQL injections needs to exploit a vulnerability in the web application code, that for example, doesn’t correctly filter user inputs, which are typically taken from uncontrolled web forms fields. SQL injections can be classified in four main sub-classes:

- **Classic**: the attacker can directly obtain the information from a web application output
- **Blind**: the attacker can’t directly see as output the information she is trying to obtain, but she can infer it through a series of different responses driven by different injected logical statements
- **DBMS specific**: the attacker can typically escalate privileges by passing SQL commands as parameters of stored procedures that are executed in a particular context
• Compounded: the injection is combined with other common web application attacks such as DDoS attacks, DNS Hijacking, XSS etc...

Classic SQL injections are the easiest to perform in terms of complexity; we will focus this chapter on how to use them to attack a data-driven web application, and we will show standard techniques on how to ensure protection against them.

2.1.1 The basic OR injection

This technique is probably the first thing an attacker wants to try to test how a web application responds to a basic injection; it requires almost no effort given its simplicity. Let us consider a web application that queries a database to retrieve some information of a certain user that inputs his username and password:

Listing 2.1: Original query

```sql
1 SELECT name, lastname, email, address, phone, salary
2 FROM users
3 WHERE username = 'USER' and password = 'PASS';
```

If the inputs are not correctly sanitized, an attacker can insert a string in the username field that is parsed as an SQL statement and can bypass the password check. For example with the string 'OR 1=1 ;' as username input, the query becomes like follows:
Listing 2.2: OR injection

```
1 SELECT name, lastname, email, address, phone, salary
2 FROM users
3 WHERE username = '' OR 1=1 ;-- ' and password = ''; 
```

The query is parsed up to the first semicolon input by the attacker, and the rest in considered as a comment string. This allows to retrieve the information of each user, since 1=1 is a tautology.

2.1.2 The database is not read-only

Another tragic attack that can take place if the underlying database can be modified by a user, consists in sending a second query together with the first one that is also parsed as SQL statement. For example, exploiting the vulnerability of the non sanitized inputs of Listing 2.1, an attacker can inject the code to delete a table like shown in Listing 2.3.

Listing 2.3: Drop table injection

```
1 SELECT name, lastname, email, address, phone, salary 
2 FROM users
3 WHERE username = ''; DROP TABLE users ;-- ' and password = ''; 
```

If the web application allows that, the statement will bypass the password field thinking it’s a string comment, both the queries will be executed and the users table will be deleted. Probably the attacker doesn’t know the names of the tables, and she can try to find them exploiting the
same vulnerability through a different SQL injection that applies the same concept described so far. An example is given in the following section.

### 2.1.3 Finding the names of the tables

There are several ways to find a table name by injecting SQL code, we are going to describe one that uses a sub-select to guess them like shown in Listing 2.4.

```sql
1  SELECT name, lastname, email, address, phone, salary
2  FROM users
3  WHERE username = ' ' AND 1=(SELECT count(*) FROM items) ;-- ' and password = ' ';
```

We don’t care of how many tuples does the `items` table contain, if it exists the query will be parsed and executed, while if it doesn’t exist, it will return an error.

### 2.1.4 The UNION injection

Let us say that the attacker found a table name, and he is interested in retrieving sensitive information from it, but the only exploitable input doesn’t affect a query on that table. A powerful solution is to inject SQL code that appends to the original result, other tuples selected from other tables with the UNION clause. An example is given in Listing 2.5 where the attacker wants to get the usernames and passwords of the administrators of the web application.
Listing 2.5: Find table name injection

```
SELECT name, lastname, email, address, phone, salary
FROM users
WHERE username = '' UNION ALL SELECT username, password, 0, 0, 0 FROM administrators
    ;-- ' and password = '';
```

The four zeros are necessary to select the same number of attributes selected by the original query, otherwise the UNION operation would fail. This injection is very powerful if the attacker knows the database schema, since it could allow the retrieval of any desired information.

## 2.2 Mitigations

In this section we are going to describe the most adopted mitigations to protect data-driven web applications from SQL injection attacks. We begin with the description of the most common patterns that are followed when code is written with security concerns, and we finish with the introduction of the state of the art solutions to enhance the security of already existing web applications, that are the starting point of our study.

### 2.2.1 Sanitize the inputs

The easiest mitigation to avoid the injection of SQL code, consist in blacklisting all those characters that are common in the SQL syntax, and are useless for the purpose of a certain inputs of the web application; quotes, escapes or semicolon could be marked as dangerous character and not be allowed in the input of a username. This approach is surly a good starting point, but while it’s easy to blacklist some of the dangerous characters, it’s impossible to list all
of them. A better approach would be to sanitize the inputs allowing only the good characters: for example the input of an email address could be checked with a regular expression that only allows letters, numbers and few others symbols (@ - _ +). The whitelisting approach is surly an improvement, but still doesn’t really solve the problem since even allowed characters can be troublesome: let us consider a selection filtered by a numeric PIN, even avoiding special characters allow to inject dangerous code like shown in Listing 2.6.

**Listing 2.6: Regular characters attack**

```sql
1 SELECT attribute1, attribute2, attribute3 
2 FROM table 
3 WHERE PIN = 12345678 OR TRUE ;
```

2.2.2 Quote saving

Quote saving is surly a better approach than the previous ones; sanitizing a username field with a whitelist of characters can be enough, but could fail in an address field where strings such as *O’Connell Street, Dublin* should be allowed. Quote saving can help in this situations by doubling the quotes, but becomes difficult to handle with database systems such as MySQL that allow quote escaping themselves; for example the drop table attack could still be performed by putting a backslash in front of the first input quote: the latter would be doubled by the systems obtaining the following query.
Listing 2.7: Drop table injection with quote saving

```sql
SELECT name, lastname, email, address, phone, salary
FROM users
WHERE username = ' \''; DROP TABLE users ;-- ' and password = ' ;
```

The username becomes the single quote, and the drop table query is still correctly executed.

### 2.2.3 Bound parameters

This is probably one of the most important steps if we really want to protect a web application from injection attacks. While the previous approaches were concatenating the user input to the SQL query, and then parse and execute the final string, the bound parameters allow to consider the user input as only ”data” that has nothing to do with the SQL statement. Nowadays this technique is supported by nearly all the database interfaces, and gives enormous security benefits.

Listing 2.8: Bounded parameters

```sql
SELECT name, lastname, email, address, phone, salary
FROM users
WHERE username = '$USER' and password = '$PASS';

$USER = "Whatever you input here, is only data!"
$PASS = "I'm also bounded!"
```
In Listing 2.8 we can observe that the username and the password are two parameters with an assigned string (the user input) that will be treated as "simple data" and not as part of the SQL query.

### 2.2.4 Fine Grained Access Control

Other than enforcing security at the application layer, it can become very useful to exploit the powerful access control functionalities of the underlying DBMS. A common pattern to restrict the access to the database resources is the use of GRANT and REVOKE privileges to read, update, delete etc ...only certain tables by certain users; for example a normal user might only have the privileges or reading the *items* table, this can be obtained as shown in Listing 2.9.

**Listing 2.9: Database privileges**

```sql
1. REVOKE ALL ON *.* FROM 'user@host';
2. GRANT SELECT ON database.items FROM 'user@host';
3. FLUSH PRIVILEGES;
```

### 2.3 Problem definition

All the mitigations described so far are a pattern to follow during the development of a data-driven web application. The issue comes if we want to protect an already existing web application and be minimally invasive on the source code. It’s well known that a legacy application gives the advantage of having a highly tested code, and any change must be carefully planned to avoid unexpected and new malfunctions; instead of modifying the source code, the
database schema or any access control policy that directly affect the system, we want to explore the possibility of adding extra features to enhance the data protection.

2.3.1 **State of the art solutions**

The current solutions to this problem can mainly be classified into two areas: query modification techniques and query validity checks. The first idea of query modification started in 1975 [3] with the purpose of guaranteeing to have no integrity violations in the definition of the queries. This approach has been extensively studied during the years, with the first application to security purposes in 2004 [4] where the authors incorporated privacy policy enforcement into an existing data-driven application environment. This study has been highly considered in the following years, and adapted to slightly different topics such as information leakage through channels like user-defined procedures [5] and access control methods enhanced with parameterized views [6]. Anyway it’s also been criticized in the same year for the performance impact and the erroneous results it might generate if not correctly applied [7], and the second method of query validity check has been proposed. Instead of modifying the query to retrieve a result that respects the privacy policies, they propose a solution in which they classify a query as valid, ”if it can be answered using only the information contained in the authorization views available to the user”. The drawbacks of this approach are the higher source code modification required to make it work as expected, and the huge complexity in case of extension to handle aggregates; for this reason we preferred to take the cue from the study of query modification techniques and extend them with our own research deeply explained in Chapter [3].
2.3.2 Goal

The goal we want to achieve is to introduce an additional protection layer in between the web application and the database level to enhance the protection against classic SQL injections, that has to be minimally invasive on the source code and the performance of the application.
CHAPTER 3

APPROACH

In this chapter we will discuss about the query modification techniques that can be exploited in order to protect a database from unwanted access to sensitive data. Our analysis takes the cue from two important works on this topic: the first one \cite{4} puts the basis of query modification for a cell level access control, while the second one \cite{7} is more focused on the extension of those concepts with transformation algorithms that combine multiple restrictions on the same table. Our method will take some of those parts and extend them with other techniques to handle aggregates. In the following sections I’ll present our approach for the definition of security policies and a novel approach for allowing to retrieve aggregated results without disclosing individual tuples.

3.1 Security policies

Many solutions have been discussed on the different ways in which security policies can be specified: from traditional languages like Transaction Datalog \cite{8} \cite{9} to more specific approaches like P3P \cite{10} or EPAL \cite{11}. These solutions are mainly adopted to express high level privacy statements on web sites and can be interpreted by web browser for security checks; in our specific case we don’t need the majority of their features and we will limit our specification to the definition of SQL queries, that will be taken as they are and used as a view that represent a table restriction. Therefore our policies are nothing more than a SQL select statement that
retrieves what can be seen by the web application user, and is used later on as a target view that replaces the original table that the policy itself is protecting. All the further characteristics that surround the policies (e.g. start and expiration date) will be discussed in Chapter 4.3.

3.2 Query rewriting and modification technique

The complexity of the policy can vary depending on many factors: it can be a simple select statement that filters out some tuples of a table, it can be a more complex query that also selects aggregated values, or it might even need to combine different aggregates of different columns. The query modification process described in this section explains how to handle case by case the writing process of these policies and we will make available a set of rules that have to be followed in order to correctly protect the table as we want, without leaking sensitive information and especially without changing the correctness of the result seen at the application level. The automation of this process will be later discussed in Chapter 4 here we will present the theoretical concepts of this technique.

To introduce each technique I will always refer to the example table reported in Table I after which I’ll generalize and globally extend the solution.
### TABLE I: EXAMPLE DATABASE TABLE - ITEMS

<table>
<thead>
<tr>
<th>book</th>
<th>price</th>
<th>category</th>
<th>author</th>
<th>books_sold</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>PA</td>
<td>CA</td>
<td>AA</td>
<td>BA</td>
</tr>
<tr>
<td>B</td>
<td>PB</td>
<td>CB</td>
<td>AB</td>
<td>BB</td>
</tr>
<tr>
<td>C</td>
<td>PC</td>
<td>CC</td>
<td>AC</td>
<td>BC</td>
</tr>
<tr>
<td>D</td>
<td>PD</td>
<td>CD</td>
<td>AD</td>
<td>BD</td>
</tr>
<tr>
<td>E</td>
<td>PE</td>
<td>CE</td>
<td>AE</td>
<td>BE</td>
</tr>
<tr>
<td>F</td>
<td>PF</td>
<td>CF</td>
<td>AF</td>
<td>BF</td>
</tr>
</tbody>
</table>

#### 3.2.1 Common rules

Even if we are going to study case by case how to write the policies, there is a simple correctness rule that must be satisfied in every case: a query that retrieves a legal result from the original database, must always retrieve the same legal result when the tables are protected by the policies. There are two main situations in which it’s important to take care of it:

- When protecting a single table: it’s fundamental not to filter out any necessary attribute that needs to be selected or used by the original query. For example if a legal query needs to filter the books of Table I by category, than each policy involved in the protection of that table must select the category attribute to make it available. This has to happen even if the attribute will not be visible to the end user.
• When protecting a table joined with other tables: in this case it’s fundamental to respect what we call the one-to-one lossless join property. The concept has already been introduced by a previous study [12] and states that "A join $R \bowtie S$ is lossless with respect to $R$ if it preserves all rows of $R$. ... A join $R \bowtie S$ is a one-to-one join if a row from $R$ joins with at most one row from $S$. A join $R \bowtie S$ is one-to-one lossless with respect to $R$ if the join is lossless with respect to $R$ and one-to-one with respect to $R$.". In other words: a join itself has to preserve all the tuples involved and cannot add other ones (e.g. cross join), only a where clause will possibly filter the result.

3.2.2 Simple Policies

The most simple policy that the web application administrator may want to write, is filtering some tuples of a table with a where clause. Oracle Virtual Private Database [13] has a built in functionality that already applies this kind of filter when the where condition is common to every column of the table; it’s an fine grained access control method at row level. Anyway the protection of a table might need a cell level access control as explained in the Hippocratic Database work [4], and we adapt their solution using the case statement in our work. Let us say that one wants to select all the book IDs, the price only if less than 50$ and the category of book C from Table I the three policies that will be specified are reported in Listing 3.1
Listing 3.1: Simple policies

1. SELECT book FROM items;
2. SELECT price FROM items WHERE price <= 50;
3. SELECT category FROM items WHERE book like 'C';

The merging process will build a unique policy reported in Listing 3.2:

**Listing 3.2: Simple policies merged**

```sql
SELECT
    book AS book,
    CASE WHEN price <= 50 THEN price ELSE NULL END AS price,
    CASE WHEN book like 'C' THEN category ELSE NULL END AS category,
    NULL AS author,
    NULL AS books_sold
FROM items
WHERE (TRUE) OR (price <= 50) OR (book like 'C');
```

In general: each attribute is selected with its own condition through the `case` statement, the attributes not specified are selected as `null` and the final result is filtered with all the conditions in `or`. Note that any optimization step on the query is left to the database, especially the ones that require algebraic simplifications like having `true` in `or` with other conditions. The final result we would obtain with the example discussed is shown in Table II, where PC and PD are prices greater than 50$. 


3.2.3 One-One aggregate policy

A more interesting case comes when there is the need of extracting an aggregate value from a dataset without revealing the individual tuples that are aggregated. With one-one we indicate a single aggregate among avg, sum, min, max and count that acts over a single column of the table, more complex cases will be described in the following sections. Since we are describing the process of how policies relative to the same table can be merged, we are not going to describe the case in which a user is only allowed to see an aggregated value; this would be a single query that will be used as replacement of the protected table. Very simple and practical examples of the scenario that we are describing are: a user is allowed to see all the salaries of the employees of his department, but only the average of the other ones; or a user can only see the number
of books of a certain category, but not their title or author. The general idea is to append a single value to the already selected tuples, that when will be taken into account to compute the aggregate by the original query, will return the correct result.

3.2.3.1 Avg

The first aggregate that we consider is \textit{avg}, and we describe the result we want to achieve starting from the usual example of \textit{Table I} assigning some values to the category attribute for a clearer explanation: $CA = 5, CB = 5, CC = 3, CD = 3, CE = 2, CF = 1$. Let us say we want to see all the information about the books of \textit{category} = 5 and the average of the prices of every book with \textit{category} $\geq 2$. The translation into policies is reported in \textit{Listing 3.3}.

\begin{verbatim}
Listing 3.3: Avg one-one: initial policies
1 SELECT * FROM items WHERE category = 5 ;
2 SELECT AVG(price) FROM items WHERE category >= 2 ;
\end{verbatim}

What we want to obtain is a result set (shown in \textit{Table III}) in which the first two books are selected as they are, and a price value PX is appended such that when we try to retrieve the average price of the books with \textit{category} $\geq 2$ we obtain the correct result.
TABLE III: EXAMPLE RESULT WITH AVG - ITEMS VIEW

<table>
<thead>
<tr>
<th>book</th>
<th>price</th>
<th>category</th>
<th>author</th>
<th>books_sold</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>PA</td>
<td>5</td>
<td>AA</td>
<td>BA</td>
</tr>
<tr>
<td>B</td>
<td>PB</td>
<td>5</td>
<td>AB</td>
<td>BB</td>
</tr>
<tr>
<td>null</td>
<td>PX</td>
<td>3</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

Let us call $N_1$ the number of tuples selected by the first policy ($N_1 = 2$ in this case) and $N_2$ the remaining tuples used by the second policy to compute the average ($N_2 = 3$ in this case). Note that $N_2$ cannot be negative, otherwise it would mean that we want to compute the average of a subset of elements of the first query, which doesn’t need any additional policy to be correctly computed. The average price retrieved from the merged policy will be the same of the original one only if:

\[
\frac{PA + PB + PC + PD + PE}{N_1 + N_2} = \frac{PA + PB + PX}{N_1 + 1}
\]

which resolved for $PX$ gives:
\[ PX = \frac{PA + PB + PC + PD + PE}{N_1 + N_2} + \]
\[ + \frac{(PC + PD + PE) \times N_1}{N_1 + N_2} - (PA + PB) \times N_2 \]

Now let us generalize it and call:

- \( P_1 = \sum_{k=1}^{N_1} P_k \)
- \( P_2 = \sum_{k=1}^{N_2} P_k \)
- \( P_1 + P_2 = \sum_{k=1}^{N_1} P_k + \sum_{k=1}^{N_2} P_k = \sum_{k=1}^{N_1+N_2} P_k \)

In our case \( P_1 = PA + PB \) and \( P_2 = PC + PD + PE \). So \( PX \), that is the price value we want to append, results to be:

\[ PX = \frac{(P_1 + P_2) + (P_2 \times N_1) - (P_1 \times N_2)}{N_1 + N_2} \]

Now we need a way to compute \( N_1, N_2, P_1 \) and \( P_2 \) in SQL, calculate \( PX \) and append it in the view. \( N_1 \) and \( P_1 \) simply correspond to the count and the sum of price values selected by the first policy, while \( N_2 \) and \( P_2 \) correspond to the count and the sum of price values selected by the second policy minus those selected by the first one. \( PX \) can be calculated is a single inline selection and then appended using the union clause. Listing 3.4 shows the resulting merged policy of the explained example.
The first important thing to notice is the use of the `nullif` keyword at line 5-8 when $PX$ is computed: since SQL aggregates completely discard null values, in order to avoid mistakes in the computation of the average by diving the sum with the count, we replace every occurrence of null with a zero; we are not invalidating any result because from the point of view of the end user it doesn’t affect the result. A second fundamental point is the selection of one value of the
category attribute at line 9-11: without it, the original query would retrieve a wrong value if it tries to filter with \( \text{category} \geq 2 \); in general we can say that every attribute that appears in the aggregate policy has to be selected. In this case the selected value will be a random value among those that are selected as not belonging to \( \text{category} = 5 \). A last concept that deserves a comment is the right join at line 17: this operation is not joining two tables of the database, it’s simply a concatenation of the four values \( N_1, N_2, P_1, P_2 \) in a single row temporary table. The need of specifying the right join instead of a normal join is not necessary in this case, it becomes essential in the case we also want to group by an attribute: let’s say we want to group by author, lines 15-19 of Listing 3.4 have to be replaced with Listing 3.5.

Listing 3.5: Avg+group by one-one: merged policy

```
1 (SELECT author, COUNT(price) as N1, SUM(price) as P1
2     FROM items where category >= 2 AND category = 5 group by author) t1
3      RIGHT JOIN
4 (SELECT author, COUNT(price) as N2, SUM(price) as P2
5     FROM items where category >= 2 AND NOT (category = 5) group by author) t2 ;
6     ON t1.author = t2.author ;
```

In this case there is the need to match the authors of the first selection with the ones of the second selection, preserving all the ones of the second selection. What we are basically saying, is that \( N2 \) and \( P2 \) values of the tuples selected by the second policy (grouped by author), need to be appended to the final result even if they don’t match any author already selected by the first policy.
We treat these three aggregates in a unique section since they all require the same merging process; the explanation will only treat the *sum* for clarity but it can be easily extended also to *min* and *max*. Let us say that a user can access all the information about books of *category* = 5, but can only see the sum of all the books sold with *category* ≥ 2; the policies that will be written to express it are reported in *Listing 3.6*.

Given the power of SQL, the complexity of this merging becomes much easier with respect to the one discussed for the *avg*. The associative property of the sum allow us to simply calculate the sum of all the books sold with *category* ≥ 2 that haven’t already been selected as individual tuples (those with *category* = 5 in this case), and appended it as a single value *BX*. The resulting view is reported in [*Table IV*](#). In this case *BX* = *BC* + *BD* + *BE*, and when the original query will compute the sum of books sold whose *category* ≥ 2, the result will be given by the sum *BA* + *BB* + *BX* which is equivalent to the one selected from the unprotected table (thus correct). *Listing 3.7* shows how to computer it in SQL in the resulting merged policy.
TABLE IV: EXAMPLE RESULT WITH SUM - ITEMS VIEW

<table>
<thead>
<tr>
<th>book</th>
<th>price</th>
<th>category</th>
<th>author</th>
<th>books_sold</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>PA</td>
<td>5</td>
<td>AA</td>
<td>BA</td>
</tr>
<tr>
<td>B</td>
<td>PB</td>
<td>5</td>
<td>AB</td>
<td>BB</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
<td>3</td>
<td>null</td>
<td>BX</td>
</tr>
</tbody>
</table>

Listing 3.7: Sum one-one: merged policy

```
1 SELECT * FROM items WHERE category = 5
2     UNION ALL
3 SELECT
4     NULL as book,
5     NULL as price,
6     category,
7     NULL as author,
8     sum(books_sold) as books_sold
9 FROM items
10 WHERE category >= 2 AND NOT (category = 5) ;
```

A good thing about this approach (even if not elegant in terms of syntax) is that SQL already selects the category of the first element of the second portion of the policy at line 6, without the need of specifying a further subquery to select it; this saves a full table scan that is necessary in the case of the `avg` (line 9-11 of Listing 3.4).
3.2.3.3 Count

With the count clause the solution has to be different. This aggregate counts how many tuples are present in the table, so we cannot simply append one value, but we have to make available a number of tuples that actually matches the original correct value. Let us consider again the example of Table II with \( CA = 5, CB = 5, CC = 3, CD = 3, CE = 2, CF = 1 \): we allow the user to see all the information of the books of category = 5 and only the number of books of category \( \geq 2 \). The policies that express it are reported in Listing 3.8.

Listing 3.8: Count one-one: initial policy

```sql
1 SELECT * FROM items WHERE category = 5;
2 SELECT COUNT(book) FROM items WHERE category >= 2;
```

We need to hide somehow all the values of books with category \( \geq 2 \) and category! = 5, preserving the number of tuples. A possible solution would be to retrieve all null values like shown in Table V.
There are two main issues with this solution. The first one is that we are not showing any information about the category, so when the original query will filter the result by category, the null values will not be counted; on the other hand showing the correct category for each tuple also reveals sensitive information that is not specified in the policies, leading to an unwanted information leakage. The second issue is given by the behavior of count: if the original query will select count(*), than the result retrieved will be correct; if the query selects count(book) than all null values will not be counted, giving a wrong result. To solve the first problem it’s enough to adopt the same solution used with the avg aggregate (line 9-11 of Listing 3.4): an inline subquery to select a random value among the ones not belonging to category = 5. In this case though, the operations of selection, order by, rand() and limit have to be computed for each hidden tuple, resulting in a huge computational time overhead when their number increases;
a preferred solution for this specific case is computing a single value only once, exploiting the cache of MySQL when subqueries are specified in the select clause. When a subquery of this kind is not dependent by external mutable factors (e.g. the use of the rand() function that returns a different result everytime it’s called), MySQL computes the result only the first time and caches it for reuse. The second issue instead can be solved by selecting a fixed fake value (e.g. number zero) that will only be used to return the correct result without showing real data or any permutation of it, when a specific attribute is counted. The resulting table at this point would be the one reported in Table VI.

<table>
<thead>
<tr>
<th>book</th>
<th>price</th>
<th>category</th>
<th>author</th>
<th>books_sold</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>PA</td>
<td>5</td>
<td>AA</td>
<td>BA</td>
</tr>
<tr>
<td>B</td>
<td>PB</td>
<td>5</td>
<td>AB</td>
<td>BB</td>
</tr>
<tr>
<td>0</td>
<td>null</td>
<td>3</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td>null</td>
<td>3</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td>null</td>
<td>3</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

The merged policy that generates this result is the one showed in Listing 3.9.
In contrast to the other aggregates, this solution needs to be reviewed if we want to consider the case of the `distinct` keyword. In such a case, selecting a fixed value for all the tuples like shown in line 4 of Listing 3.9 would reduce in every case the `count` to one since only a single value is present. A possible solution would be to select the hash of the value, hash functions are not biunique but the collision probability of two different values is generally very low for the purpose of this application if we use for example `sha1`. It’s perfectly compatible with MySQL and very easy to use: line 4 would be replaced with `sha1(book) as book`.

### 3.2.3.4 Other variants

In this section we are going to describe some variants relative to the selection of `one-one` aggregates that might give rise to questions. The first variant is the use of the `group by` -
having clause in all the other cases we haven’t mentioned (sum, min, max, count): since the second portion of the query (after the union clause) selects from the original table, and not from temporary values like those generated in the approach with the avg, the only variation that might affect the merging algorithm is the addition of the group by - having clause after the last where clause, as in a common SQL query.

The second variant is the case in which a table to protect needs to be joined with another one: a conclusion of what has been explained in Chapter 3.2.1 is that each table of a query coming from the web application will be replaced with a view that will generally be a subset of elements of the original one. The only reason why we want to replace table R with a runtime view of R >> S is that we need to specify a condition over R that depends on attributes contained in S; anyway the original query is not expecting any attribute of S, so even if present in the view they will never be selected. In case there is the need of joining the protected table with other ones, it’s enough to add the join clause as specified in the policy; all the attributes that the join involves though, are also automatically selected for the same reason explained in the first rule of Chapter 3.2.1.

A last comment is about other common operators like order by and limit: the former is totally useless for the semantic purpose of the policy since it only affects the order of the tuples that might also be overwritten from the web application query, the latter could be used by simply adding it at the end of each policy.
3.2.4 **One-Many aggregate policy**

In this section we describe how to merge the policies that we call *one-many* aggregate: which are those that make available the same aggregate for more than one attribute. For example one may want to protect Table I by making available all the books of category = 5 and both the average of the prices and the books sold of category ≥ 2. Each aggregate has again a different solution, we start analyzing the most simple case for each of them, proceed by taking into account more complex policies and conclude by generalizing the merging process.

3.2.4.1 **Avg**

Let us say we want to add a policy to the example of the *one-one avg* (Chapter 3.2.3.1), and we also want to be able to see the average of books sold with category ≥ 2. This is the easiest case we can expect since both the aggregate policies have the same where clause and the same group by clause (absent in this case). Listing 3.10 shows the specification of these policies.

```
Listing 3.10: Avg one-many: initial policies

1  SELECT * FROM items WHERE category = 5 ;
2  SELECT AVG(price) FROM items WHERE category >= 2 ;
3  SELECT AVG(books_sold) FROM items WHERE category >= 2 ;
```

It’s evident that the solution adopted with the *one-one avg* needs to be extended, and we do it by simply joining the temporary values created to compute the PX and BX values to append to the individual tuples. In this case the resulting view will have to look like Table VII.
TABLE VII: EXAMPLE RESULT WITH AVG ONE MANY - ITEMS VIEW

<table>
<thead>
<tr>
<th>book</th>
<th>price</th>
<th>category</th>
<th>author</th>
<th>books_sold</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>PA</td>
<td>5</td>
<td>AA</td>
<td>BA</td>
</tr>
<tr>
<td>B</td>
<td>PB</td>
<td>5</td>
<td>AB</td>
<td>BB</td>
</tr>
<tr>
<td>null</td>
<td>PX</td>
<td>3</td>
<td>null</td>
<td>BX</td>
</tr>
</tbody>
</table>

To add the computation of $BX$ it’s enough to calculate another pair of $N_1, N_2, P_1, P_2$ that we will call respectively $N_3, N_4, P_3, P_4$ and to match a correct SQL syntax that requires different identifiers. *Listing 3.11* reports the merged query of the discussed example. To generalize the concept we can say that an automatic process that implements this modification, needs to insert the formula to calculate the $X$ value for each attribute involved in the aggregation (lines 5-8 and 13-16 in the example case), has still to select each attribute that appears in the *where* or *group-by* clause with a subquery (lines 9-11 in the example case) and has to append the joins of the temporary values $N_t, N_z, P_t, P_z$ for each aggregated attribute (lines 24-28 in the example case). A good point of this approach is that it requires tiny modifications even for more complex cases in which the original policies that involve aggregates have different *where* clauses and/or *group-by* clauses. If the *where* clauses are different, it’s enough to change them in the final part of the merged policy (lines 25 and 28 in the example case); if the *group-by* clauses are different or only exist for one policy, we have to apply the same modification already described with *Listing 3.5* and right join the temporary tables on any common attribute they have.
Listing 3.11: Avg one-many: merged policy

```sql
SELECT * FROM items WHERE category = 5
UNION ALL
SELECT NULL as book,
    ( (ifnull(P1, 0)+ifnull(P2, 0)) +
    (ifnull(N1, 0)+ifnull(P2, 0)) -
    (ifnull(N2, 0)+ifnull(P1, 0))
) / (ifnull(N1, 0)+ifnull(N2, 0)) as price,

    ( SELECT category FROM items
    WHERE category >= 2 AND NOT (category = 5)
    ORDER BY rand() LIMIT 1 ) as category,
    NULL as author,
    ( (ifnull(P3, 0)+ifnull(P4, 0)) +
    (ifnull(N3, 0)+ifnull(P4, 0)) -
    (ifnull(N4, 0)+ifnull(P3, 0))
) / (ifnull(N3, 0)+ifnull(N4, 0)) as books_sold
FROM
    (SELECT COUNT(price) as N1, SUM(price) as P1
    FROM items where category >= 2 AND category = 5) t1
RIGHT JOIN
    (SELECT COUNT(price) as N2, SUM(price) as P2
    FROM items where category >= 2 AND NOT (category = 5)) t2
RIGHT JOIN
    (SELECT COUNT(books_sold) as N3, SUM(books_sold) as P3
    FROM items where category >= 2 AND category = 5) t3
RIGHT JOIN
    (SELECT COUNT(books_sold) as N4, SUM(books_sold) as P4
    FROM items where category >= 2 AND NOT (category = 5)) t4;
```
3.2.4.2 Sum - Max - Min

Given the simplicity of the solution found for this category of aggregates (*Chapter 3.2.3.2*), also the extension with a further aggregate policy will be comparable. We will treat again only the *sum* in our example, but the solution is exactly the same for *min* and *max*. Starting from the most trivial case, let us add a policy to the ones used in the *one-one sum*.

Listing 3.12: Sum one-many: initial policy

1) SELECT * FROM items WHERE category = 5 ;
2) SELECT SUM(books_sold) FROM items WHERE category >= 2 ;
3) SELECT SUM(price) FROM items WHERE category >= 2 ;

The obvious extension of the solution is the selection of all the attributes involved in the aggregation, like shown in *Listing 3.13* at lines 5 and 8.

Listing 3.13: Sum one-many, same where clause: merged policy

1) SELECT * FROM items WHERE category = 5
2)    UNION ALL
3) SELECT
4)    NULL as book,
5)    SUM(price) as price,
6)    category,
7)    NULL as author,
8)    SUM(books_sold) as books_sold
9) FROM items
10) WHERE category >= 2 AND NOT (category = 5) ;
Unfortunately this requires some changes both when the *where* or the *group-by* clauses are different. In the first case the simplest solution is to select each aggregate in a dedicated select subquery, and then only retrieve a single tuple. *Listing 3.14* is the merged policy of a little variation of the original ones where the prices are summed only when $category \geq 3$:

```sql
SELECT * FROM items WHERE category = 5
UNION ALL
SELECT NULL as book,
(SELECT sum(price) FROM items
 WHERE category >= 3 AND NOT category = 5) as price,
category,
NULL as author,
(SELECT sum(books_sold) FROM items
 WHERE category >= 2 AND NOT category = 5) as books_sold
FROM items
WHERE category >= 2 OR category >= 3
LIMIT 1 ;
```

This solution can be easily extended if the aggregate policies share the same *group-by* clause by simply adding it at the end, but in case they are different, even if the *where* clauses are equal, there is the need of splitting the merged policy into a union of *one-one* aggregates, each of them with its own *where* and *group-by* clause. For example the policies in *Listing 3.15* are merged into *Listing 3.16*.
Listing 3.15: Sum one-many, different group by: initial policy

1. SELECT * FROM items WHERE category = 5;
2. SELECT SUM(books_sold) FROM items WHERE category >= 2 GROUP BY author;
3. SELECT SUM(price) FROM items WHERE category >= 3 GROUP BY category;

Listing 3.16: Sum one-many, different group-by: merged policy

1. SELECT * FROM items WHERE category = 5
   UNION ALL
2. SELECT
3.   NULL as book, NULL as price,
4.   category, author,
5.   SUM(books_sold) as books_sold
6. FROM items
7. WHERE category >= 2 and NOT category = 5
8. GROUP BY author
9. UNION ALL
10. SELECT
11.   NULL as book,
12.   SUM(price) as price,
13.   category,
14.   NULL as author, NULL as books_sold
15. FROM items
16. WHERE category >= 3 and NOT category = 5
17. GROUP BY category
3.2.4.3 Count

The last option is to have different policies that need to count different attributes. The example extends again the one we have used to describe the case of one-one count adding an additional policy, in this case though, if the policies share the same where clause, the solution is identical to the one presented for the one-one count. Therefore we introduce the example in Listing 3.17 by specifying different ones:

Listing 3.17: Count one-one: initial policy

```sql
1 SELECT * FROM items WHERE category = 5 ;
2 SELECT COUNT(book) FROM items WHERE category >= 2 ;
3 SELECT COUNT(books_sold) FROM items WHERE category >= 3 ;
```

Note that the proposed solution (Listing 3.18) is necessary to select an appropriate category value for each policy. In this way it might seem that we are retrieving a higher overall number of tuples, but we need to remember that counting a single attribute is different than counting with \texttt{count(*)}, and the null values are discarded from the computation. Moreover applying directly this solution, makes very easy to modify it when one or more group-by clauses are added: it’s sufficient to append them at the end of the selection they affect.
Listing 3.18: Count one-many: merged policy

```
SELECT * FROM items WHERE category = 5

UNION ALL

SELECT
  0 as book,
  NULL as price,
  ( SELECT category FROM items
    WHERE category >= 2 AND NOT (category = 5)
    LIMIT 1 ) as category,
  NULL as author,
  NULL as books_sold
FROM items
WHERE category >= 2 AND NOT category = 5

UNION ALL

SELECT
  NULL as book,
  NULL as price,
  ( SELECT category FROM items
    WHERE category >= 3 AND NOT (category = 5)
    LIMIT 1 ) as category,
  NULL as author,
  0 as books_sold
FROM items
WHERE category >= 3 AND NOT category = 5
```
3.2.5 Many-One and Many-Many aggregate policies

It’s possible that a more complex query coming from a web application selects more than one aggregate, both on the same column or different columns. The procedures discussed up to now have been developed with the purpose of covering the union of all the queries that select from a certain table, that is replaced in every occurrence with a unique view that satisfies the union of all the constraints. For example if a query $Q$ has its own where condition $W_Q$ and a query $P$ has its own where condition $W_P$, and both of them select from table $T$, we have found solutions to always replace table $T$ with the same view $T_V$ merged from multiple initial policies, such that $T_V$ will allow both the queries $Q$ and $P$ to retrieve the correct result. We showed that this concept is valid for any number of queries that respect the condition to be simple, one-one or one-many. When it comes to have a single aggregate query that selects different aggregates of the same attribute (what we call many-one policy) we still can proceed by modifying the discussed solutions for each combination of the aggregates: for example a query needs to retrieve the sum and the avg of prices of some book, the min and the max, or any other combination. The issue comes if we still want to protect multiple queries (so possibly with different where or group-by clauses) with the same unique view, and use it to replace every occurrence of the protected table. Such an approach becomes unfeasible for two main reasons:

- Complexity: trying to find a process to merge multiple queries of the kind describe in this section, other than becoming quite complex itself, also increases the complexity of
the merged policy in such a way that it might require huge resources to be computed at
the database level.

• Number of different cases: if we think about all the possible combinations we might meet,
especially when it comes to merge many-many aggregate policies, we face a huge number
of ad-hoc solutions to be applied to each case. This other than generating an enormous
quantity of modification solutions, would also require a complex parsing system just to
scan the policies and understand which transformation has to be performed.

This issue can be partially solved by associating each query coming from the web application
with a merged policy that will replace the protected tables involved, still trying to maximize
the number of possible merges to allow to handle a lower total number of policies. Anyway this
solution becomes very handy when the web application sends many simple queries; sometimes
instead we can face very complex queries that retrieve different attributes of different columns
and perform other various mathematical operations among the aggregated values in a single
selection, and the most reasonable way to protect them would be trying to relax the policy
constraints, even if might not guarantee a full protection. In conclusion, while the limited cases
of one-one and one-many policies allowed us to introduce all the techniques to merge them, it
wouldn’t make sense to start listing the enormous amount of different solutions for the many-
one and many-many policies. Ad-hoc solutions can still be found to protect queries kind that
we haven’t covered, by exploiting the basic solutions discussed so far and adapting them to the
specific case: of course if the final solution is manually found, it can be written as unique initial
policy so that no merge process needs to be invoked.
CHAPTER 4

IMPLEMENTATION

In this chapter I’m going to describe the implementation of a system that automatize the merging process of the techniques described in Chapter 3. This tool has been written with the purpose of showing the feasibility of the study made on the query modification techniques, and has been tested over a set of ten different open source jsp applications available on the project repository [14]. It can be easily adopted to protect any Java based web application, but it might require some little modifications to make it work as desired; this version of the tool is meant to be a proof of concept more than a widely adoptable solution for every existing web application. In the next sections I’m going to describe how to patch the applications with this tool, and I will focus the description on the ones available on the repository. The chapter will present in details the characteristics of the tool highlighting its advantages and disadvantages, from the initial design to the final result.

4.1 System overview

The overall system is composed by a front-end policy generator and a back-end filter that is an extension of the MySQL Java connector: the former is used by the web application administrator to generate the policies that will be used to protect the tables, while the latter is the patch that actually merges the policies and replaces the protected tables.
Figure 1: System overview

*Figure 1* is a graphical representation of the whole system. The front end is a user interface that allows to write the policies in a handy way and it’s been implemented to reduce the space of action in which the policy can be written. It’s well known that SQL allows to write queries with different syntax but same semantics, in other words the same result set can be retrieved by writing different queries that have the same meaning. In order to simplify the query modification process applied by the Query Wizard module of the back-end, the front end has been built in
such a way that the SQL syntax is limited but it’s still possible to express all the semantic
needed to apply the merging algorithms presented in Chapter 3. The front-end generates an
XML file that contains the policies and the attributes that will be used by the back-end to
modify the queries as discussed. This file is read when the web server is turned on, the policies
are taken in input by the Query Wizard module of the back-end that merges them and stores
them in memory where they are automatically kept up to date. In Figure 1 the path followed
by a query is represented with the double black arrows: when the web application sends out
a query, the Query Interceptor module of the filter catches it, checks if there is any policy in
memory that has to be applied to the tables used in the query, and the Table Replacement
module replaces them in case a policy is found. The modified query reaches the database, gets
executed, and the result set goes back to the web server through the original MySQL Java
connector.

The rest of this chapter will explain in detail each component of the system, from the initial
project to the final result and characteristics.

4.2 System front-end

The policy generator is the front-end of the system and it’s been implemented for two main
reasons: the first one is the reduction of the space of action in which a query can be written
(as already mentioned above), the second one is to make the process of writing the policies
easier for the web application administrator. The policy generator is able to check that all
the constraints explained in Chapter 3 are satisfied, and that the policies are correct both in
syntax and semantics. In this way the web application administrator can limit his knowledge
of the system to the understating of the main rules without deeply knowing how the query modification works during the policy merging process; hence it will become much easier to upgrade the system in a transparent way to the end user when further transformations will be added or the existing ones will be improved.

4.2.1 **Initial design and characteristics**

The front-end needs an active connection to the database schema that will be associated with the policies to retrieve meta data like table and column names. The mockup of the main panel of the Policy Generator shown in [Figure 2] is table based: it’s composed of a window tab for each table of the schema on which the administrator can specify the policies. The main window is divided into two frames: the upper one is used to add or edit simple policies while the lower one is for the definition of policies that involve aggregates (e.g. the ones that select the `count`, `avg`, `sum`, `min` or `max` of attributes). The reason why they have been separated is mostly a practical simplification both for the user (to keep a certain order of the policies) and the parsing phase of the back-end that doesn’t have to recognize if a query involves aggregates. Once all the policies have been specified it’s possible to export them into the XML format readable by the back-end (DTD at Listing 4.1).
Figure 2: Front-end main panel

Figure 3 shows instead the mokup of the window that pops up when a simple policy is added or edited. On the left side the user sets up the characteristics of the policy: the name that identifies it, the time range of validity (not mandatory) and the user role that the policy refers to. The user roles have been treated as an adaptation of the Bell LaPadula simple security property [15]: this property states that "a subject at a given security level may not read an object at a higher security level". In our access control rule, we say that if an object can’t be
accessed by a subject with security level $S$, than it cannot be accessed by any subject with security level $S_1 < S$ if there is no other policy that states that a subject with security level $S_2 : S_1 < S_2 < S$ has access rights on that object. In other words, if no policy is exactly specified for a subject with security level $S$, then the lowest security level policy $S_3 > S$ will be applied (if any). On the right side of the window instead there is the form that allows to write the desired policy query by limiting the power of SQL syntax, favoring a more simple query modification in the back-end modules.

Figure 3: Front-end simple policy panel
Adding or editing policies that involve aggregates present a very similar interface to Figure 3, with the only difference that the selection of an aggregate is forced and the GROUP BY and HAVING clauses are added.

4.2.2 Implementation details

The entire front-end has been implemented in Python using the PySide Qt framework and it has been called Policy Generator. The decision of using this language and this framework has been driven by the willing of having a cross platform tool and making the GUI implementation as fast as possible to spend more time on the model behind it. The final result is very similar to what has been designed, with some minor changes to the interactions and the addition of some functionality; for example the policy can be tested before being saved to check whether there is any syntax error. This first version of the Policy Generator can be found in the project repository in both the redistributable and source code form.

4.3 Policy file structure

The XML policy file generated by the Policy Generator has the Document Type Definition reported in Listing 4.1. The correctness of the definition is checked in the back-end before the Query Wizard module starts the modification process.
We won’t enter into the details of each element, anyway we will give a limited explanation of the less trivial ones. The columns element is simply a list of all the column names of the table they refer to, and we need it in the back-end to reconstruct the correct order and number of tuples selected in case of a UNION. The relation view element instead is the collection of the policy and all the attributes that characterize it: each policy has an associated user role, a begin date and an expiration date, which are all grouped into the relation view element.

4.4 System back-end

The back-end is the actual core of the whole system, and it’s the part that reproduce in practice the query modification techniques explained in Chapter 3. It’s been implemented in
Java extending the MySQL Java connector code, exploiting the interceptor as the only entry
level of all the queries going to the database; this allows a complete transparency at both the
application and database level. As shown in Figure 1, the Query Wizard module is the one
responsible of reading the policy file, merging the policies and storing them in memory to use
them later in the runtime construct of the view that will replace the table to protect; while the
Table Replacement module is the actual class that performs the inline view substitution. In
the following subsections I’ll present in details the whole mechanism.

4.4.1 Setup Requirements

One of the most important features of the tool is the power of being minimally invasive on
both the database and application level. The database doesn’t have to be modified at all, while
the web application needs the modification of a few lines of code in order to use the extended
version of the MySQL Java Connector instead of the original one, which is the first single entry
point to catch all the queries that go to the database.

The first thing to do at the application level is to unzip the CustomInterceptor archive available
on the project repository and place it in the external libraries folder of the web application.
The custom.jar package is the extension that will filter the queries at the interceptor level and
has to be specified as a parameter in the jdbce connection; Listing 4.2 shows an example of how
to connect a web application to the database using the custom interceptor.
Listing 4.2: Custom jdbc connection example

```java
final String driver="com.mysql.jdbc.Driver";
final String mySQL_inter="jdbc:mysql://localhost:3306/database_name";
// jdbc parameter to specify a custom interceptor
final String custom="?statementInterceptors=custom.Interceptor";

java.sql.DriverManager.registerDriver((java.sql.Driver)(Class.forName(driver)).newInstance());
java.sql.Connection conn = java.sql.DriverManager.getConnection(mySQL_inter+custom, "username","password");
```

The last necessary modifications at the application level are setting the http session information that the filter will use to identify the user of the web application (for the purpose of this work we can consider that the session is not compromised at the time of this set up), and setting the path of the policy file to read. Listing 4.3 shows how it can be done by importing the custom filter into the web application:

Listing 4.3: Session information and policy filepath setup

```html
<%@ page import="custom.SessionInfo"%>
<% Long thread_ID = new Long(Thread.currentThread().getId());
    /* session is an http session variable */
    SessionInfo.setSessionValues(thread_ID, session);
    SessionInfo.setPolicyFilepath("/.../policy.xml");%>
```
This piece of code should be added in the first page loaded of the web application, and allows the interceptor to handle different users that use the application simultaneously. Once these tiny modifications are done, the patch is applied and it will filter and modify every query that it intercepts.

4.4.2 Query modification process

The first back-end module that acts in the back-end is the Query Wizard and it is run only ones when the user session starts. It’s responsible of applying all the modification steps presented in Chapter 3 and saving the modified queries in memory for future access. It doesn’t handle the queries coming from the web application, but only the ones taken from the policy file that have been written with the Policy Generator. The attributes taken into account to merge the policies are:

- Table name
- User role
- Policy kind (simple or aggregate)

The policies are split in groups that share the table name and the user role; then each group is merged into a unique policy applying all the rules presented in Chapter 3 with respect to their kind (simple or aggregate). Only valid policies (whose time range include the current time) are merged together; if they have different time ranges, the farthest initial time and the most recent expiration time are taken. The system also keeps track of the exact time in which a currently invalid policy will become valid or a current valid one expires, and automatically recomputes everything from scratch in a parallel thread in background.
### TABLE VIII: ORIGINAL POLICIES

<table>
<thead>
<tr>
<th>Table name</th>
<th>User role</th>
<th>Start</th>
<th>End</th>
<th>Policy</th>
<th>Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>items</td>
<td>2</td>
<td>10th Aug</td>
<td>14th Dec</td>
<td>X</td>
<td>simp</td>
</tr>
<tr>
<td>items</td>
<td>2</td>
<td>20th Aug</td>
<td>31st Dec</td>
<td>Y</td>
<td>aggr</td>
</tr>
<tr>
<td>items</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>Z</td>
<td>simp</td>
</tr>
<tr>
<td>orders</td>
<td>1</td>
<td>10th Aug</td>
<td>-</td>
<td>T</td>
<td>simp</td>
</tr>
<tr>
<td>orders</td>
<td>1</td>
<td>-</td>
<td>14th Dec</td>
<td>P</td>
<td>simp</td>
</tr>
<tr>
<td>cards</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>Q</td>
<td>aggr</td>
</tr>
</tbody>
</table>

### TABLE IX: MERGED POLICIES

<table>
<thead>
<tr>
<th>Table name</th>
<th>User role</th>
<th>Start</th>
<th>End</th>
<th>Policy</th>
<th>Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>items</td>
<td>2</td>
<td>20th Aug</td>
<td>14th Dec</td>
<td>X+Y</td>
<td>simp+aggr</td>
</tr>
<tr>
<td>items</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>Z</td>
<td>simp</td>
</tr>
<tr>
<td>orders</td>
<td>1</td>
<td>10th Aug</td>
<td>14th Dec</td>
<td>T+P</td>
<td>simp</td>
</tr>
<tr>
<td>cards</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>Q</td>
<td>aggr</td>
</tr>
</tbody>
</table>
Table VIII and Table IX show respectively a set of policies generated with the Policy Generator and the same set after being modified by the Query Wizard module. The latter is stored in memory and accessed at runtime each time a query coming from the web application is intercepted by the Query Interceptor module.

4.4.3 Tables replacement

When a query coming from the web application is intercepted, the Tables Replacement module parses it and checks for each table found if there is a policy than can applied to the current table and user. The merged policies are stored in ascending order of user role, and the first policy with user role greater or equal to the session one is applied. Applying a policy literally means replacing the protected table with it every time it occurs in the FROM clause of the original query sent by the web application. Following the examples of Table IX, the original query of Listing 4.4 is transformed into the one shown in Listing 4.5 when the web application is used by a user with user_role = 2:

Listing 4.4: Original query

```sql
1 SELECT *
2 FROM items i
3 WHERE i.category_id <> 4
```
Listing 4.5: Transformed query

```sql
1  SELECT *
2  FROM (X+Y) i
3  WHERE i.category_id <> 4
```
CHAPTER 5

EVALUATION

The correctness of the system has been manually tested over a set of 10 different jsp open source applications generated by the Codecharge tool [17] [18], which are vulnerable to SQL injections attacks. For each application I have found at least two vulnerabilities, injected SQL code and displayed sensitive information such as login usernames, passwords, addresses, phone numbers etc.

Then I ran all the applications again after having patched them with the filter that I have implemented and wrote the policies to protect them; in none of them I was able to retrieve sensitive data anymore. In this chapter I will present the impact that my system has on the patched applications from the point of view of the user experience. I will make an analysis on the additional time that the filter adds to some user interactions (e.g. search of a book through a web form), comment the cases in which it worsen the performance and show that there exist some cases in which this technique can help to impact positively on the response time.

Through a series of tests I have found out that the performance highly depends on the kind of query modification applied and used by the filter, and this factor doesn’t change significantly over the set of applications, so for a clear explanation I will present and comment only the results obtained with the Bookstore application [14], a portal that can be used to find and order books, but the analysis has to be intended as general and can be extended to any web application on which this kind of filter is applied.
5.1 **Test machine**

The performance tests have been carried out on a late 2011 Mac Book Pro 15”, with a 2.7 GHz Intel Core i7 processor and 8GB of 1600MHz DDR3 RAM. The web applications used for the tests run over an Apache Tomcat 7.0 web server and are connected to a MySQL 5.6 database server (both on the local machine) through the MySQL Java Connector 5.1.30 that has been extended and patched with the filter.

5.2 **Structure of the tests**

When a user sends a request to the database (e.g. by clicking the search button on the application web form), the time slots we can identify until the result is rendered on the browser are:

- **Connection setup**: time needed to setup the connection with the server (e.g. DNS Lookup, Proxy negotiation etc . . .)
- **Request time**: time needed to send the request to the server
- **Filter query modification**: time needed by the filter to modify the query (when necessary)
- **Database execution and fetch**: time needed by the database to calculate the result of the query
- **TTFB**: time needed to receive the first byte at client side
- **Content Download**: time needed by the client to download all the content of the answer
- **Page rendering**: time needed by the client to display the result
All these time slots are important factors for the user experience, anyway for the purpose of the tests I will only consider the ones that are affected by the introduction of the filter: the filter query modification and the database execution and fetch time. All the remaining time slots depend only on the network delays between the client (the browser) and the server (the web application), and are not taken into account. So the results shown in the next section will be the sum of this two time slots and they are calculated at server side like shown in Listing 5.1.

Listing 5.1: Calculation of the filter performance impact

```java
java.sql.ResultSet rs = null;
long startTime = System.nanoTime();
// Sends sqlQuery to the database and store the result in 'rs'
rs = openrs(statement, sqlQuery);
long totalTime = System.nanoTime() - startTime;
```

Each query has been tested 500 times with a Python script [14] that sends the same web form to the web app, collects the 500 'totalTime' and computes mean and standard deviation. This is the actual result that will be shown and commented in the next section for each tested query. Moreover, each test has been repeated over three different sizes database populated with the Spawner Data Generator tool [19] to show how the filter impacts on small, medium and large size databases. Since the tests are mainly run over a single table (the one that will be protected with the policies), the database size will indicate only how many tuples are present in the target table.
5.3 Performance impacts

The tests have been carried out over two macro-categories of queries:

- Simple queries: those queries that select attributes of tuples as they appear in the database without performing any aggregation

- Aggregate queries: those queries that select aggregates (e.g. \textit{avg, count, min, max, sum}) of one or more attributes

This classification categorizes the original queries sent from the web application, not those modified by the filter; a simple query might select from a table that the filter will replace with an inline view that also calculates an aggregate, and an aggregate query might contain a table that will be replaced with an inline view that also selects simple tuples.

Each macro-category has been tested with different security policies whose goal is to make the filter generate a modified query that try to cover the worst cases in terms of execution time and fetch time. The tests are not guaranteed to cover the absolute worst cases since it would require a quite complex reverse engineering on the query modification starting from the combination of the most costly operations in a relational database. Anyway the definition of typical policies that generate complex queries that are visibly expensive to execute, is sufficient to show how the filter impacts on the application in a non-optimal case.

For each test I will report the original query sent with the web form and the policy that has been applied. The modified query can be derived by applying the concepts presented in Chapter 3 or using the Bookstore web application with the filter applied.\[14\]
5.3.1 Simple queries

Simple queries should also be split into two categories to cover the two extreme cases they can generate: selection of a high number of tuples (e.g. 90% of the total), and selection of a low number of tuples (e.g. 5% of the total). Given the uniform generation of data by the Spawner Data Generator tool [19], the percentage of tuples selected is nearly the same on the three different sizes datasets used for the tests in both cases; so if a query selects the N% of tuples from a table T with M tuples, the same query will approximately select N% of tuples also from any other instance with KM tuples of table T whose data is generated by the Spawner tool.

5.3.1.1 Test 1: selection of 90% of the tuples

The original query from which the modifications have been applied is reported in Listing 5.2.

<table>
<thead>
<tr>
<th>No.</th>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SELECT</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>i.author as i_author, i.category_id as i_category_id,</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>i.image_url as i_image_url, i.item_id as i_item_id,</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>i.name as i_name, i.price as i_price,</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>c.category_id as c_category_id, c.name as c_name</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>FROM items i, categories c</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>WHERE c.category_id=i.category_id</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>and (i.name like '%a%')</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>and (i.category &lt;&gt; 4)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>ORDER BY i.name asc</td>
<td></td>
</tr>
</tbody>
</table>

Listing 5.2: Test 1, original query
This query is generated by the web application when the user searches books whose titles contain the letter ‘a’ and they don’t belong to category 4 (about the 90% of the books are fetched). The table that we want to protect with the filter is the items table; the policies will limit the access to this table by making the filter replace every occurrence of this table in the original query with an inline view. Table X reports the results obtained with this test: the first row with a gray background represents the response time of the web application without the patch of the filter, while all the other rows show the response times when the filter is applied. Each row is relative to a different query modification due to the different security policies applied that can be found in Appendix A.1. All the results are in milliseconds.
### TABLE X: TEST 1 RESULTS IN MILLISECONDS

<table>
<thead>
<tr>
<th></th>
<th>10K</th>
<th>100K</th>
<th>1M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>StDev</td>
<td>Mean</td>
</tr>
<tr>
<td><strong>Original query</strong></td>
<td>22.3</td>
<td>1.3</td>
<td>816.8</td>
</tr>
<tr>
<td><strong>No modification</strong></td>
<td>22.8</td>
<td>1.4</td>
<td>819.5</td>
</tr>
<tr>
<td><strong>Simple Policy</strong></td>
<td>60.8</td>
<td>3.8</td>
<td>1348.3</td>
</tr>
<tr>
<td><strong>Simple+Count</strong></td>
<td>128.3</td>
<td>4.7</td>
<td>1660.8</td>
</tr>
<tr>
<td><strong>Simple+Min</strong></td>
<td>116.8</td>
<td>4.4</td>
<td>1498.3</td>
</tr>
<tr>
<td><strong>Simple+Max</strong></td>
<td>116.6</td>
<td>4.0</td>
<td>1501.4</td>
</tr>
<tr>
<td><strong>Simple+Sum</strong></td>
<td>117.8</td>
<td>4.1</td>
<td>1510.7</td>
</tr>
<tr>
<td><strong>Simple+Avg</strong></td>
<td>118.6</td>
<td>4.4</td>
<td>1513.4</td>
</tr>
</tbody>
</table>

A first observation that can be done on this result set is that in case the filter is applied but no query modification is done (because no policy has been specified on the target table), the overall impact is negligible; only a few milliseconds average increment in the case of a table with 1 million tuples.

Results obviously become worse when a simple modification is applied; the *items* table in this case is replaced by an inline view that contains only those books with `category_id = 4`, and the outer query will then select only the ones that contain an 'a' in the title. This computation
worsen the performance by 3 times in the case of a small dataset with ten thousand tuples, and by 1.5 times in the other two cases. There is also the case in which the specified policy makes the inline view compute an aggregate, and then append this result to the set of simple tuples that will compose the inline view; the performance impact gets more significant especially in the case of the smaller dataset where there is an increment greater than 5 times, while in the other two dataset it’s limited to less than 2 times. It’s interesting to notice how the solution adopted with the Simple+Count policy takes more time to compute with respect to the other aggregates; this behavior is due to the fact that the inline view selects as many null values as the number of books with category_id = 4, and the fetch phase takes more time than by simply selecting a single value calculated with the other aggregates.

5.3.1.2 Test 2: selection of 5% of the tuples

The original query from which the modifications have been applied is reported in Listing 5.3.
Listing 5.3: Test 2, original query

```
SELECT
  i.author as i_author, i.category_id as i_category_id,
  i.image_url as i_image_url, i.item_id as i_item_id,
  i.name as i_name, i.price as i_price,
  c.category_id as c_category_id, c.name as c_name
FROM items i, categories c
WHERE c.category_id = i.category_id
  and (i.name like '%a%')
  and (i.price = 200)
ORDER BY i.name asc
```

In this case the web application sends out a query that selects all the books with a title containing letter 'a' and whose price = 200 (about 5% of the books are fetched). The table that we want to protect is again the items table, and the filter will now replace it (if specified by the policy) with an inline view that contains a small number of tuples. Of course we could specify a policy that makes the inline view select a super-set of the tuples whose price = 200, but we will stick to what we have done previously and define policies that will select exactly those books with price = 200 and eventually some aggregated result. This books will then be filtered by the outer query to only those containing letter 'a' in the title.

Table XI reports the results obtained with this test: the first row with a gray background represents again the response time of the web application without the patch of the filter, while
all the other rows show the response times when the filter is applied. The policies used are available in Appendix A.2. All the results are again in milliseconds.

As in the first test, having the application patched with the filter that doesn’t modify the query, doesn’t visibly affect the user experience. The simple policy by itself instead increments the response time of about 1.5 times in each dataset, while the simple policies plus aggregates

<table>
<thead>
<tr>
<th>Number of tuples of the <em>items</em> table</th>
<th>10K</th>
<th>100K</th>
<th>1M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>StDev</td>
<td>Mean</td>
</tr>
<tr>
<td>Original query</td>
<td>4.9</td>
<td>1.1</td>
<td>45.0</td>
</tr>
<tr>
<td>No modification</td>
<td>5.4</td>
<td>1.2</td>
<td>49.3</td>
</tr>
<tr>
<td>Simple Policy</td>
<td>9.3</td>
<td>1.8</td>
<td>73.1</td>
</tr>
<tr>
<td>Simple+Count</td>
<td>95.5</td>
<td>4.5</td>
<td>917.9</td>
</tr>
<tr>
<td>Simple+Min</td>
<td>14.0</td>
<td>3.2</td>
<td>105.8</td>
</tr>
<tr>
<td>Simple+Max</td>
<td>13.8</td>
<td>3.2</td>
<td>106.4</td>
</tr>
<tr>
<td>Simple+Sum</td>
<td>13.9</td>
<td>3.3</td>
<td>105.5</td>
</tr>
<tr>
<td>Simple+Avg</td>
<td>16.7</td>
<td>3.9</td>
<td>127.6</td>
</tr>
</tbody>
</table>
worsen the performance of 3 times. In general all the behaviors follow the guideline defined
with the previous test, with the exception of the simple policy plus the count aggregate; in this
case it becomes immediately evident that the result is much worse than the others. The reason
of this huge difference is very simple and it’s due to the same explanation given for the little
difference in the previous test: to allow a potential count query (sent from the web application
) to return a correct result, the inline view created by the filter, other than selecting the 5-7%
of tuples with price = 200, also appends the remaining 93-95% of null values. While in the
first dataset the number of null values appended was very low, in this case they compose the
vast majority of the result and cause this significant drop of performance.

5.3.2 Aggregate queries

Aggregate queries are those queries that select aggregates of values and the result is typically
a number, so it doesn’t make sense to distinguish them with respect to the percentage of tuples
selected since it will be 1 in any case. What makes the difference on the performance impact
is the number of tuples selected by the inline view created by the filter, but we don’t need any
specific test to say that increasing them will take more time to compute the query result. A
more significant test to evaluate the filter query modification in this specific case is comparing
all the aggregates with each other, the policies used to test the queries will differ only on the
kind of aggregate selected.

5.3.2.1 Test 3: avg, sum, min, max

The results obtained by comparing these four aggregates with each other are basically the
same, computing avg, sum, min and max leads to very similar results. They don’t deserve
any specific comment since the time taken to compute each of them is so similar that their differences are mainly due to errors in the measure (standard deviation is not negligible) and their SQL execution plan. I’ll summarize them by commenting only the results obtained with the `avg` and comparing them with the results of the `count` in the next test.

The query sent from the web application is reported in Listing 5.4 and the applied policies can be found in Appendix A.3.

Listing 5.4: Test 3, original query

```sql
SELECT avg(price) as price
FROM items as i
WHERE category_id = 4
```

This query simply selects the average of the prices of the books with `category_id = 4` in the `items` table, which is again the table that we want to protect. The policy this time will replace this table with an inline view that can be very different from the previous tests: if we specify a policy in which we say that a user can only see the average value and cannot retrieve any other information, then the inline view will only contain one row and the results become interesting as reported in Table XII.
The third row shows the result of the situation just mentioned and it’s evident that the performance impact is null. A policy that makes the inline view select only one aggregate can be exploited in many situations in which a user with specific permissions is not allowed to see any information about a table but some aggregate values, and in such a case the impact of the filter will be completely negligible. This can be explained by the fact that the outer query will only compute the average of a single value (the value itself), that turns out to be very fast. Of course if the policy also allows to retrieve some information about individual rows (last row of Table XII), then we fall back into the examples already commented in the previous tests. The results of such a case are very different from the previous examples because both the policy and the original query are different, anyway it will deserve a comment when compared to the same situation with the count aggregate explained in the following test.
5.3.2.2 Test 4: count

The original query generated by the be application is the same of the previous one, but selects the number of the prices of the books with category_id = 4 in the items table instead of the average. The query sent is reported in Listing 5.5 and the applied policies can be found in Appendix A.4.

Listing 5.5: Test 4, original query

```
1 SELECT count(price) as price
2 FROM items as i
3 WHERE category_id = 4
```

The results obtained are shown in Table XIII.

<table>
<thead>
<tr>
<th>Number of tuples of the items table</th>
<th>10K Mean</th>
<th>10K StDev</th>
<th>100K Mean</th>
<th>100K StDev</th>
<th>1M Mean</th>
<th>1M StDev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original query</td>
<td>3.4</td>
<td>0.5</td>
<td>29.6</td>
<td>0.8</td>
<td>407.8</td>
<td>4.2</td>
</tr>
<tr>
<td>No modification</td>
<td>3.8</td>
<td>0.5</td>
<td>30.5</td>
<td>1.1</td>
<td>417.4</td>
<td>4.7</td>
</tr>
<tr>
<td>Only Count</td>
<td>5.9</td>
<td>0.5</td>
<td>46.1</td>
<td>1.4</td>
<td>565.3</td>
<td>5.1</td>
</tr>
<tr>
<td>Simple+Count</td>
<td>16.4</td>
<td>1.5</td>
<td>390.5</td>
<td>3.8</td>
<td>3049.4</td>
<td>21.6</td>
</tr>
</tbody>
</table>
From the third row of the table it’s evident that the count aggregate suffers again the fetch time due to the high number of null values selected, anyway in this situation the performance impact is limited since the outer query will use them only to perform the count, instead of fetching them again as in the first test (Listing 5.2). This fact is more evident when comparing the last row with the Simple+Avg result of the third test (Table XII): even if the count operation itself is slightly faster than the average (it can be seen by comparing the original queries results of the two tests), when the filters adds the inline view the former has again more influence on the performance.

5.4 Performance improvements

As mentioned at the beginning of this chapter, there are some situations in which the filter may improve the user experience. So far we have showed the percentage of performance drop when running a single modified query for many times, and then averaging the results obtained. We were able to do so because of the way in which MySQL handles views: as stated by the MySQL Server documentation, ”Views are stored queries that when invoked produce a result set. A view acts as a virtual table.”, so they are simply a handy way to name a query and access its result set from another query. The result set in not stored like a table but it’s generated every time the view is invoked, which makes them very easy to use since they don’t need any maintenance, but on the other hand they are very inefficient in terms of computational time especially when the result set they produce never changes or changes rarely. The exact same concept is valid for the inline views that our filter puts in place of the tables protected by the policies, with the only difference that the views are not invoked by their name but they are
entirely put inside the `FROM` clause of the original query; they can be seen as sub-queries in the `FROM` clause.

Since the inline views produced by the filter are generally a subset of the tuples of the tables that they are protecting, having a reduced stored result set that can be accessed without being recomputed each time, could lead to a performance improvement. A typical situation in which such a scenario can become very useful, is for example the search of N different books from their title: let us consider again the `Bookstore` web application where the books are stored into the `items` table and a registered user is only allowed to search for books of `category_id = 1`, so we protect this table by making only the books of `category_id = 1` available for the selection.

Listing 5.6: Original query

```sql
1 SELECT item_id, title, author
2 FROM items
3 WHERE category_id = 1 AND title like '%title1%'
```

Listing 5.7: Modified query

```sql
1 SELECT item_id, title, author
2 FROM ( SELECT * FROM items WHERE category_id = 1 ) as my_cached_view
3 WHERE category_id = 1 AND title like '%title1%'
```

The inline view obviously reduces the set of tuples on which the original query applies the title selection; the ideal situation that we want to achieve is the one in which by searching other titles, the books will be selected from a cached view.
Listing 5.8: Query on the cached view

```
1 SELECT item_id, title, author
2 FROM my_cached_view
3 WHERE category_id = 1 AND title like '%title2%'
```

A cached view is more commonly called *materialized view* because it acts as a normal view but the result is materialized as a table. In a certain way it is very similar to create a table with the `'CREATE TABLE ... AS'` SQL construct, with the difference that the latter cannot be refreshed and kept up to date if the tables used to construct the result change. Unfortunately MySQL 5.6 doesn’t natively support this functionality, the only caching that it performs involves the entire query, so by searching a different book title the entire query is considered to be different (even if only one character is different, the query is considered to be different) leading to a cache miss. MariaDB 10.1 [20], a fork of MySQL, enhances many of its optimization steps and natively supports sub-query caching only for sub-queries in the WHERE clause [21]. It can be patched with Flexviews [22], an external open source project that implements the behavior of materialized views and automatically maintain them when the source tables change, but doesn’t recognize the inline views automatically; instead it provides a SQL API to manually create the views and force their caching. Using such a tool would require a more complex system at the MySQL interceptor level, that sends at least four API calls to the database and catches the replies if the table involved in the modification hasn’t been cached.
before; this tool would increment too much the complexity of the system and would change the purpose of the filter itself of just modifying the query or letting it through.

5.4.1 Oracle database

Oracle RDBMS natively supports the functionality of materializing views [23]. His performances are very different from MySQL or MariaDB and they cannot be compared, anyway we are interested to understand what kind of performance improvement we could obtain from an optimization of this kind. The tests presented from now on have been carried out with the same configuration presented in Chapter 5.1 but using Oracle 12c Release 1 instead of MySQL 5.6 as database server. Another minor difference is the entry point of the query modification filter: the MySQL Java connector code is open source and can be easily extended, while the Oracle Java connector is a binary package that cannot be easily modifiable. For this reason I moved the entry point of the query modification at the web application level just for the purposes of this tests, so that the Oracle interceptor will let the final modified query pass through; this solution goes against the main goal of this work of not editing the web application code, but it is the best option if there is no easy way to modify the Oracle interceptor and it will be only used to test this particular case of result caching.

To understand when it is more convenient to materialize the inline view, let’s consider the two main time slots that give the query execution total time:

- Duration time (DT): time needed to compute the result, it highly depends on how the query is optimized
• Fetch time (FT): time needed to fetch the result, it highly depends on the size of the result and the network latency. In the case of our tests, we can consider the network component negligible since everything runs on the localhost.

The sum of these two time slots will give us the total time measured at the application level, so we need to optimize either one or the other to obtain a better result. Caching a result would lead to nullify the duration time since the result has already been computed, and reduce the fetch time. Let us consider four macro-categories of queries with respect to their total execution time:

• Low DT, Low FT: this kind of query doesn’t significantly enhance the performance when the result is cached, the total execution time is already low.

• Low DT, High FT: the bottleneck of this kind of query is the time taken to fetch the result. Typically this occurs with big result sets; having the result cached would surly significantly improve the fetch time, but requires a big cache allocation which is not always available and might fill space that could be used by several other queries of the following kind.

• High DT, Low FT: this kind of queries are the favorite to exploit the cache; they take a lot of time to execute but they typically generate a small result set. A high number of this results can be materialized and thus improve the overall user experience.

• High DT, High FT: it would be extremely helpful to store the results of this kind of queries, but still it requires a lot of cache due to the result set size.
Oracle makes a tag available to let the database administrator decide which inline view to cache. By setting this tag after the SELECT clause, the result generated by that selection will be cached and automatically accessed when the same query is invoked a second time. The maintenance phase is totally handled by the database making it very handy to use.  

Listing 5.9 is a simple example of how to cache an inline view.

```
SELECT *
FROM (  
    SELECT /* RESULT_CACHE */ item_id, title, author  
    FROM items  
    WHERE category_id = 1  
) my_cached_view  
WHERE category_id = 1 AND title like '%title1%'
```

When another query with the same inline view is sent to the database, the cached result of the view will be fetched if the `items` table hasn’t been modified (cache hit), even if the outer query is different (e.g. a different title is searched).

5.4.1.1 Test: inline view with high DT and low FT

To have an idea of what kind of improvement we can obtain with this technique, I will show the response times of the filter obtained using the Oracle database with and without caching the result of the inline view. I’ll present the results of an inline view that I have classified as having a high duration time (DT) and a low fetch time (FT). The original query generated by the `Bookstore` web application is the one in Listing 5.10.
Listing 5.10: Original query

```
1  SELECT i.category_id, avg(i.price) as price
2  FROM items
3  WHERE price >= 5
4  GROUP BY i.category_id
5  ORDER BY i.category_id
```

The `items` table is protected with the policy listed in Appendix A.5. The tests have been performed on the usual three different size databases with the results reported in Table XIV:

**TABLE XIV: ORACLE TEST RESULTS IN MILLISECONDS**

<table>
<thead>
<tr>
<th></th>
<th>Number of tuples of the <code>items</code> table</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10K</td>
</tr>
<tr>
<td></td>
<td>Mean</td>
</tr>
<tr>
<td>Original query</td>
<td>6.4</td>
</tr>
<tr>
<td>Filter - no cache</td>
<td>15.3</td>
</tr>
<tr>
<td>Filter - cache</td>
<td>2.5</td>
</tr>
</tbody>
</table>

This table is slightly different from the previous ones: the first row always represent the original query run on the `items` table, but the other two rows show the results obtained by
applying the exact same policy respectively without caching and caching the inline view, which has a very small result set that only contains five rows and takes a significant amount of time to be calculated. It’s evident how the cache can enhance the user experience: with a 1 million rows table, the filter response to the same query can be significantly faster going from the 390ms of the original to only 2.5ms of the modified one (approximately 150 times less).
CHAPTER 6

CONCLUSIONS

In this thesis we presented a solution to protect existing web applications from SQL injection attacks without modifying neither the source code nor the database schema. This solution allows the web application administrator to easily specify and maintain security policies to protect sensitive data from malicious users without worrying of the system implementation. We have shown how to achieve the goal by adding an intermediate filter in between the web application server and the database server that modifies the queries before they get executed, such that every occurrence of a table that has to be protected is replaced by a view that only contains the data that can be accessed.

The main novelty of this approach is to allow to make available data aggregates without leaking information of single tuples. The most significant advantages are the simplicity in which the policies can be expressed (especially when there is the need to specify many of them for a single table), the low maintenance required to update them or introduce new ones, the possibility to automatically activate and deactivate them by specifying their time range of validity, and the future possibility of enhancing performances in particular cases when also public license database such as MySQL or MariaDB will implement inline view caching optimization like Oracle. The costs that have to be paid to augment the security level of the applications with this solution, are the negative performance impact that complex views may introduce and the complexity of the view creation in case multiple aggregates of different columns need to be
available. We saw that the response times can become very high when the policy that protects a table becomes complex, and that is the cost to pay for a fine-grained database security.

The overall result of this thesis is the demonstration that this kind of solution is ideal for those web applications that have been coded without concerning about security and now need to be fixed. While this approach needs improvement in handling more complex policies than the ones presented in Chapters 3.2.3-3.2.4, it can already be applied to real web applications with a level of complexity similar to the ones used for testing.
CHAPTER 7

FUTURE WORK

This thesis can be extended in many directions to improve different aspects of the system. First of all the filter can be extended with the well studied techniques used in statistical databases to further ensure protection of sensitive data \cite{24,25,26} like data perturbation, query restriction or output perturbation; this improvement would prevent statistical disclosure that hasn’t been considered for the purpose of this work. In Chapter 3 we have shown that there is no unique solution to the problem of making aggregates available without disclosing sensitive data, but each aggregate needs its own ad-hoc solution and their combination also needs different ones; the filter could be improved by adding a rule-based system that quickly identifies the kind of policy that has been specified and applies the correct query modification. This would make the maintenance of the filter easier. Moreover, replacing the same table with different views, depending on the query sent by the web application, can help to find easier solutions for what we have called many-one and many-many policies with different where clauses, and the merging phase can be extended to cover more complex cases. A further enhancement is the integration of the control that Oracle VPD \cite{13} applies to DELETE, INSERT and UPDATE statements since they are never used with aggregates; it only performs row level access control and could be improved for the UPDATE statement, the only one that may need a cell-level access control. Moreover, the current implementation is an extension of the MySQL Java Connector which can be only used to connect Java based applications to the underlying database,
extending it to other common web languages such as *php* would cover a wider range of web applications that can exploit this solution.
APPENDIX

POLICIES USED IN THE TESTS

A.1 Test 1

This appendix lists the policies used to test the filter performance impact in Test 1.

Listing A.1: No Modification

```xml
<?xml version="1.0" encoding="UTF-8"?>
<table_list schema="bookstore"> </table_list>
```

Listing A.2: Simple Policy

```xml
<?xml version="1.0" encoding="UTF-8"?>
<table_list schema="bookstore">
  <table name="items"> <columns> item_id, category_id, name, author, price, product_url, image_url, notes, is_recommended, rating, rating_count </columns>
  <relation_view kind="simple" name="Top secret books protected">
    <user_role>1</user_role>
    <begin_date>1970-01-01 00:00:00</begin_date>
    <expiration>2100-01-01 00:00:00</expiration>
    <policy>SELECT * FROM items WHERE category_id &lt;&gt; 4 </policy>
  </relation_view>
</table_list>
```
Listing A.3: Simple+Count

```xml
<?xml version="1.0" encoding="UTF-8"?>
<table_list schema="bookstore">
  <table name="items">
    <columns>item_id, category_id, name, author, price, product_url, image_url, notes, is_recommended, rating, rating_count</columns>
    <relation_view kind="simple" name="Category 4 books cannot be searched">
      <user_role>1</user_role>
      <begin_date>1970-01-01 00:00:00</begin_date>
      <expiration>2100-01-01 00:00:00</expiration>
      <policy>SELECT * FROM items WHERE category_id &lt;&gt; 4 </policy>
    </relation_view>
    <relation_view kind="aggregate" name="Can only see aggregate of Top secret books">
      <user_role>1</user_role>
      <begin_date>1970-01-01 00:00:00</begin_date>
      <expiration>2100-01-01 00:00:00</expiration>
      <policy>SELECT count(price) FROM items WHERE category_id = 4 </policy>
    </relation_view>
  </table>
</table_list>
```

All the other Simple+Aggregate tests use the same policy of Listing A.3 where count at line 15 is replaced by the other aggregates (avg, min, max, sum).
APPENDIX (continued)

A.2 Test 2

This appendix lists the policies used to test the filter performance impact in Test 2.

Listing A.4: No Modification

```xml
<xml version="1.0" encoding="UTF-8"?>
<table_list schema="bookstore">
</table_list>
```

Listing A.5: Simple Policy

```xml
<xml version="1.0" encoding="UTF-8"?>
<table_list schema="bookstore">
  <table name="items">
    <columns>item_id,category_id,name,author,price,product_url,image_url,notes,is_recommended,rating,rating_count</columns>
    <relation_view kind="simple" name="Books that cost 200\$ are protected">
      <user_role>1</user_role>
      <begin_date>1970−01−01 00:00:00</begin_date>
      <expiration>2100−01−01 00:00:00</expiration>
      <policy>SELECT * FROM items WHERE price = 200</policy>
    </relation_view>
  </table>
</table_list>
```
All the other Simple + Aggregate tests use the same policy of Listing A.6 where `count` at line 15 is replaced by the other aggregates (\(avg\), \(min\), \(max\), \(sum\)).
APPENDIX (continued)

A.3 Test 3

This appendix lists the policies used to test the filter performance impact in Test 3.

Listing A.7: No Modification

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<table_list schema="bookstore">
</table_list>
```

Listing A.8: Only Avg

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<table_list schema="bookstore">
  <table name="items">
    <columns>item_id, category_id, name, author, price, product_url, image_url, notes, is_recommended, rating, rating_count</columns>
    <relation_view kind="aggregate" name="Avg price of all books with category = 4">1
      <user_role>1</user_role>
      <begin_date>1970-01-01 00:00:00</begin_date>
      <expiration>2100-01-01 00:00:00</expiration>
      <policy>SELECT AVG(price) FROM items WHERE category_id = 4</policy>
    </relation_view>
  </table>
</table_list>
```
Listing A.9: Simple+Avg

```xml
<?xml version="1.0" encoding="UTF-8"?>
<table_list schema="bookstore">
  <table name="items">
    <columns>item_id, category_id, name, author, price, product_url, image_url, notes, is_recommended, rating, rating_count</columns>
    <relation_view kind="simple" name="Books that contain letter a">
      <user_role>1</user_role>
      <begin_date>1970-01-01 00:00:00</begin_date>
      <expiration>2100-01-01 00:00:00</expiration>
      <policy>SELECT item_id, name, price FROM items WHERE name like '%a%'</policy>
    </relation_view>
    <relation_view kind="aggregate" name="Avg price of all books with category = 4">
      <user_role>1</user_role>
      <begin_date>1970-01-01 00:00:00</begin_date>
      <expiration>2100-01-01 00:00:00</expiration>
      <policy>SELECT AVG(price) FROM items WHERE category_id = 4</policy>
    </relation_view>
  </table>
</table_list>
```

A.4 Test 4

These policies are exactly the same of the ones used in Test3, where the \textit{avg} aggregate is replace with \textit{count}. 

A.5 Oracle test

This section lists the policy used to test the filter performance impact when the Oracle database has been used.

Listing A.10: Oracle test

```xml
<?xml version="1.0" ?>
<table_list schema="bookstore10K">
<table name="items">
  <columns>item_id, category_id, name, author, price, product_url, image_url, notes,←
            is_recommended, rating, rating_count</columns>
  <relation_view kind="simple" name="Only show info about authors whose name ←
                    is Guy">1
    <user_role>1</user_role>
    <begin_date>1970-01-01 00:00:00</begin_date>
    <expiration>2100-01-01 00:00:00</expiration>
    <policy>SELECT name, author, price, item_id, category_id FROM items WHERE ←
            author like '%Guy%' </policy>
  </relation_view>
  <relation_view kind="aggregate" name="Show the average price for each ←
                              category of books">1
    <user_role>1</user_role>
    <begin_date>1970-01-01 00:00:00</begin_date>
    <expiration>2100-01-01 00:00:00</expiration>
    <policy>SELECT AVG(category_id) FROM items WHERE price &gt;= 5 GROUP BY ←
           category_id </policy>
  </relation_view>
</table_list>
```
CITED LITERATURE


CITED LITERATURE (continued)


# VITA

## NAME
Paolo Bruzzo

## EDUCATION
- Master of Science in "Computer Science", University of Illinois at Chicago, December 2015, USA
- Specialization Degree in "Computer Engineering", December 2015, Politecnico di Milano, Italy
- Bachelor’s Degree in "Computer Engineering" - July 2013, Italy

## LANGUAGE SKILLS
<table>
<thead>
<tr>
<th>Language</th>
<th>Proficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Italian</td>
<td>Native speaker</td>
</tr>
<tr>
<td>English</td>
<td>Full working proficiency</td>
</tr>
</tbody>
</table>

- 2012 - IELTS examination (7.0/9)
- A.Y. 2014/15 One Year of study abroad in Chicago, Illinois
- A.Y. 2013/14 Lessons and exams attended exclusively in English
- A.Y. 2009/10 Three Months of study abroad in Queenstown, NZ

## SCHOLARSHIPS
- Spring 2015 Research Assistantship (RA) position (20 hours/week) with full tuition waiver plus monthly stipend

## TECHNICAL SKILLS
<table>
<thead>
<tr>
<th>Level</th>
<th>Skills</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>PHP, XQuery, CSS</td>
</tr>
<tr>
<td>Average</td>
<td>Bash, Javascript, HTML, C++</td>
</tr>
<tr>
<td>Advanced</td>
<td>C, Java, Python, SQL</td>
</tr>
</tbody>
</table>

## WORK EXPERIENCE AND PROJECTS
- Dec ’14 - May ’15 Research Assistant at UIC
- Aug ’12 - Sep ’12 Technical Assistant at Meyer S.R.L
- Other Experiences:
C-minus Compiler: Compiler front-end for the C-language.

Networking tools: Client and Web Server, DNS Client, Reliable link over UDP. [C, Python]

CHAP: Implementation of the Challenge Authentication Protocol logic. [Python]

Chicago Safe Travel: Application that helps users make decisions about their travel plans throughout Chicago. [Javascript, HTML, D3, CSS]

Heterogeneous Droid Cluster: Performance analysis of a cluster of Android mobiles over WiFi. [C, MPI, HPL]

MLSP Challenge: Diagnose schizophrenia mining data coming from MRI scans. [Python, Weka, Java, R]

Travel Dream: Dynamic web application developed to support a travel agency to sell travels. [J2EE, Glassfish, MySQL]