Designing a SQL Query Rewriter to Enforce Database Row Level Security

by

Xiao Meng Zhang

B.S., Massachusetts Institute of Technology (2014)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2016

Certified by . . . .
Samuel R. Madden
Professor
Thesis Supervisor

Accepted by . . . .
Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Designing a SQL Query Rewriter to Enforce Database Row Level Security

by

Xiao Meng Zhang

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

This thesis presents the design and implementation of *Row Level Security*, a fine-grained access control mechanism built on top of a database-agnostic data sharing platform called DataHub. Existing access control mechanisms for database systems are typically coarse-grained, in the sense that users are either given access to an entire database table or nothing at all. This is problematic with the rise in popularity of data sharing, where users want to share subsets of data in a table with others, rather than the entire table. *Row Level Security* addresses this problem by allowing users to create security policies that define subsets of data others are able to access, and enforces security policies through a query rewrite mechanism. This work presents *Row Level Security*, as well as an evaluation of its performance overhead costs and ease of use.

Thesis Supervisor: Samuel R. Madden
Title: Professor

Acknowledgments

I am incredibly lucky to have met so many wonderful people during my past six years at MIT who have helped me get to where I am today. Coming to this school has been my longtime childhood dream and studying here has been as magical as I have always imagined it to be.

First and foremost, I would like to thank my research advisor, Sam Madden, for giving me the opportunity to work on this interesting project. I am grateful for his invaluable support and generous help, without which this project would not have been completed.

I would like to thank my mentors Anant Bhardwaj and Eugene Wu for their support, guidance, and inspiration over the course of this project.

I would like to thank the MIT Living Labs Group (especially Albert Carter and Justin Anderson) for their technical help throughout the development of this project.

I would like to thank my friends Qian, Jesika, Carrie, Pedro, Brando, Ty, Rui, Michelle, Michael, Deborah, Sarah, and Dev for being the best friends one could possibly ask for. Thank you for accompanying me throughout my MIT journey. I will cherish the incredible memories that we have built here for the rest of my life.

Finally, I would like to thank my parents for showering me with love and support throughout my entire life. Thank you for believing in me and encouraging me even when I did not believe in myself. Without you guys, I would not be the person that I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The past decade has witnessed tremendous growth in the amount of data produced and collected in a wide range of business, industrial, personal, medical, and scientific applications. Every day, more than 2.5 quintillion bytes of data are generated and collected – from posts to social media sites, to information logged by climate sensors, to transactions recorded by both small businesses and large corporations [1].

With this explosive growth of data comes a drastic rise in the amount of data that is shared both across and within organizations, especially as organizations become increasingly reliant on the use of data for day-to-day operations. As a result, it is of fundamental importance to ensure that database systems provide secure access control mechanisms to protect the data that is stored and shared amongst different users. The unauthorized disclosure, theft, or alteration of a user or organization's information could lead to severe negative consequences for all parties involved in the data compromise.

The traditional method that database systems use to securely share data is through the definition and enforcement of access control policies [11]. Unfortunately, these security policies are only optimal for coarse-grained authorizations, where users are given permission to access a database table in its entirety, or none at all.

In this thesis, we present the design of a fine-grained access control mechanism for database systems, *Row Level Security*, that allows users to be easily granted access to subsets of data within a database table, rather than the entire table.

1.1 Motivation

As mentioned above, the proliferation of data has spawned a drastic increase in data sharing - both across and within organizations, and amongst different users. This can be seen by the development and rise in popularity of data sharing applications such as DataHub [6], Dataverse [3], and ORCHESTRA [10]. With data sharing comes a pressing need for database systems to offer secure access control mechanisms that help protect the integrity of the shared data and facilitate the process of sharing data.

Traditionally, data sharing is achieved through the enforcement of access control policies [11]. The security policies grant users certain types of access (READ or WRITE) to specific tables located within a database. However, this type of data sharing is highly coarse-grained, as users either have access to the entire table, or nothing at all. Other types of access control for data sharing involve the creation of security labels and rules for different table records and users [12]. Users are given access to records within the table based on matching labels and rules. Although this approach offers more fine-grained access, it comes with much greater complexity and space overhead, as labels must be defined for all records within the table.

Unfortunately, all the data sharing access control mechanisms presented above are inefficient (or impossible) for use cases involving multiple users sharing one table but given access to different records. This type of use case is very common in data sharing systems. Examples of such are as follows:

1. Multiple universities are collaborating on a research project and want to share the data they have collected with each other. Each dataset contains records that are marked as either public or private. The universities want to share all of their datasets with each other, but only allow other universities to see records in their dataset that are marked as public. However, they want to allow researchers within their university to see all records, both public and private.

2. A sports application initiates a fitness challenge asking users to exercise as much as they can in a week long period. All fitness data logged by users are stored within one shared database table managed by the app. The app wants users to

be able to read all the records entered by participants into the table. However, they want users to only be able to modify their own personal log records.

3. A sales firm wants to assess the effectiveness and productivity of its employees. To do so, it asks all of its employees to log records of prices and the number of items they have sold each day in a shared table, upon which aggregate computations are performed. Employees are only able to see and modify the personal records they've logged. However, the aggregate computations are shown to everyone.

In order to support the use cases described above, we need a fine-grained access control mechanism that allows table owners to grant different users access to different records within a table. This is the problem that this thesis aims to solve with the design of *Row Level Security*.

1.2 Thesis Contribution

This thesis contributes a design and implementation of *Row Level Security*, a fine-grained access control mechanism built on top of a database-agnotistic data sharing platform called DataHub. With *Row Level Security*, table owners are able to share their tables with others and grant different users access to any subset of records within the shared table. Our design is different from previous access control designs in that it allows owners to grant or restrict table access on a per record (or group of records) basis.

Specifically, we present a design and implementation of *Row Level Security* that consists of three main components:

1. A tool for table owners to define security policies that specify which record(s) a user is able to access.

2. A query rewriter that rewrites queries executed by users to conform to security policies defined on the table.

3. A row level security manager that sits between the user layer and the database layer, and applies the query rewriter to queries executed by users.

In addition, we provide an experimental evaluation of the performance of *Row Level Security* and analyze the overhead costs of utilizing such a mechanism.

1.3 Organization

The remainder of the thesis is organized as follows. Chapter 2 presents the background and related work. Chapter 3 discusses the overall system design. Chapter 4 talks about the implementation and integration of *Row Level Security* with DataHub. Chapter 5 presents the performance evaluation of utilizing *Row Level Security.* Chapters 6 and 7 discusses future work and concludes.

Chapter 2

Background

In this chapter, we present a literature review of previous work related to database access control mechanisms used to help manage and facilitate data sharing between users. We will discuss the limitations of these controls, and examine how row level security addresses these limitations. Finally, we will provide background on DataHub, the platform upon which *Row Level Security* is implemented, and highlight the benefits of using such a platform.

2.1 Previous Work

Access control mechanisms in relational database management systems are a well studied area of research. They have been built into relational systems ever since the first products emerged, and have continued to evolve over the years as new systems and standards are developed.

The two main access control mechanisms at the database management system level are Discretionary Access Control (DAC) and Mandatory Access Control (MAC). Building on top of these two mechanisms is Role-Based Access Control (RBAC). Recently, new database management systems have released a Row Level Access Control similar to the *Row Level Security* design this thesis presents. We will discuss all the access control mechanisms mentioned above in this section and compare them with *Row Level Security*.

2.1.1 Discretionary Access Control

Discretionary Access Control (DAC) is a type of access control that restricts access to database tables based on the identity of the users and the security policy rules that define who can (or cannot) execute which actions on which resources [13]. Under this model, all entities are classified into one of three categories: (1) *resources* (such as tables, views, and stored procedures) that need to be protected, (2) *users* that execute activities and request access to resources, and (3) *privileges* (such as READ, WRITE, UPDATE) that can be executed on resources, and must be controlled.

When a resource is first created, the creator is declared the owner of the resource and has unrestricted access to it. The owner can then grant and revoke access on the resource to other users by defining policy rules of the form:

GRANT privileges [ON resource] TO users [WITH GRANT OPTION]

Once a user has been granted access to a resource, they are able to access the resource in its entirety as long as the actions taken on the resource match the privileges granted. Furthermore, the user could grant permission on the resource to other users if they are given the grant option in their own policy rule.

Going back to the context of data sharing, a user could share data they have stored in a table by granting READ access on the table to another user. The user being granted the permission would then be able to read all records in the table. Note that under this access control model, it is impossible for users to directly share subsets of data in a table with others. All types of data sharing is ALL-OR-NOTHING; a user is either granted privilege to an entire table or nothing at all. Users could work around this problem indirectly by creating views of subsets of the data they want to share and grant access on the views to those they want to share the data with. However, this approach becomes fairly expensive and inefficient under scenarios where different users see different subsets of the shared data, as is the case with all the use cases presented in Section 1.2.

Row Level Security improves upon DAC by allowing users to share data in a table with others on a per-record basis, effectively addressing the data sharing problem described above. It does so through the use of fine-grained security policies, where table owners are able to define in the policies the records others are able to access. Thus, *Row Level Security* offers all the benefits that DAC provides at the cost of having more complicated security policies.

2.1.2 Mandatory Access Control

Mandatory Access Control (MAC) is a type of access control that restricts access to database items based on *security labels* associated with data items and users [9]. Under this model, a database administrator assigns security labels to both users and data items, and users are only able to access data that has a lower security level than that of their own security label. Whenever a user attempts to access data managed under MAC, the database system compares the security label of the user with that of the objects being accessed. If the user's security label clearance exceeds that of the object, access is granted. Otherwise, access is denied.

What makes the MAC model attractive is that it supports the assignment of security labels to data items at any level of granularity [14], whether it's at the record level, column level, or table level. Regardless of the granularity chosen, access control is automatically applied to all users accessing the data in the same manner. This labeling flexibility is convenient as it makes it possible to enforce both fine-grained and coarse-grained access control, depending on the needs of the particular use case.

Unfortunately, the main limitation of the MAC model is that it is extremely costly to implement and maintain [8]. First and foremost, MAC requires substantial planning before it can be effectively implemented. The database administrator needs to pre-define a hierarchy for ranking security labels and assign labels to both data items and users. Second, all access control rules and permissions are only able to be modified by the database administrator. It is not possible for individual users to change the access control of a resource, even if the user owns that resource. This limits the scalability of such a model. Third, even when successfully implemented, this system requires a high management cost due to the need to constantly update labels to accommodate new data, users, and changes.

In the context of data sharing, MAC allows data to be shared with others through the use of security labels. To share subsets of data in a table with different users, the database administrator would need to assign security labels to records in such a way that each subset of data has a unique identifier that users could be granted access to. Although this approach is expensive and tedious, especially for large datasets, it is possible to meet the data sharing use cases described earlier in Section 1.2 using MAC.

Row Level Security is a much cheaper alternative to address the data sharing and access control problem compared to MAC. It does not require the assignment of security labels to all the data items that need to be access controlled; it does so through the creation of a small set of security policies. *Row Level Security* is more scalable in that any data owner can define policies and grant access to other users; this power does not rest solely with the database administrator. Lastly, *Row Level Security* is also more maintainable because access control is automatically applied to new data and users entered into the system, and security policies can be easily updated and modified.

2.1.3 Role-Based Access Control

Role-Based Access Control (RBAC) is a type of access control that restricts access to database items based on the roles of the user and the privileges granted to those roles [7]. Under RBAC, users are assigned to specific *roles* depending on their responsibilities and job functions, and access *privileges* are granted to various roles. When a user attempts to execute a particular task, they are able to do so as long as their roles are granted the necessary privileges to perform the task. Users can easily be reassigned from one role another. Furthermore, permissions can be easily granted and revoked from roles as new applications and data are incorporated into the database system.

RBAC is widely used in most commercial database systems. Its popularity stems from the fact that it follows a real-world approach to structuring access control [8]. In the real world, organizations are composed of people with different titles and responsibilities. Following this model, it's convenient to assign users to roles based on their titles and grant roles privileges matching the responsibilities a role is expected to perform.

RBAC is often implemented in conjunction with DAC or MAC. It improves upon DAC and MAC in that it does not require all users to be explicitly granted every privilege they need to accomplish their tasks, reducing the number of security policies and labels necessary [14]. However, regardless of whether it is implemented with DAC or MAC, it suffers from the same data sharing problems that those two access control models face. The only improvement RBAC makes is in allowing users to better manage the security policies defined for users. RBAC doesn't change the underlying data sharing structure that DAC and MAC follow. Therefore, the data sharing limitations experienced by RBAC are the same as those discussed for DAC and MAC [8].

The current design of *Row Level Security* does not support granting privileges to roles that users are assigned to. Instead, *Row Level Security* follows an approach similar to DAC, where users can have security policies created defining the privileges that they have on particular datasets (or subsets of datasets). In this regard, RBAC does a better job of supporting more concise security policies. However, again, *Row Level Security* allows for fine-grained record level access control in the context of data sharing that is difficult to achieve with RBAC.

2.1.4 Row Level Access Control

In recent months, Microsoft [4] and PostgreSQL [5] have released a new type of access control called row level access control (RLAC) that is integrated with their database management systems. RLAC is very similar with the *Row Level Security* design presented by this thesis, and is geared to explicitly target the data sharing problem of granting different users access to different subsets of data within a database table.

Under the RLAC model [5], users are able to create security policies on tables that they own to grant access to other users. Each security policy consists of: (1) a *user* to whom the policy grants access, (2) a *SQL expression* that evaluates to TRUE for all records the user is able to access in the table, and (3) a *SQL command* (SELECT, INSERT, ALL, etc) or *role* that the security policy applies to. Users can create multiple security policies for a table and can easily grant, revoke, and modify these policies. Whenever a user issues a query against a shared table, the SQL expressions of all security policies defined on the table that match the SQL command issued by the user or the user's role will be appended as predicates to the query issued. This ensures that the query executed by the user only applies to records in the table to which the user is granted access.

Going back to the context of data sharing, it is very straightforward for a table owner to share data that they have with others using RLAC. To do so, he simply needs to create a security policy granting other user access to it (or to subsets of data within the table). In the use case where multiple users share different subsets of a data table, a security policy can be created for each user defining which subset of records the user should be able to access (using the *SQL expression*). This access control mechanism resolves all the data sharing challenges experienced by MAC, DAC, and RBAC, and is both scalable and easy to maintain.

As mentioned earlier, RLAC is very similar to *Row Level Security*. Its development by Microsoft and PostgreSQL was done in parallel with the research of this thesis. The main idea and insight of using security policies to control data access and appending policy predicates to user executed queries is identical in both designs. The only difference between the two is that in *Row Level Security*, security policies can only be defined on *SQL commands* and not *roles*. In this thesis, we present one way of designing and implementing *Row Level Security* and analyze the performance of using such a design.

2.2 DataHub

In our design and implementation of *Row Level Security*, we decided to build it on top of a database-agnostic data sharing platform called DataHub [2]. Our motivation for doing so is that *Row Level Security* is geared towards addressing the problem of facilitating access control on data shared amongst users. DataHub, being a platform designed to give users the ability to perform collaborative data analysis [2], fits this use case extremely well. In this section, we provide background on DataHub and discuss how *Row Level Security* fits in with its collaborative data sharing feature.

DataHub is a hosted platform that allows users to store their data in a centralized location and better organize, manage, share, and collaborate on the data stored [2]. It consists of three key components: (1) a flexible data storage that supports data sharing and versioning, (2) an app ecosystem that helps with querying, cleaning, and visualizing data, and (3) a set of language-agnostic hooks that allows analysts to manipulate data using different languages (R, Scala, Python, etc) directly inside DataHub. Using DataHub, a user could upload a large dataset to his account, clean it using one of many apps DataHub offers, and share it with other DataHub users to collaboratively analyze together. All the components and tools that DataHub provides are to facilitate this process of online data collaboration.

However, the type of data sharing possible inside DataHub is limited to that of Discretionary Access Control. When an owner shares a dataset with another user, the user receives privileges (READ or WRITE) on the entire dataset. It is impossible to share subsets of data with individual users. The only way to do so would be to copy the subset into a separate view or table, and share the copied version. This is unfortunate, because for an application like DataHub that specializes in data collaboration, use cases requiring fine-grained access control (as those described in Section 1.2) often come up. Integrating *Row Level Security* with DataHub is a natural solution to solving this problem. Doing so will benefit DataHub by improving its data collaboration feature, and will also benefit *Row Level Security* by allowing it to be used and tested in a practical, real-life setting.

Chapter 3

System Architecture

In this chapter, we present the system architecture of *Row Level Security.* We begin with a high-level overview of how *Row Level Security* is integrated on top of DataHub. Then, we discuss in detail the various components of *Row Level Security* that work together to enforce fine-grained access control.

3.1 Overview

Row Level Security is a fine-grained access control mechanism implemented on top of DataHub to help manage and facilitate the sharing of data between users. Access control is enforced through the use of three key components: the *Security Policy Manager*, the *SQL Query Rewriter*, and the *DataHub Manager.* Figure 3-1 shows a schema of how these components are interconnected with the DataHub framework.

The *Security Policy Manager (SPM)* is responsible for managing all the security policies defined by users on shared tables. Users can create (or modify) security policies directly using the DataHub user interface or indirectly through client side applications. All security policy accesses and manipulations pass through the *SPM*, which stores policies and their changes in the backend PostgresSQL database.

The *SQL Query Rewriter* is responsible for taking in SQL queries and applying relevant security policies to the queries it receives. Upon receiving a query, the *SQL Query Rewriter* uses the *Security Policy Manager* to look up policies associated with

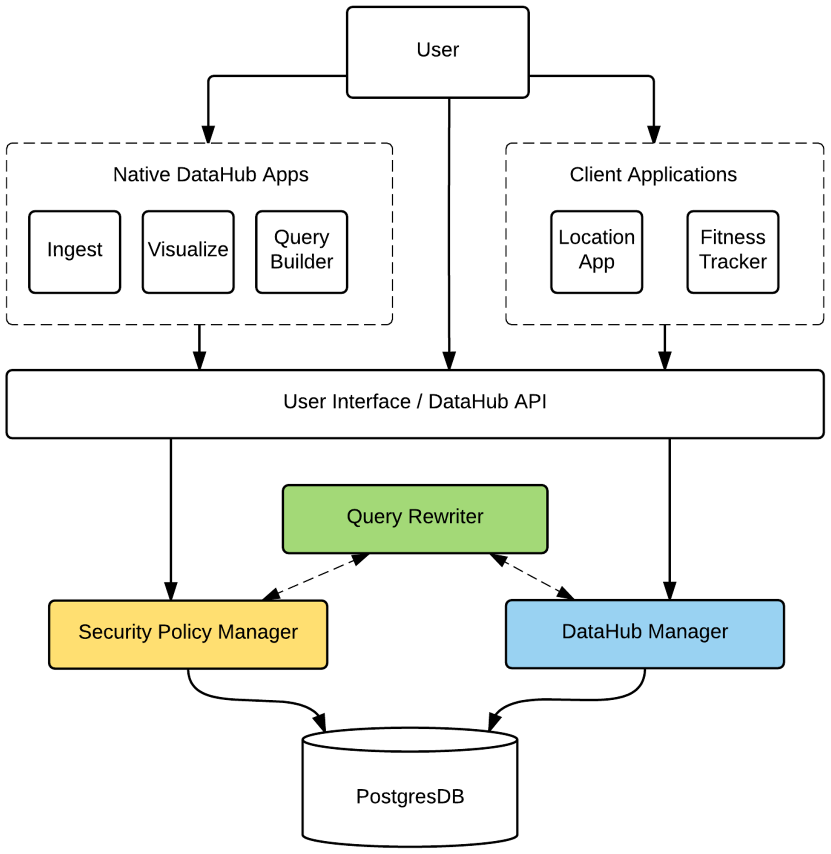


Figure 3-1: DataHub Components and Architecture

the tables accessed in the query and the user making the query call. It then rewrites the query it received to incorporate all the relevant security policies and returns the rewritten query to its caller.

The *DataHub Manager (DHM)* forms the heart of the DataHub framework and is responsible for handling all user requests that deal with manipulating data stored on DataHub. Whenever a user creates a new DataHub account, uploads a new dataset, or modifies an existing dataset in some way, the request goes through the *DataHub Manager*. *DHM* helps enforce *Row Level Security* by passing all queries executed by users (either on the DataHub UI or through DataHub/Client applications) to the *SQL Query Rewriter* to be rewritten. All rewritten queries have *Row Level Security* applied, so this is the way the system enforces *Row Level Security* access control. We will discuss the inner workings of these components in greater detail below.

3.2 Security Policy Manager

In *Row Level Security*, access control is defined through the creation of security policies. When a DataHub user shares a dataset with another user, the person the dataset is shared with is initially granted access to the entire dataset, with the privileges (READ, WRITE, ALL) granted by the owner. To restrict access in the dataset, the owner can create security policies for the user that define subsets of data for which he is able to access using the privileges given. The *Security Policy Manager (SPM)* is the component of the DataHub framework that is responsible for managing requests pertaining to the creation, modification, lookup, and deletion of security policies.

3.2.1 Security Policies

As mentioned earlier, security policies are the units in *Row Level Security* that define subsets of data a user is able to access in a shared table. DataHub users are only able to create security policies on tables that they own, for users that the table is shared with. Each security policy records the following pieces of information:

- `Table`

  This is the shared table that security policy is defined for.

- `Grantee`

  This is the user the security policy is defined for. Whenever this user executes a query that references the table specified by this policy, this security policy will be applied if the query command matches the policy type.

- `Grantor`

  This is the user that created this security policy. Due to the fact that only table owners can create security policies, the grantor is the same as the table owner.

- `Policy Type`

  This is the SQL command type that the security policy applies to. There are five policy types: SELECT, INSERT, UPDATE, DELETE, and ALL. The security policy is applied whenever the grantee executes a query against the table with a SQL command that matches this policy type. Defining the policy type to be `ALL` implies that the policy will be applied to all SQL commands executed by the grantee against the table.

- `Policy`

  This is a SQL predicate that defines the subset of records in the table the grantee has access to. Only records for which this predicate evaluates to TRUE will be accessible by the grantee.

Multiple security policies can be created for the same user on a shared table. However, each security policy must be unique. All matching policies are joined together and applied to queries executed by the user on the table.

3.2.2 Create Security Policies

There are two ways for users to create security policies on shared tables. The first is to use the *Security Policy Manager API*, which we will discuss in Section 4.1.1. The second is to use a grant access command that we have defined to explicitly create security policies for *Row Level Security.* Both methods go through the *Security Policy Manager.*

Grant Access Command

The grant access command is a way for users to create security policies in a SQL like manner. It is modeled after the way permissions are granted in Discretionary Access Control. The syntax for issuing this command is as follows:

```
GRANT [access type] ACCESS
TO [user]
ON [table name]
WHERE [policy]
```

The brackets denote information about the security policy that the user must insert. This command is passed to the *Security Policy Manager*, which parses the statement and creates the security policy if the syntax is correct.

3.2.3 Update Security Policies

Updates to security policies can only made using the *Security Policy Manager API* (Section 4.1.1). Currently, *Row Level Security* does not support any SQL like commands for users to use to update existing security policies.

3.2.4 Revoke Security Policies

Similar to the way security policies are created, there are two ways for users to revoke existing policies. One is to use the *Security Policy Manager API*, which we will discuss in Section 4.1.1. The other is to use the revoke access command.

Revoke Access Command

The revoke access command is a way for users to delete security policies in a SQL like manner. The syntax for issuing this command is as follows:

```
REVOKE [access type] ACCESS
TO [user]
ON [table name]
WHERE [policy]
```

The brackets denote information about the security policy that the user must insert. This command is again passed to the *Security Policy Manager*, which parses the statement and deletes any security policies found to match what is specified.

3.3 SQL Query Rewriter

The SQL Query Rewriter is the primary driver behind the enforcement of *Row Level Security*. It takes in queries that users want to execute, and transforms them into new ones that incorporates all the relevant security policies defined for the users and the tables accessed in the queries. The query rewriter achieves this task using the following protocol:

1. Identify all the tables accessed by the query.
2. Find all security policies defined on the tables identified in the previous step that match the command type of the query and the name of the user.
3. Rewrite the original query to include all the security policies found in the previous step.

Figure 3-2 shows the query rewriting process. We will go over it in detail in the sections below.

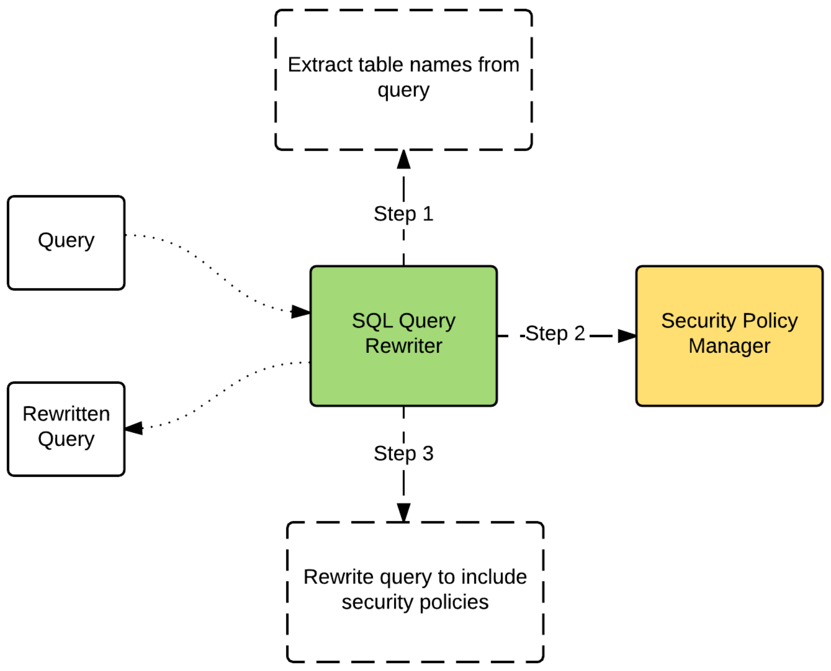


Figure 3-2: The Query Rewriting Process

3.3.1 Identifying Relevant Security Policies

The first two steps taken by the SQL Query Rewriter are fairly trivial. In the first step, the query rewriter preforms a linear scan through the query and checks on each element to see whether it refers to a table. Doing so allows it to identify all the tables referenced by the given query. For each table identified, the rewriter utilizes the *Security Policy Manager* to look up security policies defined on the table that match the command type (SELECT, INSERT, etc) of the query and the name of the user who is executing the query. This allows the rewriter to construct a list of security policies applied to each of the tables referenced in the query.

3.3.2 Rewriting the Query

The more challenging step in the query rewriting process lies with actually rewriting the query and applying the security polices identified to the given query. This step is different depending on the SQL command used to access the table. For *SELECT* commands, we replace the table names with subqueries that only select records the user executing the query is able to access. For *UPDATE* and *DELETE* commands, we append the security policy restrictions as predicates to the given query. Finally, for *INSERT* commands, we check whether the security policy allows the user to insert into the specified table – if so, the query is rewritten as is; if not, an error is raised alerting the user of the lack of permission.

SELECT Queries

For tables accessed by SELECT commands, we rewrite the query by replacing the table name with a subquery selecting the records the user is able to access in that table. The subquery achieves this task through the use of a WHERE condition that includes the predicates defined in relevant security policies – this ensures that subquery only selects records from the table that the user is granted access to see through the security policies.

In the case that multiple security policies exist on the same table for the SELECT command, the predicates are OR'ed together in the subquery. This is because different security policies could grant a user access to different subsets of data. Therefore, when a user tries to access a table, he should be granted access to the union of records that he is given access to across all the security policies defined for him.

Figure 3-3 provides a set of examples of SELECT queries before and after they are rewritten by the SQL Query Rewriter.

Security Policy	Rewrite	Query
A: [Count > 10]	Before	SELECT * FROM A;
	After	SELECT * FROM (SELECT * FROM A WHERE Count > 10);
A: [Count > 10]	Before	SELECT * FROM A WHERE ID=3;
	After	SELECT * FROM (SELECT * FROM A WHERE Count > 10) WHERE ID=3;
A: [Count > 10]	Before	SELECT * FROM A WHERE COST > 100 GROUP BY Type;

	After	SELECT * FROM (SELECT * FROM A WHERE Count > 10)
		WHERE COST > 100 GROUP BY Type;

A: [Count > 10, Name='Alice']	Before	SELECT * FROM A;
	After	SELECT * FROM (SELECT * FROM A WHERE Count > 10 OR Name='Alice');
A: [Count > 10] B: [Name='Bob']	Before	SELECT * FROM A INNER JOIN B on A.ID = B.ID;
	After	SELECT * FROM (SELECT * FROM A WHERE Count > 10) INNER JOIN (SELECT * FROM B WHERE Name='Bob') ON A.ID = B.ID;

Figure 3-3: SELECT Queries Before and After the Rewrite

Correctness Argument: Replacing table names with subqueries in the manner above successfully restricts the user's access to records that he is given permission to see. This is because the use of the subquery immediately constrains the global set of records accessed in the query to the ones the user is granted permission to access. Regardless of the operation or mixture of operations the user performs in his query – be it a selection, join, aggregate, or filter – they are acting on an already safe and restricted set of records. Thus, the result is access controlled as well, to records the user is granted permission to see.

UPDATE Queries

The way the query rewriter handles UPDATE queries is by appending the predicates of relevant security policies to the WHERE clause at the end of the queries. If no matching security policies are found, the given query is returned unchanged.

However, in the case that multiple security policies are found, the predicates of those policies are ANDed together in the WHERE clause of the update. Note that this is different from the SELECT case where predicates are ORed together. The reason this design decision is made is because we believe modifications to a record should be placed under stricter regulations than reads to the record. When there are multiple security policies defined for the UPDATE command, the user should only be able to access records that satisfy the predicates of all the policies, rather than any one of them.

Figure 3-4 provides a set of examples of UPDATE queries before and after they are rewritten by the SQL Query Rewriter.

Security Policy	Rewrite	Query

A: [Review > 5]	Before	UPDATE A
		SET Salary = 5000
		WHERE Grade = ‘A’;

	After	UPDATE A
		SET Salary = 5000
		WHERE Grade = ‘A’ AND Review > 5;

A: [Review > 5, Dept=‘Math’]	Before	UPDATE A
		SET Salary = 5000
		WHERE Grade = ‘A’;

	After	UPDATE A
		SET Salary = 5000
		WHERE Grade = ‘A’ AND Review > 5 AND Dept=‘Math’;

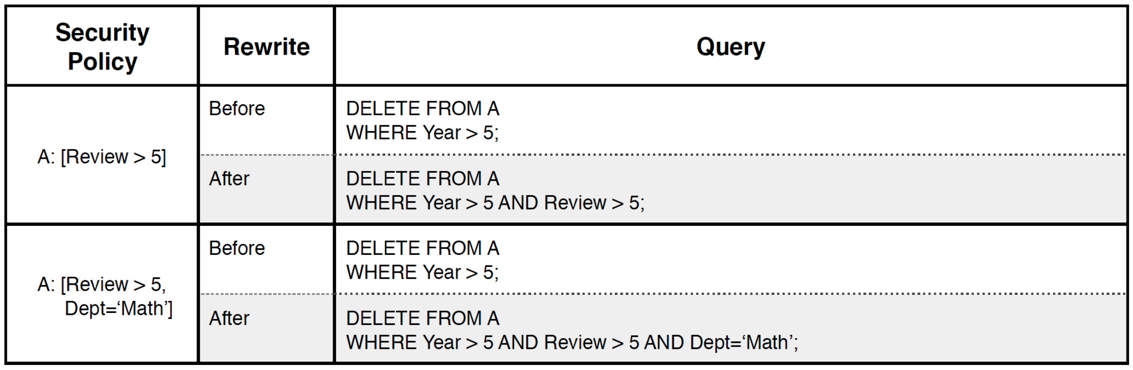
Figure 3-4: UPDATE Queries Before and After the Rewrite

Correctness Argument: Appending security policy predicates to the WHERE clause of update queries safely ensures that users are only able to update records for which they are granted permission to do so. This property holds trivially because the purpose of the WHERE clause in an update query is to define the set of records for which the update applies to. It is impossible for updates to happen to records for which predicates in the WHERE clause do not evaluate to True (correspondingly, records outside of those a user is granted permission to update).

DELETE Queries

The query rewriter rewrites DELETE queries in the same manner as it does UPDATE, which is to append the predicates of relevant security policies to the WHERE clause of the queries. If no matching security policies are found, the given query is returned unchanged. And if multiple security policies are found, then the predicates are ANDed together in the WHERE clause of the delete query. This decision is again made to impose stricter regulations on writes than reads.

Figure 3-5 provides a set of examples of DELETE queries before and after they are rewritten by the SQL Query Rewriter.



Security Policy	Rewrite	Query
A: [Review > 5]	Before	DELETE FROM A WHERE Year > 5;
	After	DELETE FROM A WHERE Year > 5 AND Review > 5;
A: [Review > 5, Dept='Math']	Before	DELETE FROM A WHERE Year > 5;
	After	DELETE FROM A WHERE Year > 5 AND Review > 5 AND Dept='Math';

Figure 3-5: DELETE Queries Before and After the Rewrite

Correctness Argument: Appending security policy predicates to the WHERE clause of DELETE queries safely ensures that users are only able to delete records for which they are granted permission to do so. This property holds trivially because the purpose of the WHERE clause is to define the set of records to delete, and the command only applies to records for which the predicates evaluate to True.

INSERT Queries

The query rewriter does not rewrite INSERT queries. Instead, it verifies that the user has permission to insert into the table specified in the query. Users who are granted insert access to shared tables have a security policy created with the policy "INSERT = True". Therefore, the query rewriter ensures that such a policy exists for the table specified in the query. If so, the rewriter returns the given query unchanged. Otherwise, it raises a error informing the user of a lack of permission.

Mixed Queries

Sometimes, the SQL queries constructed by users are more complicated than singular INSERT, SELECT, DELETE, and UPDATE queries; they feature a mixture of the four. There may be cases where an UPDATE query involves setting a value equal to something extracted from a SELECT query, or where a SELECT query consists of various nested subqueries. This is fine and does not affect the correctness of this design in enforcing *Row Level Security*.

When there are mixed queries, this query rewriting approach correctly enforces *Row Level Security* access control as long as the tables in the query are paired up with the right SQL commands when the security policy lookup is performed. This is important because it ensures the right security policies are found for each table, in the context of the SQL command that is applied to the table. Assuming the query rewriter has the right security policies for each table, the way it performs the rewrite correctly enforces *Row Level Security* access control, as demonstrated in the correctness arguments above.

When the query rewriter parses a query to look for the SQL command that corresponds to a table name in the query, it always uses the command that lies on the same parenthetical level as the table. This guarantees that the right SQL command and table are used to perform the security policy lookups. The reason this holds is that when there are mixed queries, each of the mixed queries are represented as a subquery and are enclosed inside their own parentheses. Therefore, if a SQL command rests on the same parenthetical level as a table, then they are guaranteed to match each other. This ensures that we will always find the right security policies for tables in a query, which means that the query rewriter will rewrite the query in a way that correctly enforces *Row Level Security*.

3.4 DataHub Manager

The *DataHub Manager (DHM)* forms the core of the DataHub framework and is responsible for processing all requests that deal with manipulating data stored on DataHub. Whenever an application or user registers for an account on DataHub, uploads a dataset, or modifies an existing dataset in some way, the request is passed directly to *DHM* to process.

Historically, the way *DHM* handles requests is by identifying the SQL query associated with the requests, executing them against the PostgresSQL database, and returning the response to the caller.

To integrate *Row Level Security* with DataHub, we need to change *DHM* to pass request queries to the SQL Query Rewriter first, and then run the rewritten queries against Postgres. This ensures that all the requests that DataHub users execute (and corresponding SQL queries) have *Row Level Security* applied to them.

Chapter 4

Implementation

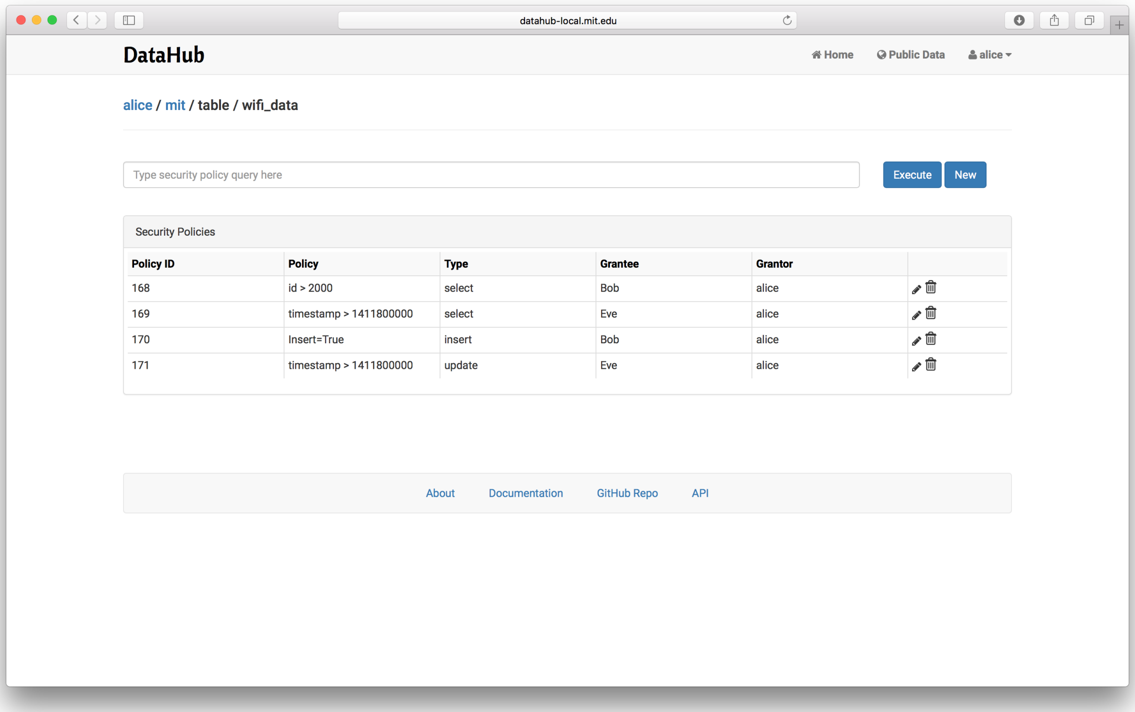
In this chapter, we describe the key implementation details of the *Security Policy Manager* and *SQL Query Rewriter*, two main components of *Row Level Security* that work to enforce fine-grained access control on top of DataHub. We begin with a brief user interface overview of how the *Row Level Security* system works before going into the finer implementation details on its various components. We then highlight interesting edge cases encountered while implementing this system.

4.1 User Interface

From the perspective of the user, there are three areas on DataHub that they could use to interact with and set up row level security for datasets they want to share: the Security Policy Page, the Security Policy Table, and the Dataset Page.

4.1.1 Security Policy Page

The Security Policy Page (Figure 4-1) is a place the user could visit for each dataset they have created on DataHub. This page displays all the security policies the user has defined for the table that it corresponds to. It also allows users to create new security policies (Figure 4-2) and modify or delete existing ones.



Policy ID	Policy	Type	Grantee	Grantor
168	id > 2000	select	Bob	alice
169	timestamp > 1411800000	select	Eve	alice
170	Insert=True	insert	Bob	alice
171	timestamp > 1411800000	update	Eve	alice

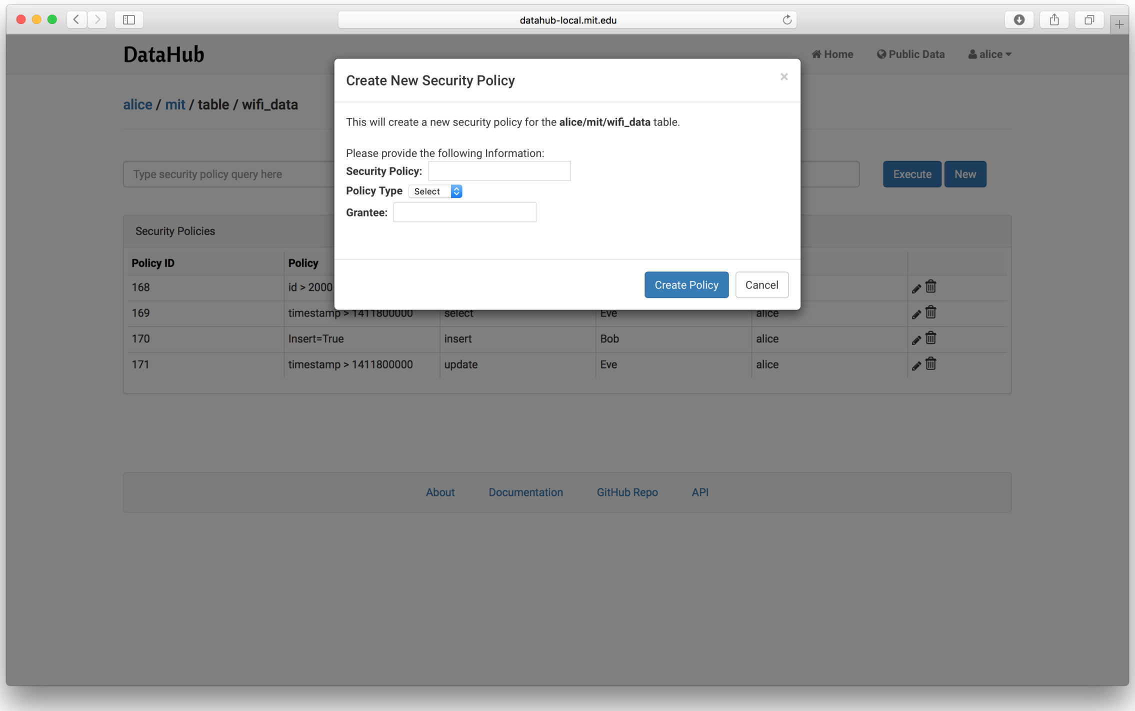
Figure 4-1: An example Security Policy Page. This page lists all security policies defined for its table, and allows users to create, modify, and delete policies.

4.1.2 Security Policy Table

The Security Policy Table (Figure 4-3) is a DataHub table that lists all the security policies the user has defined over all of his datasets. Users can issue standard SQL queries against this table to filter and modify security policies. An interesting thing to note is that all the security policies defined by DataHub users are stored inside one global Security Policy Table. However, row level security is applied to this global table to only allow users to see and modify the policies they've created. The table shown on this page is the global Security Policy Table with RLS enforced.

4.1.3 Dataset Page

The Dataset Page is the place the user visits to see the records of a table or dataset they have created or that has been shared with them on DataHub. On this page, users can type in the query they wish to execute, and press the run button to see the



alice / mit / table / wifi_data

Type security policy query here

Create New Security Policy

This will create a new security policy for the **alice/mit/wifi_data** table.

Please provide the following Information:
Security Policy:
Policy Type Select
Grantee:

Create Policy Cancel

Security Policies

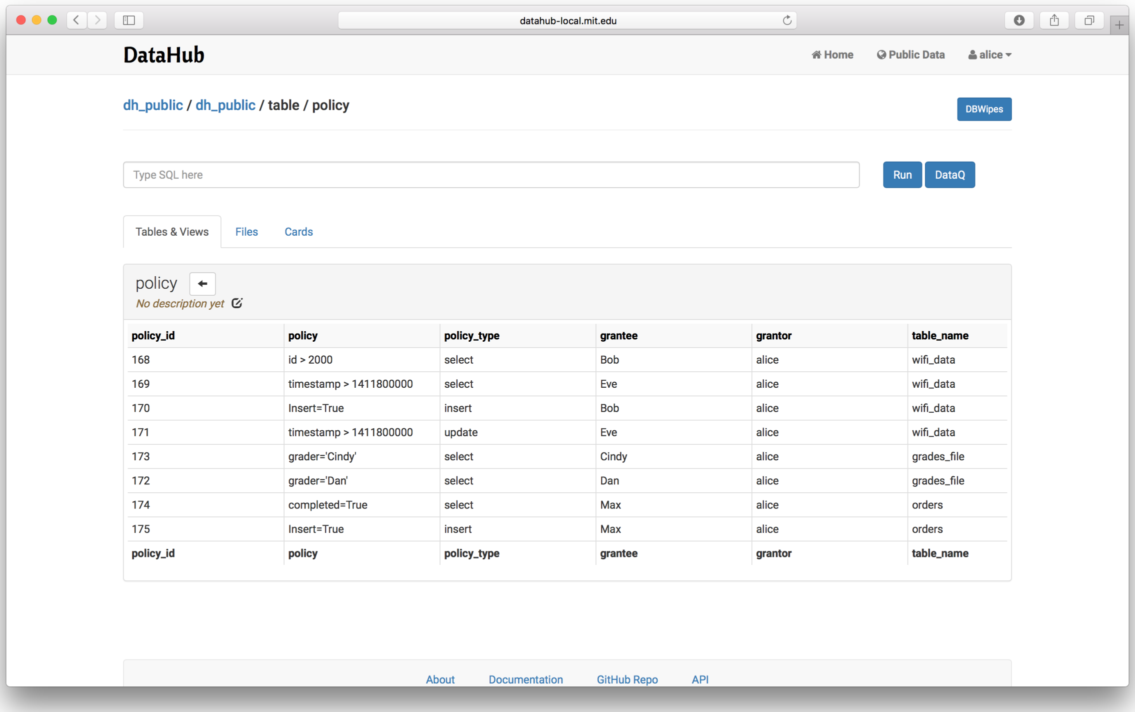
Policy ID	Policy				
168	id > 2000				
169	timestamp > 1411800000	select	Eve	alice	
170	Insert=True	insert	Bob	alice	
171	timestamp > 1411800000	update	Eve	alice	

Figure 4-2: Creating a new security policy.

results. Behind the scenes, the query entered by the user is passed to the *DataHub Manager*, which in turn calls the query rewriter to be rewritten to meet all row level security restrictions. The results that users see have *Row Level Security* applied, and are only records the users have permission to see.

4.2 Security Policy Manager

The *Security Policy Manager (SPM)* is implemented as a wrapper on top of Postgres that helps users create, modify, and delete security policies on tables that they own. It provides a core set of functions that users and applications could use to manage their security policies, as well as a special parser that allows users to create and delete policies in a SQL-like manner.

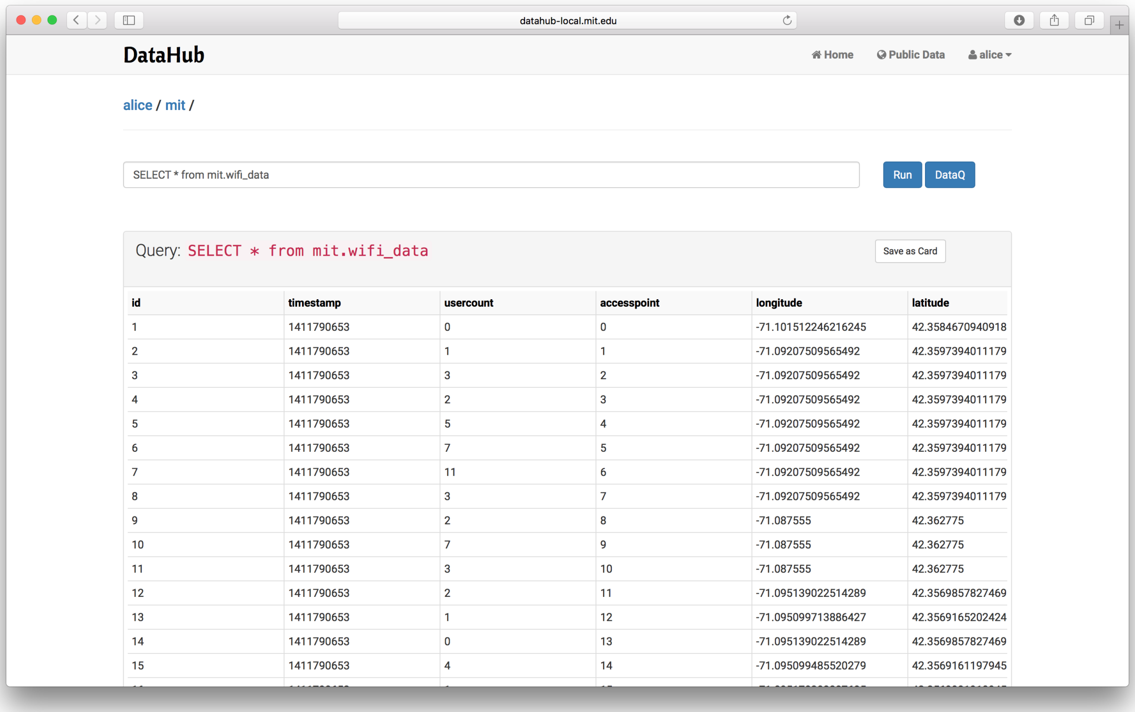


policy_id	policy	policy_type	grantee	grantor	table_name
168	id > 2000	select	Bob	alice	wifi_data
169	timestamp > 1411800000	select	Eve	alice	wifi_data
170	Insert=True	insert	Bob	alice	wifi_data
171	timestamp > 1411800000	update	Eve	alice	wifi_data
173	grader='Cindy'	select	Cindy	alice	grades_file
172	grader='Dan'	select	Dan	alice	grades_file
174	completed=True	select	Max	alice	orders
175	Insert=True	insert	Max	alice	orders
policy_id	policy	policy_type	grantee	grantor	table_name

Figure 4-3: The global Security Policy Table with RLS enforced.

4.2.1 Core API

The core methods offered by the *Security Policy Manager API* are shown below in Figure 4-5. Once users and applications successfully establish a connection with the *SPM*, they can refer to these methods directly to manage their security policies.



SELECT * from mit.wifi_data

Query: `SELECT * from mit.wifi_data`

id	timestamp	usercount	accesspoint	longitude	latitude
1	1411790653	0	0	-71.101512246216245	42.3584670940918
2	1411790653	1	1	-71.09207509565492	42.3597394011179
3	1411790653	3	2	-71.09207509565492	42.3597394011179
4	1411790653	2	3	-71.09207509565492	42.3597394011179
5	1411790653	5	4	-71.09207509565492	42.3597394011179
6	1411790653	7	5	-71.09207509565492	42.3597394011179
7	1411790653	11	6	-71.09207509565492	42.3597394011179
8	1411790653	3	7	-71.09207509565492	42.3597394011179
9	1411790653	2	8	-71.087555	42.362775
10	1411790653	7	9	-71.087555	42.362775
11	1411790653	3	10	-71.087555	42.362775
12	1411790653	2	11	-71.095139022514289	42.3569857827469
13	1411790653	1	12	-71.095099713886427	42.3569165202424
14	1411790653	0	13	-71.095139022514289	42.3569857827469
15	1411790653	4	14	-71.095099485520279	42.3569161197945

Figure 4-4: Dataset page showing Row Level Security restricted records.

4.2.2 SQL Command Parser

The *Security Policy Manager* also enables users to create and delete security policies in a SQL-like manner. The syntax for these commands are discussed in Section 3.2.2 and 3.2.4. The way this is implemented is through the use of a parser in the *SPM*. Because the syntax of the SQL-like security policy commands is predefined, the *SPM* parser uses regex to extract out policy information from the commands and the existing API to either create or remove policies the commands define.

4.3 SQL Query Rewriter

The SQL Query Rewriter is implemented as a Python class that is accessible by the DataHub Manager. It provides a public APPLY_ROW_LEVEL_SECURITY(query) method that the DataHub Manager calls to obtain rewritten queries with row level security enforced. Recall that different SQL queries (with different commands) have

Method	Specifications
create_security_policy(table, grantee, policy_type, policy)	Creates a new security policy for the grantee on the specified table if no matching policies exist.
update_security_policy(policy_id, table=None, grantee=None, policy_type=None, policy=None)	Updates the policy identified by the policy_id to satisfy the changes specified by the caller.
find_security_policy(policy_id=None, table=None, grantee=None, policy_type=None, policy=None)	Returns a list of security policies that match the criteria specified by the caller.
remove_security_policy(policy_id)	Removes the security policy identified by policy_id.
remove_matching_policies(table=None, grantee=None, policy_type=None, policy=None)	Removes all security policies that match the criteria specified by the caller.
execute_security_policy_command(sql_command)	Executes the SQL like security policy command specified by the caller.

Figure 4-5: Methods Offered by the Core Security Policy Manager API

their queries rewritten in different ways. As a result, APPLY_ROW_LEVEL_SECURITY utilizes different subroutines to rewrite queries based on their SQL commands. We will discuss the main APPLY_ROW_LEVEL_SECURITY method along with the various query rewriting subroutines below.

4.3.1 Main Query Rewrite

The main query rewriting method is called APPLY_ROW_LEVEL_SECURITY. It is a public method accessible by the DataHub Manager and is the main method used to rewrite queries to incorporate row level access control. The way this method works is it first identifies the command type of the query it is rewriting – whether it is an insert, select, delete, or update – and then delegates the rewrite to the appropriate subroutine. Again, the reason it is set up this way is because different SQL commands require different query rewrite behaviors.

In order to identify the command types of SQL queries it receives, this method uses an open source SQL library called SQLParse. SQLParse provides a PARSE method that takes in a SQL query and breaks it down into lists of tokens, each of which is either a SQL keyword, identifier, predicate, or subquery. Given the list of parsed tokens for a query, we can identify the SQL command by looking at the first token. Once the SQL command is found, the query is passed to the matching subroutine to be rewritten. The rewritten query has *Row Level Security* enforced upon it. The pseudocode for this method is presented in Algorithm 1.

Algorithm 1 The APPLY_ROW_LEVEL_SECURITY Algorithm

```
 1: function APPLY_ROW_LEVEL_SECURITY(query)
 2:     tokens ← SQLPARSE(query)
 3:     switch tokens.first do
 4:         case SELECT
 5:             return APPLY_RLS_SELECT(query)
 6:         case INSERT
 7:             return APPLY_RLS_INSERT(query)
 8:         case UPDATE
 9:             return APPLY_RLS_UPDATE(query)
10:         case DELETE
11:             return APPLY_RLS_DELETE(query)
12:     end case
13: end function
```

4.3.2 Select Subroutine

This is the subroutine used to rewrite SQL queries that have been identified to be SELECT queries. The way we do so is by first tokenizing the given query using the SQLParse library. Next, we build the rewritten query by iterating through the tokens one at a time and processing each token. If a token contains a subquery, we call the main APPLY_ROW_LEVEL_SECURITY method on the subquery (to ensure that it is row level security access restricted) and append the results to our rewritten query. If a token is the identifier for a table name, we rewrite the token into a subquery that selects from the specified table with all matching security policy predicates appended in the where, and append the subquery to the rewritten query. Otherwise, we append the token to the rewritten query as it is. Algorithm 2 presents the pseudocode for this subroutine.

Algorithm 2 The SELECT Rewrite Subroutine

```
function APPLY_RLS_SELECT(query)
    tokens ← SQLPARSE(query)
    result ← empty string
    for token in tokens do
        if CONTAINS-SUBQUERY(token) then
            result ← result + APPLY-ROW-LEVEL-SECURITY(token)
            goto for
        end if
        if not CONTAINS-TABLE(token) then
            result ← result + token
            goto for
        end if
        table ← GET-TABLE-INFO(token)
        policies ← FIND-SECURITY-POLICY(table, user, SELECT)
        result ← result + CONSTRUCT-SUBQUERY(table, policies)
    end for
    return result
end function
```

4.3.3 Insert Subroutine

This is the subroutine used to rewrite SQL queries that have been identified to be INSERT queries. The way we do so is by first tokenizing the given query using the SQLParse library. Next, we again build the rewritten query by iterating through the tokens one at a time and processing each token. If a token contains a subquery, we invoke the main APPLY_ROW_LEVEL_SECURITY method on the subquery (to ensure that it is row level security access restricted) and append the results to our rewritten query. Otherwise, we append the token to the rewritten query as it is. Before moving on, we perform a final check on the token to see if it is a table name identifier, and if so, store the name of the table.

After we're done processing all the tokens, we have a rewritten query that is row level security access controlled. However, before executing the query, we need to validate that the user has the permission to insert into the table. We do so by looking up security policies granted to the user on the table name found for the query. If there exists a policy granting the user access to insert into the table, we return the rewritten query. If not, we raise a permissions exception alerting the user to the lack of permissions. Algorithm 3 presents the pseudocode for this subroutine.

Algorithm 3 The INSERT Rewrite Subroutine

```
1: function APPLY_RLS_INSERT(query)
2:     tokens ← SQLPARSE(query)
3:     result ← empty string
4:     table ← None
5:     for token in tokens do
6:         if CONTAINS-SUBQUERY(token) then
7:             result ← result + APPLY-ROW-LEVEL-SECURITY(token)
8:             goto for
9:         end if
10:        result ← result + token
11:        if CONTAINS-TABLE(token) then
12:            table ← token
13:        end if
14:    end for
15:    policy ← FIND-SECURITY-POLICY(table, user, INSERT)
16:    if policy.predicate == "INSERT=True" then
17:        return result
18:    end if
19:    raise PERMISSIONS EXCEPTION
20: end function
```

4.3.4 Update Subroutine

This is the subroutine used to rewrite SQL queries that have been identified to be UPDATE queries. The way we do so is by first tokenizing the given query using the SQLParse library. Next, we build the rewritten query by iterating through the tokens one at a time and processing each token. If a token contains a subquery, we invoke the main APPLY_ROW_LEVEL_SECURITY method on the subquery (to ensure that it is row level security access restricted) and append the results to our rewritten query. Otherwise, we append the token to the rewritten query as it is. Before moving on, we perform a final check on the token to see if it is a table name identifier, and if so, store the name of the table.

After we're done processing all the tokens, we take the table name found and look up security policies defined on the table for the user. We then concatenate the predicates of all the security policies found in the WHERE clause of the rewritten query. This produces a rewritten query that only accesses and updates records to which the user is granted permission. Algorithm 4 presents the pseudocode for this subroutine.

Algorithm 4 The UPDATE Rewrite Subroutine

```
 1: function APPLY_RLS_UPDATE(query)
 2:     tokens ← SQLPARSE(query)
 3:     result ← empty string
 4:     table ← None
 5:     for token in tokens do
 6:         if CONTAINS-SUBQUERY(token) then
 7:             result ← result + APPLY-ROW-LEVEL-SECURITY(token)
 8:             goto for
 9:         end if
10:         result ← result + token
11:         if CONTAINS-TABLE(token) then
12:             table ← token
13:         end if
14:     end for
15:     policies ← FIND-SECURITY-POLICY(table, user, UPDATE)
16:     for policy in policies do
17:         result ← result + " AND " + policy.predicate
18:     end for
19:     return result
20: end function
```

4.3.5 Delete Subroutine

The subroutine used to rewrite DELETE queries is the exact same as that of the UPDATE subroutine. The only difference is that all instances of term "update" are replaced with "delete". The logic and implementation for both subroutines are identical, so we will not reiterate what was presented earlier.

4.3.6 Edge Cases

While implementing the *SQL Query Rewriter*, we encountered two interesting edge cases that are worthy of discussion. The first is queries that involve nested subqueries, and the second is queries that use table aliasing.

Nested Queries

Occasionally, users create complex SQL queries that involve tons of subqueries, many of which are nested within each other. With queries like these, we need to ensure that our query rewrite protocol successfully applies *row level security* to each subquery layer, so that users are truly restricted to accessing only the records they have permission to touch.

The SQL Query Rewriter achieves this task by checking each token first to make sure that it does not contain a subquery when it appends the token to the rewritten query. In the case that the rewriter detects the presence of a subquery, it will apply a query rewrite to the subquery token first, and append the result of that to the rewritten query. This is done in all of the query rewrite subroutines.

Following this approach ensures that when we rewrite a query containing nested subqueries, each subquery layer will be rewritten to be row level access restricted in a recursive manner.

Table aliasing

Sometimes SQL queries are defined in a manner that utilizes aliasing, where column names or table names are aliased to be a different (oftentimes shorter) name. Unfortunately, when SQLParse tokenizes a SQL query, it bundles the alias together with the column or table name into one token. This is fine with column names that are aliased, because the SQL Query rewriter appends those tokens to the rewritten query without modifications. However, this is problematic with aliased table names, because looking up security policies for a table name that has an alias appended is guaranteed to return incorrect results (the name will not exist).

To address this problem, the SQL query rewriter does a special parsing check when it processes table identifier tokens. Specifically, it performs an extra check to see whether the token contains an alias with the table name. If so, it extracts the table name from the alias, and uses the table name when it looks up security policies. It also ensures the alias is inserted back next to the table names in the query rewrites.

Chapter 5

Evaluation

In this chapter, we present an evaluation of the performance costs of utilizing *Row Level Security*, as well as a location tracking application that shows the ease of use in setting up row level access control with our design.

5.1 Experiments

In order to evaluate the overhead costs of using *Row Level Security* to enforce fine-grained access control, we ran various experiments measuring the slowdown in query processing that RLS introduces. Specifically, we measured the query processing throughput of using *Row Level Security* on real world WiFi data against Yahoo Cloud Serving Benchmark (YCSB) workloads. In the sections below, we will discuss the experimental setup, dataset used, performance of using *Row Level Security* on core YCSB workloads, and an analysis of the results.

5.1.1 Experimental Setup

All of our experiments were run on a single MacBook Pro machine with one quad-core Intel Core i7 processor clocked at 2.3GHz. The machine has 16GB of DRAM, 500GB of Flash Storage, and runs MAC OS X 10.11.4. For each experiment, a local instance of DataHub was started up and hosted on the machine described. The dataset used

Table 5.1: Schema of the MIT Campus WiFi Dataset

Name	Type
ID	int
Access Point ID	int
Timestamp	float64
User Count	int
Longitude	float64
Latitude	float64

was uploaded to an account created on the DataHub instance, and a Java test program issued query requests to DataHub on the uploaded dataset. The test program sent query requests to DataHub one at a time, where a new request is sent upon receiving the results from executing the previous request. The number of requests processed per second is measured as the throughput of the query.

5.1.2 Dataset

We used the MIT Campus WiFi dataset for our experiments, which contains information about the number of users connected to each one of the 3550 WiFi access points at MIT measured in 10 minute intervals. The dataset is 35MB in size and contains 509118 records total.

Each data point contains the following items: *ID* (a unique identifier for the record), *Access Point ID* (a unique identifier for the access point), *timestamp* (an epoch timestamp of when the data was recorded), *user count* (the number of users connected to the access point at that time), *Longitude* (the longitude of the access point), and *Latitude* (the latitude of the access point). Table 5.1 shows the schema of the WiFi dataset.

5.1.3 Read Only Workload

In this experiment, we measure the performance cost of using *Row Level Security* to enforce access control under a read only workload, where queries issued against the

Table 5.2: Queries issued for the read only workload

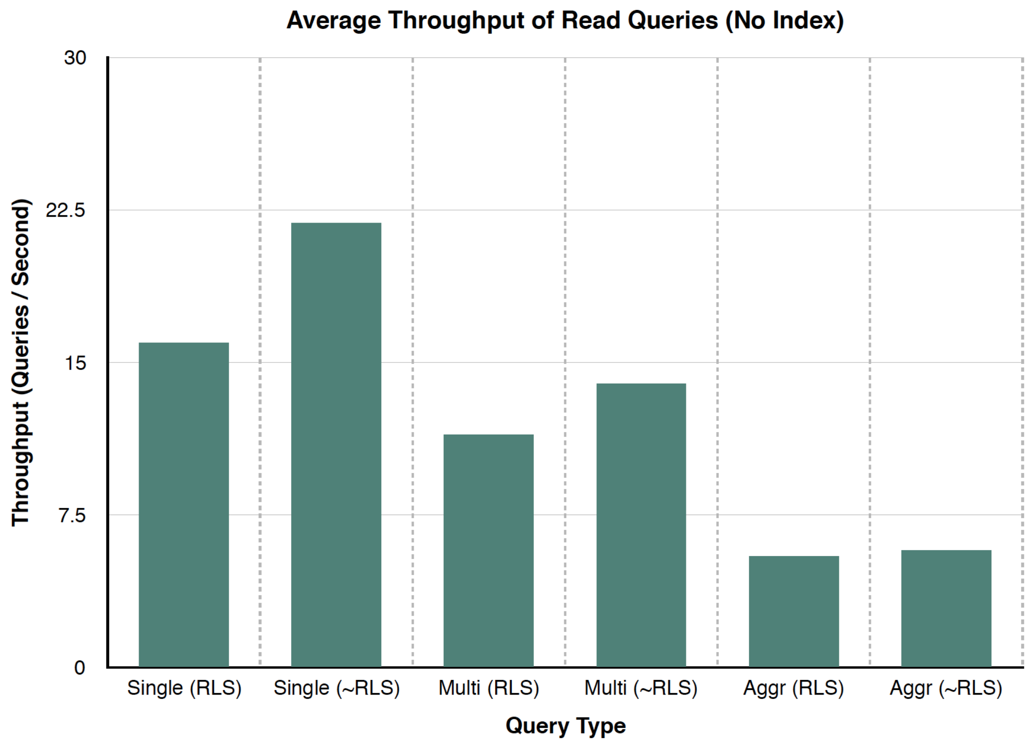
Query Type	Query
Single Read	SELECT * FROM wifidata WHERE ID=1
Multi Read	SELECT * FROM wifidata WHERE Timestamp=1411790653
Multi Read/Aggr	SELECT usercount, count(*) FROM wifidata GROUP BY Usercount

dataset are 100% reads. There are three types of query requests issued against the dataset: (1) reading a single record, (2) reading multiple records, and (3) reading multiple records and aggregating the results. Table 5.2 presents the specific queries that were executed.

We ran the queries against DataHub both in the absence of *Row Level Security* and with *Row Level Security* enabled. In the case where *Row Level Security* was used, we defined security policies granting the query executer read access to the entire dataset. We also ran the queries after setting up indexes on the columns being filtered, to see whether indexes affect the throughput of the system. From the throughputs that were obtained, we computed the query rewrite time (in milliseconds) by identifying the PostgreSQL query execution time and DataHub processing time and combining those results with the expected query processing time. Appendix A provides specific details on the computations performed to extract the query rewrite time from average throughputs.

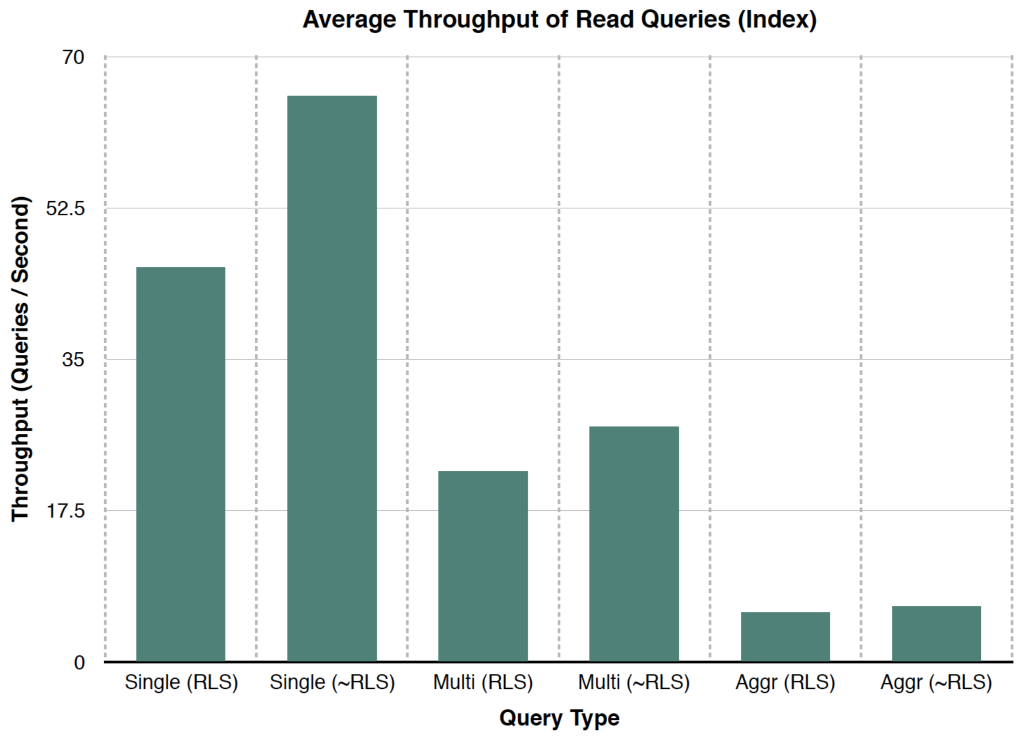
Figure 5-1 and Figure 5-2 show the average throughputs of the three queries executed with and without indexes defined. The throughputs are measured in queries per second and RLS represent queries executed with *Row Level Security* enabled, ∼RLS the queries without. Table 5-3 and Table 5-4 show a comparison of the query rewrite times and PostgreSQL execution times for the each of the queries on indexed and unindexed tables. The time is measured in milliseconds and *Row Level Security* is enabled (otherwise, there would be no query rewrites).

Figure 5-1: Average Throughput of Read Only (No Index) Queries



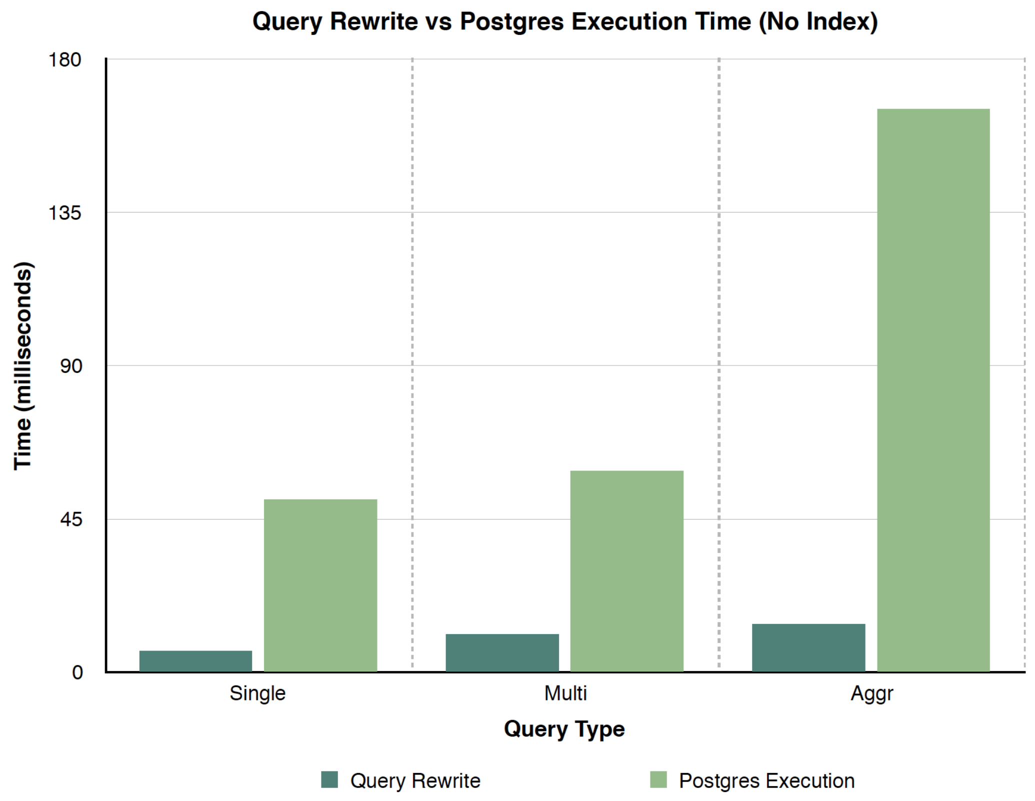
This graph presents the average throughputs of single, multi, and aggregate queries on an unindexed table. RLS signifies queries executed with *Row Level Security* enabled, and ∼RLS queries executed without.

Figure 5-2: Average Throughput of Read Only (Index) Queries



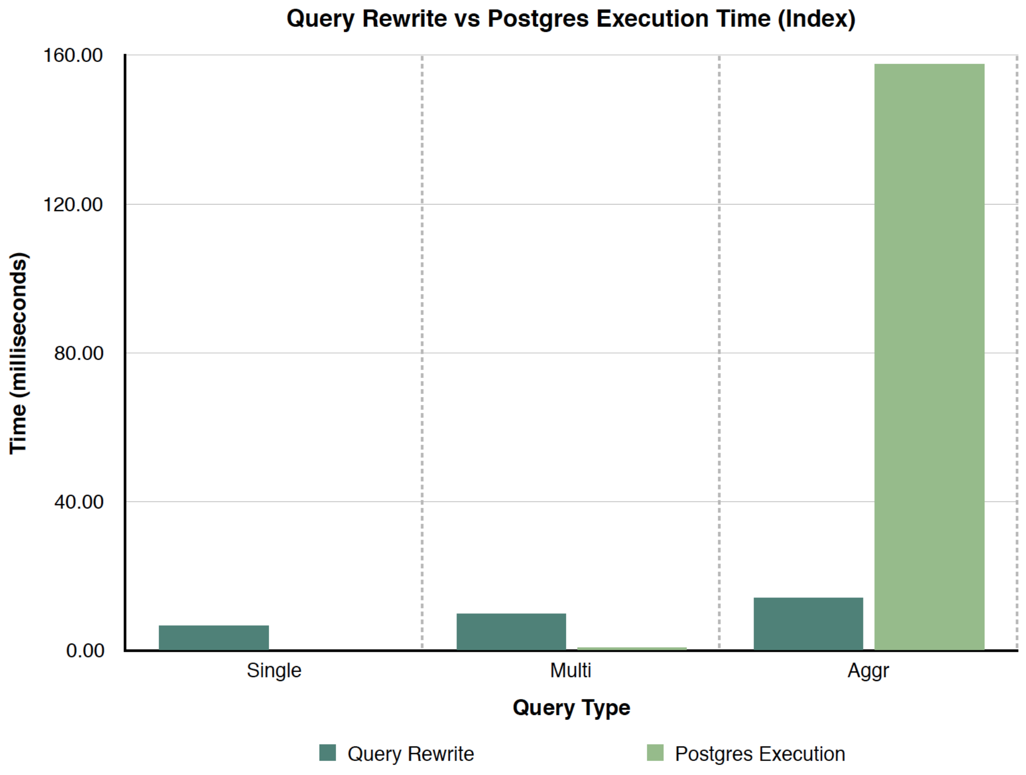
This graph presents the average throughputs of single, multi, and aggregate queries on an indexed table. RLS signifies queries executed with *Row Level Security* enabled, and ∼RLS queries executed without.

Figure 5-3: Query Rewrite vs PostgreSQL Execution Time (No Index)



This graph presents a comparison of the query rewrite time and PostgreSQL execution time over the three queries on an unindexed table.

Figure 5-4: Query Rewrite vs PostgreSQL Execution Time (Index)



This graph presents a comparison of the query rewrite time and PostgreSQL execution time over the three queries on an indexed table.

Analysis

From the data collected (Figure 5-1), we observe a 28% decrease in query throughput with the use of row level security on single queries, a 20% decrease on multi-record queries, and an 8% decrease on aggregate queries for tables without indexes. This is expected because with row level access control enabled, we introduce additional sub-query layers to the original query when we perform the query rewrite. This increase in complexity increases the time that it takes for DataHub to process the query, which in turn lowers query throughput.

When indexes are enabled on the dataset (Figure 5-2), the overall throughput for read queries increases significantly. This is expected because the main purpose of indexes is to improve query lookups. But even with the use of indexes, we observe an identical trend of decreasing query throughput when we enable row level security. For single record queries, we observe a 30% decrease, for multi-record queries a 20% decrease, and for aggregate queries a 11% decrease. This pattern is consistent with what we observed earlier on non-indexed datasets.

What is interesting about these results is the drop in throughput decreases as the query complexity increases. On both indexed and non-indexed tables, the throughput decrease drops from 30% to 20% to 10% as the complexity of the query increases. We believe this behavior is caused by the fact that as query complexity rises, the limiting factor on query throughput is the time that it takes PostgreSQL to process a query, rather than overhead costs that row level security incurs.

A closer examination of the query rewrite and PostgreSQL query execution times supports our hypothesis above. With single record queries on a non-indexed table (Figure 5-3), the query rewrite takes 6.15ms and the PostgreSQL execution takes 51ms. With multi-record queries, the rewrite takes 11.07ms and the PostgreSQL execution 58.8ms. Finally, with aggregate queries, the rewrite takes 13.8ms and the query execution 165ms. From this, we can see that the fraction of time the rewrite takes compared to the PostgresSQL execution drops significantly as query complexity increases. Thus, with simple queries, the use of row level security results in a huge difference in query throughput, as the costs of row level security is expensive relative to that of query execution. However, with more complicated queries, query execution times begin to dominate, making the costs of row level security more negligible and less noticeable. The same case holds for datasets that are indexed (Figure 5-4), where the use of an index makes the difference between rewrite and execution times even more pronounced for simpler queries.

Lastly, one final thing to note is the increase in query rewrite times as query complexity rises. This can be seen from the dark green bars on Figure 5-3 and Figure 5-4, and is expected because with more complicated queries, there are more query tokens to process and security policies to look up; resulting in a net increase in query rewrite time.

5.1.4 Write Only Workload

In this experiment, we measure the performance cost of using row level security to enforce access control under a write only workload, where queries issued against the dataset are 100% writes. There is only one query that we issued against the dataset, which was to insert a new record into the dataset, as shown below:

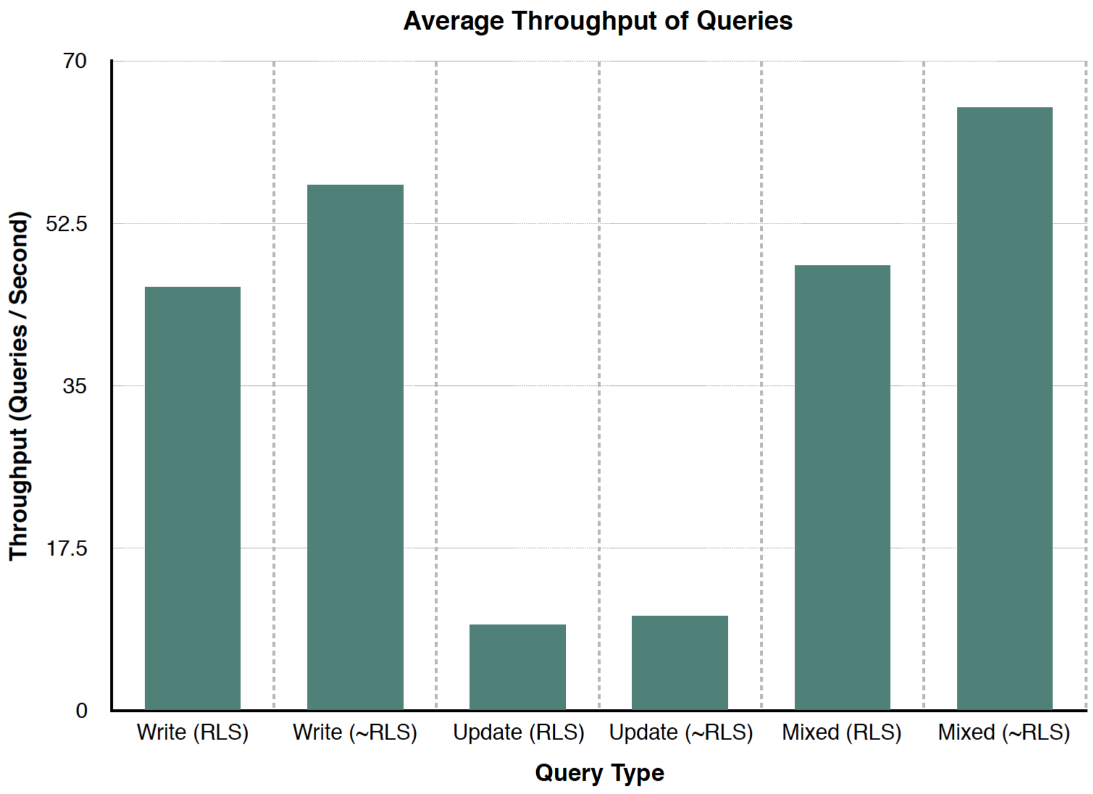
INSERT INTO wifidata VALUES(0, 0, 100000, 0, 72.314159, -72.314159)

Again, we performed the insert with and without row level security enabled. When it is enabled, the query executor is granted full insert permission to add new records to the table. The throughput of our experiment averaged over 10 trials is shown in Figure 5-5. Write(RLS) and Write(∼RLS) represent the throughputs of the query executed with and without *Row Level Security*, respectively.

Analysis

From the data collected, we observe a consistent 20% decrease in query throughput with the use of *Row Level Security.* This is expected, and is caused by the additional check that *Row Level Security* performs to ensure that the query executor has permission to insert into the table. The cost of this check (equivalent to the query rewrite time) is 3.93ms.

Figure 5-5: Average Throughput of Various Queries



In this table, we present the average throughput of write only, update only, and mixed queries as measured in queries per second.

5.1.5 Update Only Workload

In this experiment, we measure the performance cost of using row level security to enforce access control under a update only workload, where queries issued against the dataset are 100% updates. There is only one query that we issued against the dataset, which was to set the user count to zero for all records where the timestamp equals a certain value, as shown below:

UPDATE wifidata SET usercount=0 WHERE timestamp=1411796653

This query is executed both with and without row level security enabled. In the case that row level security is enabled, the query executor is granted permission to update all records in the dataset where the ID > 0. This is true for all records in the table. The throughput and query rewrite time averaged over 10 trials is shown on Figure 5-5.

Analysis

From the data collected, we observe a 22% decrease in query throughput with the use of *Row Level Security.* The slowdown is rather unsurprising and is caused by the lookup and incorporation of the security policy to the query executed. The cost of performing the query rewrite is 4.42ms.

5.1.6 Mixed Workload

In this experiment, we measure the performance cost of using row level security to enforce access control under a mixed workload involving 50% reads and 50% writes. This workload is more realistic and is modeled after the real world, where queries that users execute are oftentimes a mixture of reads and writes. The two queries executed by this workload are the single read query used in the read only workload (Section 5.1.3) and the write query used in the write only workload (Section 5.1.4). An index is defined on the ID attribute of this table, and is used by the single read query. The throughput of this experiment averaged over 10 trials is shown in Figure 5-5. We did not compute the query rewrite time for this experiment because the query type is set at random, so computing an expected time based on 50% reads and 50% writes does not generate any new value nor insight.

Analysis

From the data collected, we observe a 26% decrease in query throughput with the use of *row level security.* The throughput decrease is lower than that of a read only workload and higher than that of write and update only workloads. This is expected because this particular workload use a mixture of those queries, so it is natural to see a throughput decrease somewhere in between that of those two workloads.

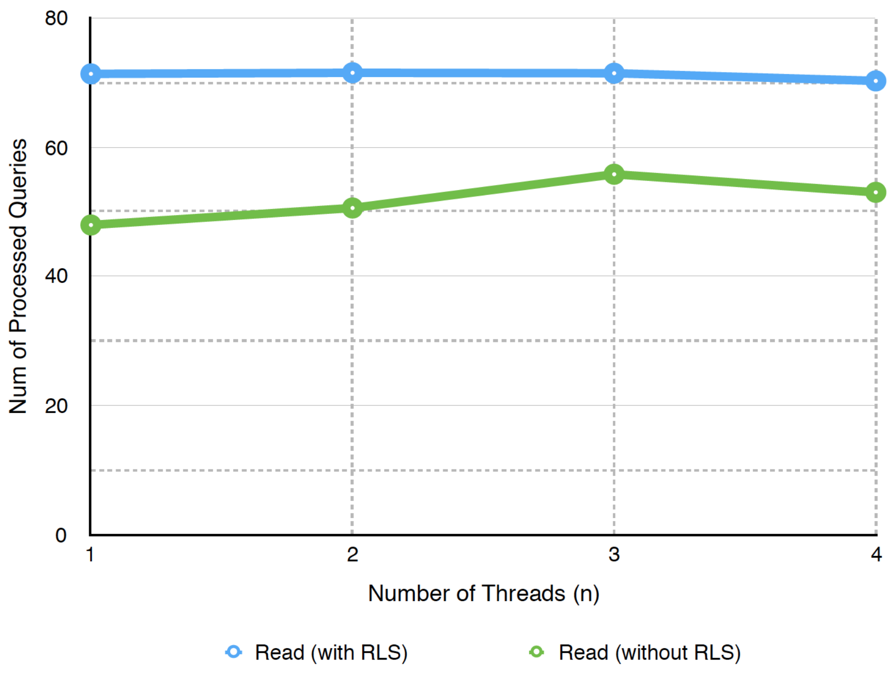
5.1.7 Query Processing Scale Up

In this final experiment, we perform a stress test that measures the maximum query throughput obtainable using DataHub in the presence and absence of *Row Level Security.* The backend PostgreSQL database is capable of processing queries concurrently up to a certain extent. This experiment is designed to assess the strain that row level security puts on PostgreSQL, and see how it affects the maximum query processing throughput that the database provides.

To perform this experiment, we revised the Java test program to spin up multiple threads and send query requests to DataHub using all of the threads concurrently (recall that each thread only sends a new request when it receives a response for its previous request). We then summed up the number of requests each thread sent to obtain the maximum number of query requests that DataHub processed for that thread count. We ran this experiment with n = 1, 2, 3, and 4 threads using the single record read query. Figure 5-6 compares the maximum query requests processed with and without *Row Level Security* enabled.

Surprisingly, we observe that the number of processed requests does not scale up with an increase in the number of threads issuing requests. Even in the absence of row level security, the total number of processed requests stay constant at approximately 70 queries as we increase the thread count. When we introduce row level security, the total requests count drop to about 50 queries, for obvious reasons described earlier.

Figure 5-6: Plot of Processed Queries Count



Further inspection into this problem revealed that the cause of this behavior is due to limitations in the DataHub framework. DataHub is implemented using the Django Python framework, and is not built to support multithreading. Thus, even when we have multiple test threads sending requests to DataHub concurrently, DataHub still processes each request one at a time. Therefore, this restriction prevents us from learning anything new through this experiment.

5.1.8 Discussion

From these experiments, we learned that as expected, the use of *Row Level Security* does introduce additional costs that decrease the query processing throughput of DataHub. For read queries, the throughput decrease is approximately 30% on tables both with and without indexes. This cost is purely for single record lookups. In cases where we have more complicated reads that require additional computations, the time to perform those computations dominates the costs introduced by row level security. For write queries, the throughput decrease is 20%, and for update queries, 22%. Finally, for workloads that involve a mixture of read and write queries, the decrease in throughput is 26%. Overall, the cost of enforcing *Row Level Security* is not too bad for the benefits that it provides. When we apply *Row Level Security* to more complicated queries (multi record and aggregate), the slowdown caused by *Row Level Security* goes down to 20% and 10%. This suggests that the overhead RLS introduces will become less noticeable as users execute queries of moderate complexity.

5.2 Location App

Finally, to evaluate the ease of use in setting up *Row Level Security* to help users access control shared datasets, we created a location tracking application that utilizes row level security to manage a table shared by an unknown number of DataHub users.

5.2.1 Purpose

The purpose of this application is to allow its users to record their current locations using the app and to later on see all the places they've recorded their locations from. All the location records are stored inside one global table owned by the application developer on DataHub. Whenever a DataHub user registers with the app, they are automatically granted access to the global table. However, the user is restricted via *Row Level Security* to only be able to read and modify his own location records. Of course, the user is granted full write access to the table, so he is able to insert new records into the table whenever he wants.

5.2.2 Design and Implementation

The way to use *Row Level Security* to meet the access control needs of this application is fairly straightforward. We just need to define three security policies on the global table for each DataHub user that registers with the app:

1. The user can only read location records for which the username in the record matches his username.

2. The user can only update location records for which the username in the record matches his username.

3. The user can insert into the table.

Once these security policies are in place, all of the user's interactions with the dataset are automatically restricted to records that he created. All the queries that he issues will be rewritten to incorporate these security policies that prevent him from accessing other users' location data.

Figure 5-7 presents the implementation of the permissions method the location app uses to grant its users access to the shared location. In this method, we first check whether the user of interest is the application owner – If so, no action is taken, as the owner already has access to the shared table. Otherwise, we create the three security policies discussed earlier for the user. This will automatically grant the user row level restricted access to the shared table.

```
def give_user_access_to_table(request, username):
    if username == owner_username:
        return
    # Grant user row level security policies on the table
    select_access = ("grant select access to {username} on locations_table "
                     "where username='{username}'").format(username=username)
    insert_access = ("grant insert access to {username} on locations_table "
                     "where INSERT='True'").format(username=username)
    delete_access = ("grant update access to {username} on locations_table "
                     "where username='{username}'").format(username=username)
    for access in [select_access, insert_access, delete_access]:
        post_security_policy(request, access)
```

Figure 5-7: Python implementation of using security policies to enforce RLS

5.2.3 Discussion

As demonstrated by the location app, it is very easy for both application developers and users to use row level security to enforce access control on shared tables. All that is necessary is the creation of security policies describing the set of records for which a user is able to access in the table (along with the type of access given). In fact, in the context of the location app, the access control objective was met using less than ten lines of code.

Chapter 6

Future work

This design of *Row Level Security* is only an initial study of the way to enforce record level access control in the context of data sharing. Although many corporations are in the process of developing and rolling out database systems that support fine-grained access control, there is very little literature that discusses how such a system could be built. We believe there are many ways this system could be optimized and improved, both in terms of safety and performance, which we will present below.

6.1 Minimize Subquery Rewrites

Currently, we rewrite all table references in SELECT queries as subqueries, regardless of whether security policies exist for that table. This results in extraneous query rewrites and policy lookups and introduces unnecessary complexity into the rewritten query for no benefit. This should be revised so that a subquery rewrite only occurs if security policies exist for the table of interest. Alternatively, an even better design would be to avoid introducing subqueries whenever possible, and to instead append policy predicates to existing WHERE clauses. This requires much more advanced query parsing and rewriting logic, especially in the case with joins and nested queries. However, it adds minimal complexity to the rewritten query.

6.2 Combine Security Policies

Currently, the predicates of all relevant security policies are appended to the rewritten query – either in a new subquery or to the WHERE condition of the original query. This may sometimes lead to redundancy when we have overlapping security policies. For example, if a user is granted read access to the record where ID=5, and read access to records where ID>3 in two separate policies, then including the predicate of the first policy in the rewritten query is unnecessary. In situations like this, we should discard the predicate of the first policy when we perform the query rewrite. Alternatively, we could remove the first security policy completely (or merge similar policies) when we detect that the user is granted second policy. This would improve the complexity of the rewritten query when users are granted many security policies.

6.3 Validate INSERT and UPDATE Queries

Finally, this idea is more of a safety improvement than a performance optimization. Currently, users are able to execute insert and update queries against a table as long as they are granted access to the table or record being modified. However, no checks are performed on the change the user makes in the query. This means that a user could theoretically update a record in such a way that he no longer has access to the record after the update, due to the change not passing existing security policies. Our system should be revised to validate changes made by users make to ensure that situations like this do not occur.

Chapter 7

Conclusion

This thesis presented a design and implementation of *Row Level Security*, a fine-grained access control mechanism built on top of a database-agnostic data sharing platform called DataHub. Our design allows users to obtain record level access control on datasets shared with others through the creation of security policies. Security policies are enforced through a query rewrite system that modifies queries executed by users to incorporate relevant restrictions. In our experiments, we studied the way the query rewrites affect the query throughput performance of DataHub. Our results indicate the use of row level security induced a 15-30% throughput decrease in query processing. However, this could be a worthy trade off to make, given the ease with which it allows applications and users to maintain and enforce record level access control.

Appendix A

Computing Query Rewrite Times

This appendix presents the method that was used to compute the calculation of query rewrite times discussed in the evaluation section.

Step 1: Compute query execution time
Recall that in all of our experiments, we measured the throughput (in queries/second) of executing each type of query. From the throughput, we can obtain the time it takes (in milliseconds) for each query to execute using:

$$\text{Query Execution} = \text{1000ms} \ / \ \text{Throughput} \tag{A.1}$$

Step 2: Compute DataHub processing time
We know that query execution time for queries without RLS is composed of:

$$\text{Query Execution} = \text{DataHub processing} + \text{Postgres execution} \tag{A.2}$$

Thus, we can compute DataHub processing time (in milliseconds) by plugging in query execution time (from A.1) and Postgres execution time (calculated by measuring the time it takes Postgres to execute the non-RLS query)

Step 3: Compute query rewrite time

Finally, we know that query execution time for queries with RLS is composed of:

$$\text{Query Execution} = \text{DataHub processing} + \text{Rewrite} + \text{Postgres execution} \qquad (A.3)$$

We know the query execution time for queries with RLS (computed using A.1), the DataHub processing time (computed using A.2), and the Postgres execution time (calculated by measuring the time it takes Postgres to execute the RLS query). Plugging these values into A.3 gives us the query rewrite time (in milliseconds) that we want.

Bibliography

[1] Bringing big data to the enterprise. `http://www.ibm.com/software/data/bigdata/what-is-big-data.html`.

[2] Datahub. `https://datahub.csail.mit.edu`.

[3] The dataverse project. `http://dataverse.org`.

[4] Row-level security. `https://msdn.microsoft.com/en-us/library/dn765131.aspx`.

[5] Row security policies. `http://www.postgresql.org/docs/9.5/static/ddl-rowsecurity.html`.

[6] Anant Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aarom L. Elmore, Samuel Madden, and Aditya G. Parameswaran. Datahub-collaborative data science and dataset version management at scale. *Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

[7] Ramaswamy Chandramouli and Ravi Sandhu. Role-based access control features in commercial database management systems. In *Proceedings of the 21st National Information Systems Security Conference*, May 2004.

[8] Bokefode Jayant D., Ubale Swapnaja A., Apte Sulabha S., and Modani Dattatray G. Analysis of dac mac rbac access control based models for security. In *International Journal of Computer Applications*, volume 104, 2014.

[9] Zhu Hong and Feng Yu-cai. Study on mandatory access control in a secure database management system. *Journal of Shanghai University*, 5:299–307, December 2001.

[10] Zachary G. Ives, Todd J. Green, Grigoris Karvounarakis, Nicholas E. Taylor, Val Tannen, Partha Pratim Talukdar, Marie Jacob, and Fernando Pereira. The orchestra collaborative data sharing system. *ACM SIGMOD Record*, 37:26–32, September 2008.

[11] Akshay Patil and Prof. B. B. Meshram. Database access control policies. *International Journal of Engineering Research and Applications*, 2:3150–3154, May-June 2012.

[12] Walid Rjaibi and Paul Bird. A multi-purpose implementation of mandatory access control in relational database management systems. In *Proceedings of the Thirtieth international conference on Very Large Databases*, number 30, pages 1010–1020, 2004.

[13] Pierangela Samarati and Sabrina Capitani de Vimercati. Access control: Policies, models, and mechanisms. *Foundations of Security Analysis and Design*, 2171:137–196, October 2001.

[14] Ravi S. Sandu. Relational database access controls. In *Handbook of Information Security Management*, pages 145–160. Auerbach Publishers, 1995.