Data security in cloud-centric multi-tenant databases

Rebecca Camilleri

Supervisor: Prof. Joseph G. Vella
Co-Supervisor: Prof. Ernest Cachia

September 2023

Submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy[1].

L-Università ta' Malta
Faculty of Information & Communication Technology

[1] https://www.um.edu.mt/ict/students/formsguidelines/

Abstract

Cloud computing has swiftly gained momentum in the technological industry, owing to its numerous benefits, notably its ability to scale, ensure availability, and lower expenses. Countless organisations have transitioned their operations to the cloud, and it is anticipated that nearly all businesses will rely on cloud-based systems in the forthcoming years.

Cloud computing has initiated significant transformations, particularly in the realm of Software as a Service (SaaS), where conventional database management systems (DBMSs) have evolved into Cloud-DBMSs (CDBMSs). Alongside this transition and within the same domain, there has been a shift from conventional single tenant database systems towards multi-tenant architectures that facilitate efficient sharing of the underlying infrastructure and resources.

Despite the prior-mentioned advantages, organisations continue to exhibit hesitancy in adopting multi-tenant database systems particularly due to concerns regarding data security. The prospect of storing data from multiple tenants on the same server, or potentially within the same database tables, amplifies apprehensions of unauthorised access. Within this context, the focus of this dissertation revolves around the domain of cloud-centric database security, with particular focus on the unique intricacies presented by multi-tenant architectures.

The aim is that of creating an automated process that aids software houses and database administrators in implementing security assertions for multi-tenant database systems prior to storing data online. This automation, in the form of a Computer-Aided Software Engineering (CASE) tool, enables the generation of tenant-specific secure database profiles, taking into account prevalent threats in CDBMSs and a comprehensive list of security requirements. A relational data model and SQL scripts are utilised as the main basis of this thesis whilst database design diagrams ensure that security is inbuilt from the initial stages of database design. Conclusively, a comprehensive analysis conducted on the proposed tool across various dimensions, including system coverage, performance, and security, demonstrates its successful attainment of the objectives established prior to its development.

The research work disclosed in this publication is partially funded by the
Endeavour Scholarship Scheme (Malta). Scholarships are part-financed
by the European Union - European Social Fund (ESF) -
Operational Programme II – Cohesion Policy 2014-2020
"*Investing in human capital to create more opportunities and promote the well-being of society*".

European Union – European Structural and Investment Funds
Operational Programme II – Cohesion Policy 2014-2020
"Investing in human capital to create more opportunities and promote the well-being of society"
Scholarships are part-financed by the European Union - European Social Funds (ESF)
Co-financing rate: 80% EU Funds;20% National Funds

Acknowledgements

The successful completion of this research journey is indebted to the inspiration, assistance, and support of numerous individuals who have played an invaluable role.
I am immensely grateful to my supervisors, Prof. Joseph G. Vella and Prof. Ernest Cachia, who not only sparked my interest in Computer Information Systems during my undergraduate years, but also granted me the opportunity to undertake this research endeavour. Their mentorship and guidance throughout the duration of this research have been instrumental in the realisation of this work.

The invaluable experience and opportunities gained through my employment as a lecturer at the CIS department at the University of Malta have also left an indelible mark on my journey. In fact, I would like to extend my special thanks to all my colleagues, whose continuous support, words of comfort, and encouragement have been instrumental throughout.

Last but certainly not least, I am forever indebted to my beloved husband, Christian, my dear parents, Oriana and Patrick, and my brother Gianluca whose unwavering support and belief in my abilities have been my anchor throughout this journey. Their continuous support, unconditional love, constant presence, and steadfast encouragement have been the cornerstone of my strength, inspiration, and the key to this achievement.

Preface

This research project spanned from October 2016 to September 2023, encompassing various stages of work. The initial phase, which involved requirements elicitation, identification of design diagrams, and the establishment of essential SQL scripts, was concluded by September 2020. These milestones were presented as part of the report submitted to fulfill the requirements for the transfer of studies from M.Phil. to a Ph.D. degree. The subsequent chapters of this thesis represent the novel work conducted between 2021 and 2023.

Throughout this research journey, several contributions emerged in the form of published works, including parts of this thesis and complementary research endeavours. These contributions are outlined below:

- Zahra, R., Vella, J.G., and Cachia, E. (2019). “Specification of requirements to ensure proper implementation of security policies in cloud-based multi-tenant systems”. International Journal of Information and Communication Engineering, vol. 13(5).
- Zahra, R., and Vella, J.G., (2020). “Incorporating security features in system design documents utilized for cloud-based databases”. In Information Systems and Management Science: Conference Proceedings of 3rd International Conference on Information Systems and Management Science, pp. 46-57. Springer International Publishing.
- Camilleri, R., and Vella, J.G., (2022), “Automated Generation of Multi-tenant Database Systems Based upon System Design Diagrams”. In 5th International Conference on Information Systems and Management Science, pp. 329-340. Springer International Publishing.

Contents

List of Figures

List of SQL Code

List of Tables

List of Abbreviations

CASE	Computer Aided Software Engineering
CDBMS	Cloud Database Management System
CRUD	Create, Read, Update, Delete
CSP	Cloud Service Provider
DAC	Discretionary Access Control
DB	Database
DBA	Database Administrator
DBMS	Database Management System
DFD	Dataflow Diagram
EBNF	Extended Backus-Naur Form
ELH	Entity Life History
ERM	Entity Relationship Management
JSON	JavaScript Object Notation
MAC	Mandatory Access Control
RBAC	Role-based Access Control
RLS	Row-level Security
SaaS	Software as a Service
SQL	Structured Query Language
UML	Unified Modelling Language

1 Introduction

1.1 Overview

Cloud computing has emerged as a rapidly advancing concept in the technology domain [1]. Regarded as a transformative force comparable to the World Wide Web [2], it has become pervasive, with 90% of organisations utilising cloud-based services as reported by the Cloud Security Report [3]. Furthermore, significant growth is anticipated for cloud technology, as demonstrated by a report from Allied Market Research which suggests that the global cloud computing industry's value could reach $1.25 trillion by 2028 [4].

Cloud Computing encompasses a progressive approach to computing, providing access to resources and services over the Internet. This technology holds significant appeal for organisations, primarily due to its multitude of benefits, such as reduced data acquisition time and lowered costs. Additionally, the flexible nature of cloud systems eliminates the need for detailed advanced preparation and resource provisioning, as they can be readily scaled as required [5].

One of the significant advancements brought about by cloud computing is the transformation of Database Management Systems (DBMS) through the introduction of Cloud Database Management Systems (CDBMSs) [6]. As per G. Kashinath and K. Vineet ‘s report [7], Cloud Database Management Systems (CDBMSs) currently generate over half of the total database revenue. This trend is driven by organisations seeking cost-effective and scalable solutions through outsourcing database storage, operations, and management.

Multi-tenancy is a prevailing architectural approach in database systems, especially within the context of Software as a Service (SaaS) applications. It refers to the capability of a software system to serve multiple users or tenants from a single shared instance of the application. Each tenant typically operates within their own logically isolated space, with their data and configurations segregated from those of other tenants. [8]. This type of architecture offers numerous benefits, including optimised resource utilisation, cost-effectiveness, simplified maintenance, and scalability [9]. The

design of multi-tenant database systems revolves around three main scenarios: separate database, separate schema, and same table, each offering distinct features and considerations, as elaborated in Section 2.7. In such systems, tenants accessing the same system may possess unique requirements, necessitating client-specific customisation to cater to their needs [10].

Ensuring data isolation and security is of paramount importance in multi-tenancy, as albeit accessing the same application, tenants must have exclusive access to their own data [9]. The Cloud Security report by Cybersecurity Insiders [11], highlights that security continues to be the primary obstacle to the widespread adoption of cloud-based solutions. The report also reveals that 76% of organisations express heightened concerns regarding the rampant nature of cloud security threats. This stems from several factors, including tenant uneasiness regarding data coexistence, a lack of evidence regarding implemented security control mechanisms by Cloud Service Providers (CSPs), and an increased risk of cyber-attacks due to data being stored online rather than on-premises [12]. Recognising these challenges, security must be addressed comprehensively to foster trust and mitigate potential risks in multi-tenant environments.

Given the complexity of the subject matter, it is not only reasonable but also rational to have security concerns. It is important to recognise that no system can be entirely foolproof, as even a single vulnerability can provide an entry point for attackers. However, the database designer must diligently address and eliminate multiple weaknesses to attain a secure system. Moreover, while security requirements can be identified, the implementation of mechanisms to fulfil those requirements often presents significant challenges. This complexity is further amplified in cloud-based multi-tenant systems, where additional security considerations come into play. This complexity is even more pronounced in cloud-based multi-tenant systems due to the introduction of additional security considerations. Firstly, security requirements in such systems are dynamic, evolving over time in response to emerging threats and vulnerabilities. Consequently, a dynamic and adaptable security approach becomes necessary, which can be intricate to implement. Secondly, in cloud-based multi-tenant systems, there are added security concerns be-cause of the shared nature of the environment. Multiple organisations or tenants utilise the same infrastructure and resources within

a multi-tenant cloud system. This shared environment introduces complexities related to data isolation, access control, and the assurance of tenant data privacy and security [12]. Moreover, the architectural characteristics of cloud computing, with its distributed nature and reliance on net-worked services, can introduce vulnerabilities that traditional on-premises systems might not encounter. Additionally, depending on external cloud service providers further compounds the complexity. Securing data and operations in an environment not under direct physical control necessitates a high level of trust and robust security measures. Unfortunately, there is a common tendency among developers to perceive security aspects as an additional burden, typically not given priority until a system failure or breach occurs [13]. This is not ideal as robust security access control mechanisms, such as role-based access control (RBAC), should be implemented throughout the development process. Implementing such controls fosters a secure environment by managing user permissions based on roles which safeguard sensitive data access [14].

Security breaches can occur at different layers within a system. While all layers are important to consider, this research primarily focuses on the top layer, namely the 'Application' layer, with a specific emphasis on cloud-based multi-tenant SaaS database systems. Database security should not be viewed in isolation, as it is influenced by various layers and components. Incorporating security features into an established database design can be challenging once the system is already in place.

It is the consideration of the limitations mentioned above (further elaborated in Chapter 2), that has led to the following research problem:

The lack of an automated software tool capable of seamlessly incorporating security requirements during the initial stages of the database design process presents a significant obstacle to the broader adoption of cloud-based multi-tenant solutions.

1.2 Research Hypothesis

In response to the critical concern delineated in Section 1.1, the introduction of an effective approach is essential. Within this context, the following research hypothesis has been formulated:

The design, development and subsequent implementation of an automated CASE tool, that incorporates multi-tenant cloud-based security requirements from the initial stages of database design are expected to yield enhanced security and operational efficiency within multi-tenant database systems.

By leveraging such a tool, software houses and database administrators (DBAs) can create secure and holistic systems tailored to meet the unique requirements of each tenant at both design and run time.

1.3 Aims and Objectives

This research endeavour aims to contribute to the field of security within the field of cloud-based multi-tenant database systems by investigating the outlined research hypothesis, mentioned in the previous section. Within this context, the aim of this dissertation is:

Development of an automated CASE tool which incorporates multi-tenant cloud-based security requirements from the initial stages of database design.

The intended users of this tool are database administrators and software houses that offer online multi-tenant applications. Following consideration of both security and user-specific needs, the proposed CASE tool adeptly determines appropriate control mechanisms to implement distinct security profiles for each tenant that align with their selected multi-tenant approach.

In achieving the above-formulated aim, a set of objectives were laid down:

1. Conducting a literature review so that common threats and risks in multi-tenant CDBMSs are identified and possible mitigation techniques discovered.
2. Presenting a holistic set of requirements for a secure-multi-tenant cloud-based system.
3. Identifying Structured Diagrams which provide a global overview of the application so that security measures can be directly implemented on these diagrams.
4. Developing automated Create, Read, Update, Delete (CRUD) matrices which assist in determining access control rights and privileges.
5. Considering the inclusion of roles interacting with database systems and the effect on security if horizontal and vertical partitioning is introduced.

6. Developing automated SQL code incorporating database objects together with roles, privileges, accesses, and other tenant-specific security considerations to automatically generate unique security profiles for each tenant.
7. Developing test databases adopted in multi-tenant SaaS applications upon which the developed SQL scripts are tested on.
8. Ensuring that the proposed tool achieves comprehensive coverage of the entire database system while minimising operational requirements to avoid disruption of normal operation.

1.4 Scope of the Problem Domain

The research conducted in this dissertation addresses the complex and multifaceted nature of security in multi-tenant cloud-based systems. The aim is to shed light on the security challenges that hinder the widespread adoption of such systems. Since this dissertation focuses specifically on security in relation to cloud-based multi-tenant databases, areas such as General Data Protection Regulation (GDPR) compliance and attacks on different layers of the system are beyond the scope of this study. Also, in the context of this dissertation, it is assumed that each tenant application operates in isolation, with no data connectivity allowed to or from other applications.

Furthermore, this dissertation primarily addresses security issues pertaining to data at rest, specifically those in relation to data that is stored within the database. While acknowledging the significance of data security during transmission between systems or over a network, including the implementation of encryption mechanisms, it is important to note that these aspects are not within the realm of this research.

Additionally, the primary focus of the proposed tool is to create individualised security profiles for tenants within a multi-tenant environment. It should be noted that while there are other users in this environment such as the Cloud Service Provider (CSP) or software developer, the setup and configuration of these users fall outside the scope of this dissertation, which concentrates on the development and implementation of tenant-specific security measures.

1.5 Dissertation structure

Chapter 2 lays the foundation by introducing essential concepts used throughout the dissertation. It offers insights into key aspects of data security during the software development lifecycle of multi-tenant cloud database systems, providing a comprehensive background for subsequent sections. Furthermore, it provides a review of relevant literature works that closely align with the dissertation's topic of interest.

Chapter 3 establishes a comprehensive set of requirements that form the foundation for designing the tool. It offers a comprehensive overview of the base requirements, whilst also encompassing both functional and non-functional ones. Furthermore, it highlights the analytical process employed to determine the most appropriate implementation strategies for meeting the specified requirements.

Chapter 4 delves into the key decisions and components related to the design of the proposed tool, highlighting the main overall structure. Special emphasis is placed on the crucial role of system design diagrams in the design process, shedding light on their importance and contribution.

Chapter 5 discusses key aspects related to access control and security. It details the creation of CRUD matrices and related checks, and introduces the implementation of horizontal and vertical partitioning. Additionally, it explains how specific tenant security specifications are addressed.

Chapter 6 focuses on the implementation of the proposed tool, delving into the various processes, assumptions, and decisions involved. It provides a comprehensive description of the key SQL scripts employed for the tool's operations.

In Chapter 7, the proposed tool is validated by means of a case study centred around the 'Banking System'. The chapter follows a step-by-step approach, providing a thorough exploration of the entire process that ultimately culminates in the generation of the final secure SQL code.

Chapter 8 is dedicated to the evaluation of the proposed tool, employing a systematic approach that combines qualitative and quantitative techniques. It assesses the tool's effectiveness in terms of system coverage, performance, and security.

Chapter 9 concludes with a summary and brief discussion of the achieved results, along with an indication of potential avenues for future research.

2 Literature Review

2.1 Introduction

In this chapter, a comprehensive groundwork is established to provide a thorough understanding of the main concepts utilised throughout this dissertation. These include notions related to cloud computing, multi-tenancy, and database systems, as well as others associated with security considerations within these domains. Additionally, the chapter delves into a review of relevant literature that aligns closely with the approach being adopted throughout this study. By engaging with this body of literature, valuable insights are drawn, fostering a robust intellectual context for the research journey.

2.2 Cloud Computing

A. Rashid et al. [15], along with other notable authors in the field, highlight how cloud computing revolutionised the realm of Information Technology (IT) particularly, in the fields of service rendition and management. Allied Market Research [7], classifies cloud computing amongst the top ten most crucial computing technologies worldwide with strong growth prospects in the upcoming years. Easy resource management, scalability and flexibility are often regarded as pivotal factors contributing to the widespread popularity of cloud computing [16], [17].

T. Garg et al. [18], further add that the upfront investment is reduced exponentially, and the overall cost minimised, particularly due to the adoption of the "pay-as-you-go" model. Additionally, the acquisition time for data to be retrieved and stored is almost negligible and cloud data can be accessed by users anytime and from anywhere with notably fast adaption time. Figure 2.1 depicts the rationale behind the cloud's exponential growth.

NIST [19], defines cloud computing as "*a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (such as networks, servers, storage, application and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*". Cloud Computing offers a technological means for users to access, work on, share and store information

using the Internet rather than the alternative conventional means of storing information on local infrastructures. Dominant service providers in the current market include Amazon, Oracle, IBM, and Microsoft [20].

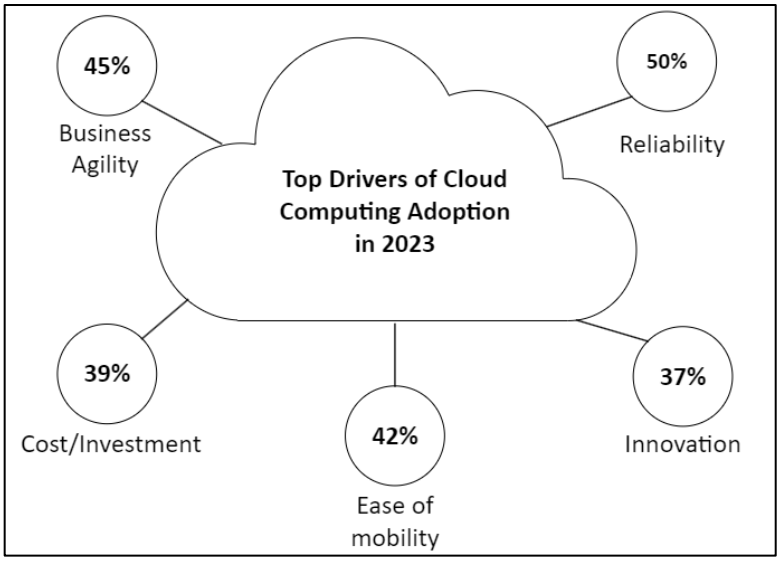


Figure 2.1: Top Drivers for Cloud Adoption [21].

P. Mell et al. [19], define cloud computing by delineating its five key characteristics. One of its primary attributes is *on-demand self-service* whereby users are granted access to technological resources and capabilities, tailored precisely to their needs, free from any interference by the service provider. Another characteristic is, *broad network access* which refers to resources being made available to users via the Internet through various platforms situated at a consumer's site. *Resource pooling* is another characteristic of cloud computing, wherein the cloud service provider shares technological resources, such as memory, storage, and bandwidth, among multiple customers. Through the creation of resource pooling technological infrastructure, the tangible technological resources become invisible to the end user who will not possess any control or knowledge over the location and formation of these physical computer resources. Furthermore, resource pooling has the potential to be applied to a multi-tenant model, which is elaborated further in the sections that follow. *Rapid Elasticity* is another cloud computing characteristic through which resources are scaled according to the user's demands. Hence, cloud computing resources become immediately available to users who may scale up or down usage according to their own needs. Lastly, *measured service* is an attribute referring to the "pay-as-you-use" model,

which is a payment method that charges the user according to resource usage. This practice is based on utility bills such as water and electricity where only the actual consumed resources are charged.

While cloud computing offers numerous benefits, it is not without its drawbacks. One significant concern is security. Storing sensitive data and applications offsite on remote servers can raise security and privacy concerns, as organisations have less direct control over their data. Data breaches and security incidents can occur, potentially compromising sensitive information. Another issue is the potential for downtime. Relying on external cloud service providers means that businesses are dependent on their uptime and network reliability, which, if disrupted, can lead to business interruptions. Additionally, there can be hidden costs associated with cloud services, as pricing models can be complex, and over time, expenses may accumulate. Finally, some organisations may face challenges with compliance and regulatory requirements when using cloud services, as legal compliance can be complex to navigate in a cloud environment [2], [22].

According to the classification by S. Subashini et al. [23], P. Mell et al. [19] and Q. Zang et al. [24], the services offered through the cloud can be grouped as follows:

1. **Software as a Service (SaaS),** which refers to the provision of the provider's software application as a service to the users. Through this model, the cloud service provider assumes responsibility for the acquisition of software, hardware, and any related updates. On the other hand, the user is simply provided with access to utilise the software application provided that there is a steady internet connection. Google Apps, Salesforce and Zoom are all examples of the SaaS model.
2. **Platform as a Service (PaaS)**, which facilitates the provisioning of an underlying platform or development environment to users, enabling them to deploy their applications without the requirement of developing or installing any platforms on their individual systems. In this model, platform layer resources including software development frameworks and operating system support are provided to the user to develop higher-level services. Google AppEngine, Microsoft Azure, and IBM Bluemix are all examples of PaaS.
3. **Infrastructure-as-a-Service (IaaS)** whereby network, hardware and storage resources are provisioned to end users in the form of services on demand. In this

model, users do not have control over the underlying infrastructure but retain the ability of managing the operating system, storage, firewall and deployed applications. Amazon EC2, Flexiscale, and GoGrid are among examples of IaaS solutions.

Further to the aforementioned models, A. Bagaeen et al. [25], S. Roy et al. [26] and [27] also refer to *Storage as a Service (STaaS)* and *Database as a Service (DBaaS)*. The StaaS model offers cloud-based storage solutions for data management and retrieval, while DBaaS provides cloud-hosted database management systems for simplified data organisation and application development.

As the domain of cloud computing continues to evolve, it has become a pivotal force in reshaping how businesses manage their data and infrastructure. In this context, the transition from traditional Database Management Systems (DBMS) to Cloud-based Database Management Systems (CDBMS) emerges as a significant transformation. This transition not only leverages the benefits of cloud computing but also necessitates a re-evaluation of security, data management, and operational strategies. In the following section, we delve into the intricacies of this transition, exploring the challenges and opportunities it presents.

2.3 Database Management Systems (DBMS)

Databases play a pivotal role in today's information-driven world, serving as the backbone of countless applications and systems. Their importance lies in their ability to efficiently store, organise, and manage vast amounts of data. By providing structured data storage, databases enable users to easily access, retrieve, and analyse information, which is essential for informed decision-making [28]. B. Rawat et al. [29], points out that data within the most prevalent database systems is commonly structured into tables consisting of rows and columns to facilitate efficient data processing and querying. He also states that the majority of databases rely on Structured Query Language (SQL) for tasks related to data input and extraction [28].

Database Management Systems (DBMS) serve as the essential bridge that enables organisations to leverage the full potential of databases. A DBMS is software designed to interact with databases, facilitating data storage, retrieval, and manipulation. It acts as an intermediary layer between users or applications and the underlying database, offering a wide array of functionalities [29].

B. Shende et al. [30] highlight various characteristics of DBMS that contribute to their effectiveness. These characteristics include ensuring data integrity and consistency by enforcing rules and constraints, thereby maintaining the quality of stored information. The ability to provide a structured framework for defining and managing the database schema, which determines how data is organised and related, is another noteworthy aspect. Moreover, DBMS offers mechanisms for efficient data indexing and searching, enabling quick retrieval of information, even from extensive datasets. Other notable features of a DBMS system, as mentioned in various literature [29], [31] sources, encompass the provision of backup and recovery facilities, support for data sharing, and data independence. Prominent databases commonly employed in the market encompass Oracle, PostgreSQL (and its variants), MySQL (and its variants), and Microsoft SQL Server. Apart from local deployment, most cloud platforms enable the effective management and administration of these databases in the cloud [29].

2.4 PostgreSQL

PostgreSQL, a versatile and open-source software that supports both relational and object-oriented databases, is collaboratively developed by the PostgreSQL Global Development Group and receives support from organisations like Red Hat and EnterpriseDB [32]. Compatible with a wide range of operating systems and nearly all major architectures, PostgreSQL's architecture follows a client-server model. Each session involves a server process responsible for managing database files, handling client connections, and executing requested database operations. Simultaneously, the client application initiates and supervises these actions on the server, allowing communication over standard internet protocols, regardless of network connectivity. This distinction is crucial to consider, as files on the client-side may not be available on the server-side. Moreover, PostgreSQL excels in managing multiple client connections to its servers, a capability common among modern server systems. It is within this context that we must emphasise PostgreSQL's multifaceted security features [32].

PostgreSQL distinguishes itself through an extensive array of robust security features. Noteworthy among its attributes are the provision of views, triggers, advanced authentication mechanisms, and the capacity to create customised functions

that fortify data protection. Moreover, PostgreSQL's versatility extends to the implementation of diverse access control paradigms, encompassing mandatory, discretionary, and role-based access control, thereby augmenting its security capabilities. Within the context of role-based access control, PostgreSQL offers the option to leverage Row-Level Security (RLS) policies, enhancing fine-grained access control [33], [34]. Moreover, PostgreSQL demonstrates adaptability, with seamless portability to the cloud, a characteristic of utmost significance in contemporary data management.

2.5 CDBMS

Cloud computing has significantly reshaped the world of databases and DBMSs. Traditional on-premises Database Management Systems (DBMS) often struggle to keep up with the changing demands of cloud computing. As a result, there is a shift towards Cloud Database Management Systems (CDBMS) models, which are typically categorised under the Software as a Service (SaaS) cloud service model. In these models, cloud service providers host and manage shared databases, offering them as services instead of conventional products. This transition marks a fundamental change in how databases are managed, reflecting the evolving needs of modern businesses seeking flexibility, and reliability in the era of cloud computing [30].

The shift from DBMS to CDBMS is motivated by several key factors. Firstly, CDBMS offers cost-effective scalability options, making it a compelling choice for organisations aiming to manage costs while accommodating growth. Additionally, CDBMS excels in handling dynamic workloads, providing enhanced performance and scalability compared to traditional on-premises solutions. This flexibility extends to development practices, offering agile development and deployment capabilities [35].

Moreover, CDBMS prioritises fault tolerance and availability, ensuring high uptime and data resilience, crucial in today's 24/7 digital landscape. On-premises databases may face limitations in terms of capacity, prompting organisations to seek the expandable infrastructure of cloud-based solutions. Reliability is further bolstered through cloud infrastructure, and robust security measures in CDBMS environments are instrumental in safeguarding valuable data [36].

CDBMS architectures are usually made up of a storage layer, a database layer and an application layer. The storage layer provides backup of data, encryption procedures and other required mechanisms to ensure the safe storage of data. The database layer on the other hand, stores the actual data of the client, whilst the application layer is responsible for authentication mechanisms and the production of reports to ensure appropriate performance of the database [37].

2.5.1 Architecture of a CDBMS

M. Kadam et al. [37], presents an illustrative depiction of a common architecture for CDBMS, shown in Figure 2.2. The user interface and application layers, which comprise the initial two layers, act as channels for communication between end users and the data residing within cloud-based database systems. Access to stored data is granted exclusively to authenticated users. The subsequent layer in the CDBMS architecture is the database layer, where the actual data and various database objects like tables, views, and stored procedures are stored. In CDBMS, this layer can consist of multiple databases or schemas, depending on the chosen mechanisms of the CSP.

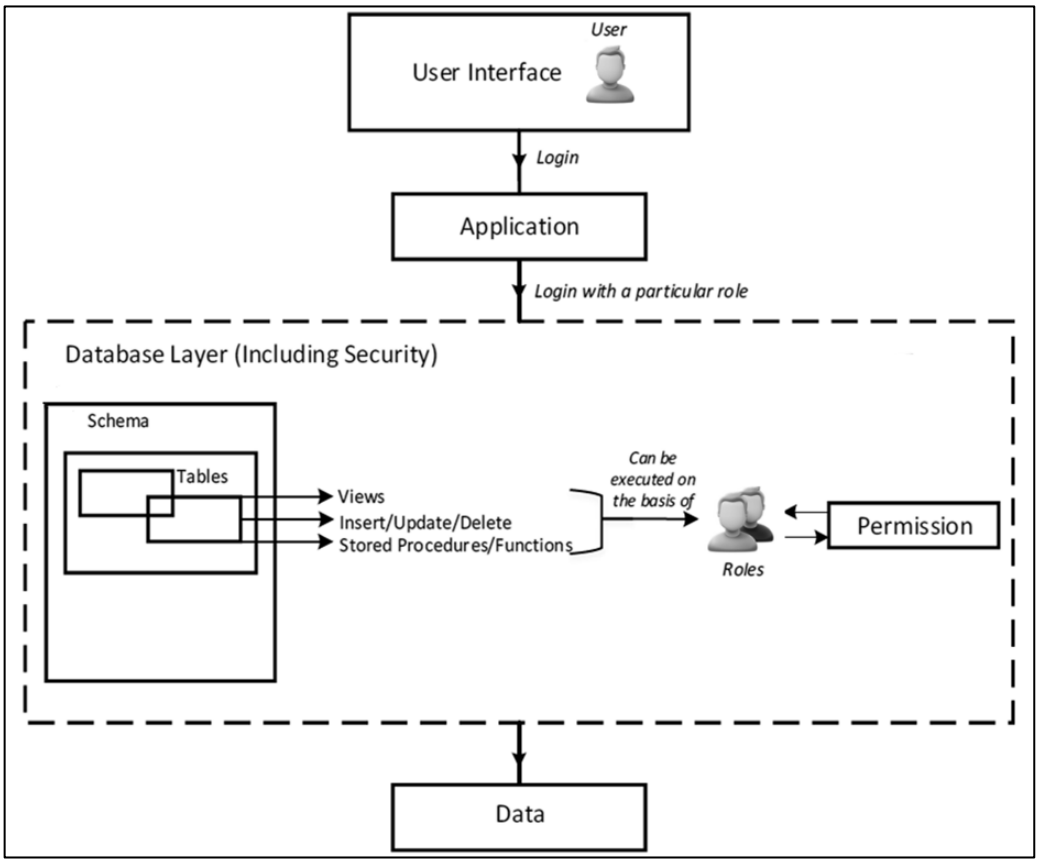


Figure 2.2: Architecture of CDBMS. Adapted from [37].

Additionally, to ensure efficiency in cases where multiple tenants utilise the same CDBMS, additional mechanisms may be included at this level. These mechanisms can involve the use of a load balancer and connection pooler. The connection pool acts as an intermediary between the database and the application layer by keeping several database connections consistently open and secured. This enables multiple client invocations and facilitates sharing between active and inactive users [35]. The final layer in the architecture, known as storage, is an essential feature for safeguarding data security. This is because, apart from the primary storage, data is typically duplicated in a separate remote location, thus enhancing data resilience. In the event of unforeseen disruptions, such as hardware failures or unauthorised data manipulation, the correct data remains readily accessible, guaranteeing its availability for recovery. The following sections provide comprehensive explanations for each feature illustrated in Figure 2.2 [37].

2.5.1.1 User and Role Creation

The establishment of users and, more specifically, roles aids in the configuration and management of a well-organised security profile. According to S. Malhotra et al. [36], defines a user is an individual with a set of credentials (typically a username and password), who can access the database system. An individual can have multiple roles, each of which may be assigned different access rights to objects. C. Türker [38], elucidates that various options are specified alongside each created role, such as the role's ability to create databases or other roles, its login capabilities, and whether it possesses superuser privileges—meaning higher privileges compared to other users. This assortment aims to grant distinct privileges to different roles. SQL Code 2.1 demonstrates the process of establishing a PostgreSQL user named 'Bob,' endowing it with login credentials, and associating it with the existing 'manager' role.

```
CREATE ROLE "bob" WITH login IN ROLE manager;
```

SQL Code 2.1: Example of Creating Role

2.5.1.2 Granting or Revoking of Privileges

In a database system, each role utilised can possess or be granted access to database objects based on their assigned privileges [39]. C. Türker et al. [38], elaborate that

privileges serve as permissions granted to users or roles and play a crucial role in implementing security mechanisms. These privileges ensure that only authorised roles are permitted to perform specific actions on database objects.

To grant access to database objects or other roles, the SQL GRANT command is employed on roles. The database objects that can be granted access include tables, columns, sequences, databases, foreign data wrappers, servers, stored procedures, functions, schemas, large objects, and tablespaces. Access rights may vary depending on the specific database object [39]. SQL Code 2.2 showcases an example of the GRANT statement, where the role 'department_manager' is granted SELECT, INSERT, UPDATE, and DELETE privileges specifically on the 'department' table.

```
GRANT SELECT, INSERT, UPDATE, DELETE
ON TABLE department
TO department_manager;
```

SQL Code 2.2: An example of GRANT command on table

Moreover, M. Rijah [40], adds that it is possible to grant membership of other roles, resulting in the inheritance of all privileges assigned to the original role by its members.

When granting this privilege, if the WITH ADMIN OPTION is specified, the newly added member gains the authority to grant or revoke membership in the role to other roles. SQL Code 2.3 provides an illustration where the role 'clerk' is granted membership to the 'employees' role without the ADMIN privilege. Consequently, the 'clerk' role does not possess the capability to grant or revoke access to the 'employees' role for other roles [39].

```
GRANT employees TO clerk;
```

SQL Code 2.3: Example of a GRANT command on role

Before executing any SQL statement on behalf of a specific role, a verification process is conducted to ensure that the privileges associated with the role are adequate to carry out the requested operation. Conversely, the REVOKE command serves as the inverse of GRANT, and it is used to revoke access rights that were previously granted to roles. This feature proves particularly valuable in dynamic scenarios

where certain users are granted access (e.g., new employees) or when others are no longer interested in accessing a specific database object (e.g., retired users) [39].

2.5.1.3 Login and Logout

A. H. Almutairi et al. [41], consider the importance of user authentication as an indispensable characteristic of any DBMS, regarding it as the foundation of any secure system. Typically, this is achieved through an individual login username and password, which can be assigned to a user or role using the WITH PASSWORD construct in PostgreSQL [33].

2.5.1.4 Database, Schema and Table Creation

Various leading scholars agree that the central element of any database management system, regardless of whether it operates locally or in the cloud, is undoubtedly the establishment of databases. To connect to the server, every user must have explicit access to a specific database, which is determined by the client's connection request [42], [43]. Z. Deineko et al. [44], clarify that every database consists of one or more schemas that act as containers. Each schema contains tables, which are the primary objects within the database, along with other elements like views.

2.5.1.5 Select/Insert/Update/Delete

At the core of database systems, data interaction revolves around four pivotal operations known as Data Manipulation Language (DML) statements. These operations are user-friendly, facilitating easy data management and viewing. These actions are exclusive to roles equipped with the necessary privileges, enabling them to perform these tasks on specific objects. The four essential operations encompass:

1. Select – employed to fetch specific rows from a table.
2. Insert – add rows to the current collection of tuples in a table.
3. Update – Modify or alter an existing value from the already existent data set.
4. Delete – Eliminate particular rows from a database table.

SQL's Data Definition Language (DDL) commands adopted for the development of databases, schemas and tables are further discussed in Chapter 6.

2.5.1.6 Views

For A. Molinaro [45], views are another essential component when it comes to implementing security measures. Views can be created on individual tables, multiple tables,

or even other views. Views serve the purpose of extracting a specific subset of data from the entire dataset, based on predetermined criteria. S. Riggs et al. [32], add that in the context of a DBMS, every user can be provided with different views of the database system. This approach is adopted to enforce data restrictions and ensure that only authorised users can access specific data. As a result, users remain unaware of any additional data beyond what is accessible through their designated views. This concept plays a crucial role in implementing discretionary access controls.

SQL Code 2.4 exhibits the code adopted for the execution of a view in PostgreSQL which extracts only particular columns from the 'products' table.

```
CREATE view products_view(product_id, product_name, units_in_stock)
AS
 SELECT product_id,
    product_name,
    units_in_stock
 FROM   product;
```

SQL Code 2.4: Example of a Create View

Literature also points out that the 'view-update' problem is a crucial concern when designing views. To ensure persistent updates, any updates made to views must also be performed on the table upon which the view is based. However, not all views defined on tables allow updates to the base table mentioned in the view. The ability to update a view relies on the structure of the query. Certain characteristics, such as employing aggregate functions or utilising the GROUP BY clause in the query, render the view non-updateable [32]. SQL code 2.5 depicts a similar view to the one shown in SQL Code 2.4. However, in this case, the view is non-updateable since the GROUP BY clause is utilised.

```
CREATE view products_view(product_id, product_name, units_in_stock)
AS
 SELECT product_id,
    product_name,
    units_in_stock
 FROM   product
 GROUP  BY product_id;
```

SQL Code 2.5: Example of a Non-Updateable View

2.5.1.7 Stored Procedures and Functions

A stored procedure, or function, is another key characteristic of DBMS architecture and refers to a collection of statements that are stored together and executed as a single call. A.R. Thakar et al. [46], underline the key distinction between these two entities. A function always returns a value after executing the SQL statements, whereas the return value in a stored procedure is optional and can be accessed through a bound parameter [34].

Stored procedures and functions are commonly used as security mechanisms as well. S. Manandhar [47], elaborates that by encapsulating logic within these procedures, users are granted access only to the procedures themselves, rather than directly accessing the underlying code or queries. This enhances security by allowing operations to be performed within a controlled environment. Additionally, the procedures themselves can include additional security checks to ensure the validity of data and the authorisation of users performing the operations. Employing correct parameter passing techniques for input within stored procedures or functions effectively mitigates database threats like SQL injection—a malicious attack that exploits vulnerabilities in input validation. This defence is established by securely integrating user inputs into queries through prepared statements, protecting the system against unauthorised commands.

In addition to the security aspects, P. Kraft et al. [48], point out that stored procedures and functions offer the advantage of being created once, stored in the database, and called multiple times as needed, thus facilitating efficient tracking of changes through a single access point.

2.5.1.8 Backups

Backups are an essential component of managing a database system. J. Logeshwaran et al. [49], emphasise the importance of backups as they offer redundancy, enabling the restoration of data in the event of an unforeseen incident that may have disrupted or corrupted the system. Furthermore, backups are instrumental in maintaining seamless business operations, a necessity in modern fast-paced, data-centric business landscape. Downtime, or periods during which systems are non-operational, can cause huge financial losses and damage a company's reputation. By restoring systems, back-

ups ensure the uninterrupted flow of essential business functions, allowing organisations to navigate disruptions and continue meeting customer demands even in adverse circumstances.

Additionally, backups provide the ability to revert to a previously functional state when required. This feature proves exceptionally useful when system modifications or updates inadvertently introduce complications. By utilising backups, organisations can return to a stable configuration, minimising the impact of emerging issues and expediting the troubleshooting process.

2.5.2 Users of CDBMS (in a Multi-Tenant Scenario)

Various scholars including R. Elmasri [50], [51] and S. Malhotra [36] refer to four predominant CDBMS users i.e. the database administrator, software developers, tenant DBA and users. Each of these stakeholders has specific roles and responsibilities, examined in detail below.

Database Administrator (DBA): The role of a DBA within a CDBMS is pivotal, given the extensive access rights associated with this position. The primary responsibilities of a DBA encompass authorising access to the entire database or specific database objects, as well as monitoring the comprehensive utilisation and management of resources among users. A more critical analysis of this role reveals both its significance and potential drawbacks. The heightened access levels required by DBAs necessitate a strong sense of responsibility and ethical considerations. If this level of access is misused or poorly managed, it can result in significant breaches of data privacy, security, and compliance.

Software Developer: The role of a software developer encompasses designing, developing, and implementing the application program. This includes tasks such as assessing client needs to determine the most suitable database implementation and conducting tests to ensure the database's performance and functionality meet the requirements. A critical perspective on this role emphasises the significance of working closely with a database administrator. This collaboration is crucial to ensuring the security and accuracy of the developed script. Relying solely on developers' judgments may result in less-than-optimal database choices, which could potentially lead to performance issues, data integrity problems, or difficulties in scaling the application to meet future demands.

Tenant DBA: The tenant DBA assumes a pivotal role in managing and administering the organisation-specific database system. Within the organisational hierarchy, this position holds the highest level of database privileges and control. However, the scope of these privileges is confined to the respective organisation, guaranteeing exclusive access to the data of their designated tenant and preventing any unauthorised access to data from other tenants.

Users (attached to tenants): Users are assigned tenant-specific roles, which vary based on the organisation's nature. In every database system, users are typically allocated roles that align with their responsibilities and access needs. Each role is characterised by a distinct set of privileges and requirements, ensuring that users maintain appropriate access levels within the system.

Figure 2.3 below, provides a visual representation of the key users and their communication, as discussed in this section.

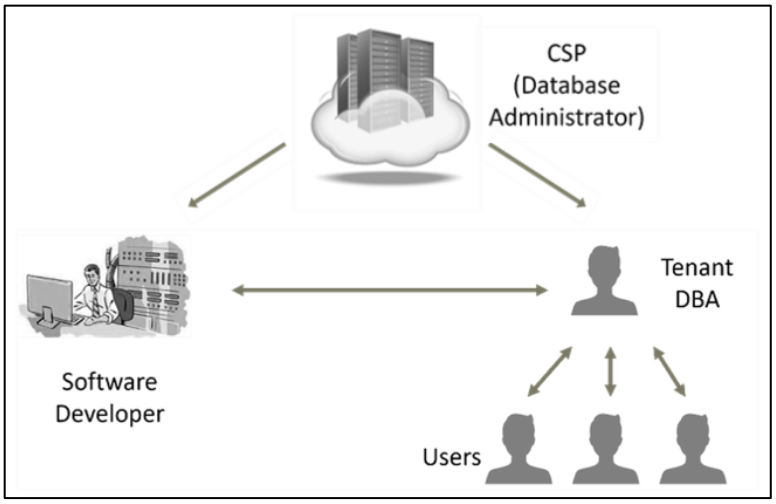


Figure 2.3: Key users in a multi-tenant cloud-based scenario. Adapted from [52].

2.6 Software as A Service (SaaS)

As highlighted in Section 2.6, SaaS is one of the primary cloud service delivery models in which users pay a monthly fee to use software rather than making a one-time purchase. Consequently, when a user makes use of a SaaS software application, the application runs from a cloud-based server rather than on the user's local device. Currently, a popular SaaS service is Microsoft's Office 365 whereby clients pay a monthly subscription fee to use Office suite applications such as Word, Excel, and PowerPoint [53], [54].

SaaS applications can be integrated within various fields to support business processes such as sales, finance, and human resources [51]. According to B. Li et al. [55], SaaS systems typically employ a predominant pricing model known as 'per-user pricing'. This means that the cost of using the system is directly tied to the number of users who are actively using it. Thus, organisations are charged according to the total number of individuals who access and utilise the SaaS system. While SaaS applications may initially appear more costly than on-premises applications, the benefits of flexibility and scalability often offset this drawback. Additionally, B. Alam et al. [56], specify that one of the primary modes of operation in SaaS applications is to serve the largest number of clients possible, allowing them to benefit from economies of scale. Software companies can offer more features and a higher level of security as a result. Customisation is typically available within the context of SaaS applications. However, software companies must ensure that the appropriate amount of customisation options are provided to tenants without straying from the main objective of a multi-tenant SaaS approach, which is to maintain a shared resource approach. It is often suggested that a 20% leeway in customisation ensures that 80% of the overall system's value is achieved [57].

Comer [58], lists various reasons why this model has gained widespread popularity. Universal access guarantee is a vital factor and implies that a user can access SaaS-based applications at any time and from any device. High availability is another reason why SaaS has experienced a surge in popularity. Data storage centres exhibit sustained operational integrity even amidst power outages due to the prevalent deployment of uninterruptible power supplies (UPS) reliant on power systems and backup infrastructure. Furthermore, data centres commonly implement redundancy by means of geographically dispersed backup mechanisms, often situated in distinct physical locations. Consequently, in the event of severe catastrophes causing damage or destruction to primary data centres, retrieval and recovery of user data can be facilitated through auxiliary backup systems.

Another notable feature of SaaS is its flexibility, exemplified by a diverse set of architectural patterns, each designed to bring distinct benefits to software distribution. The following section, explores multi-tenancy, one of the most common types of SaaS architectural patterns.

2.7 Multi-Tenancy

A primary objective of cloud service providers is to optimise resource utilisation, to achieve higher efficiency while reducing costs. This is facilitated through two key features of cloud computing: resource elasticity, which allows for flexible scaling based on demand, and multi-tenancy, which distributes software acquisition costs, licensing fees, and maintenance expenses among multiple tenants [59].

Chong and Carraro [8], explicitly mentioned the term 'multi-tenancy' for the first time whilst Bezemer and Zaidman [60], provided a more formal definition of multi-tenancy. They defined it as a mode of servicing various client organisations via one instance of a software product, hosted on the software's vendor infrastructure. Whilst multiple organisations can access the same system, their requirements may differ. Zing et al. [61], explain that such requirements are controlled by tenant-specific customisations such as the added functionality and the inclusion of data elements. Figure 2.4 depicts the primary difference between adopting an on-premises (single-tenant) scenario and a multi-tenant scenario nested at a provider. In the former, each user has their respective platform, whilst, in the latter, the application and databases are shared amongst multiple users on the provider platform.

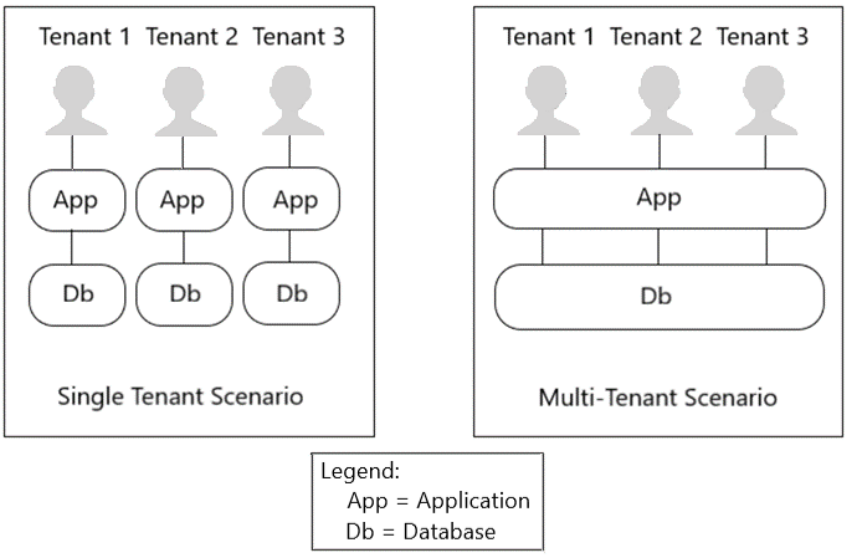


Figure 2.4: Single Tenancy vs Multi-Tenancy

The need for multi-tenancy arises from the multitude of advantages it brings. This type of architecture optimises resource utilisation and promotes cost efficiency.

By allowing multiple clients or tenants to share the same infrastructure, it minimises hardware and maintenance costs, making advanced cloud services available to a wider range of users. Scalability is another benefit of this architecture, enabling systems to effortlessly expand and adapt to varying demands. This scalability ensures that businesses can grow without being hindered by resource constraints. Additionally, multi-tenancy simplifies management, accelerates deployment, and automates updates, streamlining operations for both providers and tenants. Furthermore, the model promotes collaboration and enhances economies of scale as more participants join, fostering an ecosystem where cost-effectiveness and efficiency converge to meet the diverse needs of businesses in the digital age [62], [70].

Albeit the numerous benefits of multi-tenant adaptations, Bezemer and Zaidman [60], point out one limitation, noting that most implementations fall short in terms of providing sufficient customisation options. Another significant drawback is the potential for data security and privacy concerns. In multi-tenant environments, multiple clients or tenants share the same underlying database infrastructure. While strict access controls are put in place to isolate tenant data, there is always a risk of data leakage or unauthorised access. A security breach affecting one tenant could potentially impact the data integrity and confidentiality of other tenants.

Three dominant tactics commonly found in literature for designing multi-tenant data include: (1) Separate Database Management, (2) Separate Schema Management and (3) Same Table Management [62]-[65]. The following subsections delve into the three approaches in further detail.

2.7.1 Separate Database Management

The implementation of a separate database approach (depicted in Figure 2.5a) ensures the utmost level of data separation. Albeit sharing the same application among multiple tenants, as highlighted by F. Amadin et al. [62], this approach stores data in separate and exclusive databases. G. Karatas et al. [63], list several benefits when adopting this approach primarily that of significantly reducing security risks related to inter-tenant breaches. The author further explains that through this approach tenants will only have access to their respective databases. Moreover, since this model adopts a one-to-one approach, customisation per tenant can be realised more effectively whilst reducing the risk of compromising other tenants' system availability.

G. Karatas et al. [63] and F. Amadin et al. [62] also point out to drawbacks posed when adopting this strategy namely, the cost issue related to more storage space and resources. Compared with the other two approaches, it is the least efficient in terms of resources. As a consequence, there is a limited number of tenants which the system can house due to the limited number of databases that can be supported by the server. Additional expenses are also needed to cover the maintenance of the equipment in use in terms of hardware, software maintenance and licensing.

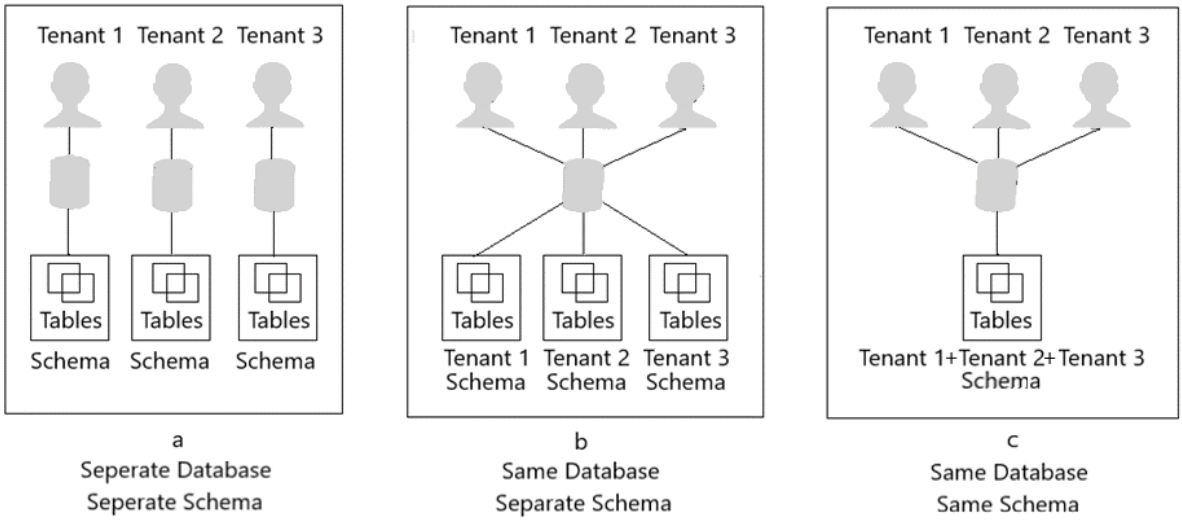


Figure 2.5: Multi-Tenant Scenarios

2.7.2 Separate Schema Management

Despite the consolidation of the tenants' data requirements within a single database instance, this multi-tenant approach ensures data separation by storing each tenant's tables or database components in separate schemas, effectively isolating them from those of other tenants. Thus, as illustrated in Figure 2.5b, upon a new tenant's subscription, a dedicated schema and its corresponding set of tables are generated within the relevant database instance. Like the previous model, according to F. Amadin et al. [62], this approach is also straightforward to implement. Among its primary benefits is the capability to incorporate additional business units as needed. Moreover, whilst the creation of tables is often based on a pre-defined set, the developer can easily add or modify columns according to the tenants' requirements thus providing ad-hoc customisation.

Apart from customisation, G. Karatas et al. [63], also add that the degree of data isolation adopted in this model is moderate as inter-schema access needs to be curtailed. This renders this approach suitable to tenants who are security-conscious but wish to find a balance between security, resource utilisation and customisation. This enables the cloud service to provide an application at a lower cost to the tenants. Apart from the benefits of this approach, A.O. Abdul et al. [66] also allude to the approach's main concerns. Specifically, they underscore the intricate nature of implementing customisations, such as fulfilling individual client needs and incorporating ad hoc modules. This complexity arises from the challenge of reusability across different tenants, potentially leading to higher expenses. The author suggests that this approach is to be avoided if multiple customisations are required.

2.7.3 Same Database and Schema (Table Management)

In the last approach, data from multiple tenants is consolidated within a single database, schema, and table, thus rendering it the most cost-effective in terms of database resource utilisation and setup simplicity (Figure 2.5c) [67]. L. Heng et al. [68], elucidate that in the same database and schema models, tables can be accessed by all authenticated users but only data appertaining to their tenancy is visible (e.g., rows in case of relational tables).

While widely endorsed by various authors in literature, including K. Jani et al. [16] —largely owing to the aforementioned advantages—G. Karatas et al. [63] offer a contrasting view. They contend that this approach presents an elevated security risk due to the collective storage of multiple tenants' data within the same table(s), and implicitly, within the same schema and database. Thus, additional mechanisms are essential to implement the system's security vis-à-vis safeguarding the tenants' data from attacks of other tenants or leakages to other tenants. Additionally, backups and recovery may not always be provided per tenant. This is because restoring a whole database or schema would mean disrupting the work of all tenants connected to that database. Another concern put forward by L. Heng et al. [68], is that the overall performance can suffer with an increase in tenants due to the increase in rows inserted in the same table. Lastly, customisations are more rigid since one-to-one approaches are not being utilised.

Following a thorough assessment of the three distinct multi-tenancy approaches delineated in this section, it becomes increasingly evident that security considerations are not just a concern but an absolute imperative. Regardless of the specific multi-tenancy approach adopted, focus on security remains pivotal. This ensures protection of sensitive data, integrity of tenant boundaries, and prevention of potential threats, all of which are foundational for establishing a robust and reliable system.

2.8 Security Factors

Multiple scholars have emphasised the criticality of security particularly in the domain of multi-tenant cloud-based database systems, as evidenced by the extensive research and publications in the field. Y. Januzaj et al. [69], argue that despite the various benefits offered by CDBMS, one of the significant barriers is security. M. Kadam et al. [37], support E. Fernandez et al.'s [70] reasoning, adding that a significant number of users still prefer to stick with local DBMSs rather than opt for a cloud-based one. The author adds that most of the security issues are challenging to monitor as they span across multiple hardware servers and data belonging to different tenants. L. M. Kaufman et al. [71] clarify that in traditional on-premises-based systems, the data, the application code and the security configurations of the organisations are stored and enforced within the organisation itself. In contrast, when utilising the cloud, data is stored remotely, and security mechanisms are owned by CSPs.

Incorporating multi-tenancy into such systems elevates the intricacy of security considerations. This escalation is predominantly a result of the increased number of potential users granted access to the system, thereby expanding the potential surface area for security vulnerabilities. Moreover, in a multi-tenant setup, since multiple clients coexist within the same database infrastructure, distinct data access requirements, security configurations, and levels of data sensitivity might be required. Managing these diverse requirements within a shared environment presents intricate security challenges.

2.8.1 Security Principles

Security in information systems relies not only on specific mechanisms but also on overarching principles that underpin effective security practices. These principles serve as guiding pillars, shaping the design, implementation, and operation of secure

systems across diverse environments. Jangjou et al. [12] mention the principle of *economy of mechanism*, which underscores simplicity and minimalism in system design. It advocates for streamlined architectures that minimise complexity and reduce the potential for vulnerabilities. By adhering to this principle, system designers can mitigate the risks associated with overly intricate systems and enhance the manageability and resilience of security mechanisms.

Another significant security principle, as highlighted by [12], [59], [70], is that of *open design*. This principle promotes transparency and relies on the openness of system design rather than secrecy to ensure security. By making security mechanisms and protocols publicly accessible, organisations foster collaboration and scrutiny, enabling the identification and resolution of vulnerabilities by a broader community of experts.

Separation of duties is another crucial principle mentioned by various scholars such as [12], [59], [70] that enhances security by requiring multiple authorisation steps for critical actions. By distributing access privileges across multiple entities, it ensures accountability and mitigates the impact of insider threats. Furthermore, Kaufman and Lori [71] make reference to the principle of least privilege which dictates that users and processes should operate with the minimum privileges necessary to perform their tasks. By restricting access rights to essential functions, organisations minimise the attack surface and mitigate the potential impact of security breaches.

2.8.2 Security Perspectives in Multi-Tenant Systems

In multi-tenant environments, the overall security of the system is equivalent to the weakest security feature present in the infrastructure. To address tenants' security-related concerns, S. Aldossary et al. [54] recommend aligning the implementation of security mechanisms by the CSP with customers' requirements. Building upon this, B. R. Kandukuri et al. [72] emphasise the importance of establishing security policies alongside security mechanisms to ensure a consistent and resilient security framework.

Additionally, M. S. Ahmad et al. [73] highlight that security enforcement typically involves a combination of prevention and detection measures. Prevention refers to taking the necessary precautions to ensure that security breaches are avoided whilst detection refers to the identification of any anomalies in the system's setup and

operations. Moreover, according to this author, it is usually advisable to have a proper defence-in-depth security strategy. Such a strategy utilises multiple security mechanisms to protect information, thus ensuring that should one of the mechanisms fail, others can be used.

2.8.3 Security Considerations in Multi-Tenant Cloud-Based Systems

S. Kanade and R. Manza [74], examine numerous security risks of multi-tenant cloud-based systems, which are subsequently corroborated by the research conducted by Brown et al. [75]. According to these authors, the main security concerns in this specific area of SaaS implementation include:

1. **Integrity** – ensuring the reliability and trustworthiness of data.
2. **Availability** – ensuring timely delivery of information to authorised users.
3. **Confidentiality** – guaranteeing restricted access to information, limited to authorised users only.

In relation to the above-mentioned security concerns, M. Ali et al. [76], observe that various studies in the area such as those presented by C. Rong et al. [77], S. Subashini et al. [23] and K. Hashizume et al. [78] focus on security issues and fail to provide solutions. Other works in the field, do propose some prevention techniques, however, they still exhibit some limitations. For instance, Z. Xiao et al. [79], specifically address security and privacy issues in multi-tenancy cloud computing and provide defence strategies for existing vulnerabilities. However, the discussion does not encompass underlying technologies that contribute to the emergence of these weaknesses.

B. Alouffi et al. [80], state that proper implementation of security policies in cloud-based multi-tenant systems can only take place if key security elements and threats are identified from the very beginning. The author distinguishes between security elements and security threats. Whilst the former refers to those areas, which should be considered for a system to be delineated as secure, the latter, refers to any dangers, which might intentionally or otherwise affect the smooth running of the system. Apart from the prior-mentioned requirements of confidentiality. Integrity and availability, common, security elements cited in literature [80] include:

1. **Static/Stored Data Security** - safeguarding data from unauthorised access.
2. **Network Security** – the adaptation of policies to monitor and prevent unauthorised access or misuse of network resources.

3. **Security Vulnerabilities** - weak spots in the system which can be exploited by malicious users.
4. **Attack Detection** – detecting timely security breaches to enable the deployment of effective mitigation measures.
5. **Auditing** – keeping logs of any operations conducted on the system to identify any potentially suspicious processes.

Identifying prevalent security threats in multi-tenant cloud-based systems is crucial to ensuring a robust and secure system. Multiple scholarly works have explored security threats in cloud systems, with notable contributions from various researchers. In 2010, the Cloud Security Alliance (CSA) conducted comprehensive research in this area, identifying the primary threats [81]. Additionally, works by Islam et al. [82], M. Kazim et al. [83], S. Kumar et al. [84] and D. Amalarethinam et al. [85] have examined and documented these threats. The following compilation represents a synthesis of the most frequently cited and commonly observed threats in literature.

1. **Misuse of Cloud**

Malicious operators exploit vulnerabilities in the system, leading to detrimental consequences for end users [86]. S. W. Bonasera et al. [87], identify Denial of Service (DoS) attacks and password cracking as primary security threats that pose risks to system availability and tenants' confidentiality and integrity, respectively. The Cloud Security Alliance [88], also argues that proper management of resources is crucial in cloud computing, as the resources are shared among multiple tenants. If one tenant consumes a large portion of the available resources, other tenants may experience a shortage of resources, resulting in limited availability. The impact of this can be comparable to that of a denial-of-service attack, as highlighted by S. Aldossary et al. [54].

2. **Vulnerable APIs and interfaces**

The Cloud Security Alliance [88], highlights the vulnerability of unprotected APIs and Interfaces as a prevalent security threat. Application Programming Interfaces (APIs), used by end users to connect and interact with cloud-based applications, require measures such as authentication and encryption for enhanced security.

3. **Malicious threats from internal users**

T. Islam et al. [82], explain that this security threat stems from individuals with access privileges, including system administrators, employees, contractors, or third-

party service providers. These individuals may misuse their access, intentionally causing harm and potentially compromising the confidentiality, integrity, and availability of data within the organisation. G. Verma et al. [89], accentuate the importance of addressing malicious threats from internal users. The author remarks that the importance of this issue is often underestimated when compared to addressing external threats. He argues that internal threats can be more perilous than external threats in certain situations because insiders have access to more in-depth information, making them more effective and capable of causing greater harm.

4. **Issues encountered due to shared technology**

Various literature works emphasise the importance of tenant data isolation in multi-tenancy. Organisations need to ensure that their data does not overlap with that of others, who may be competitors. However, guaranteeing data isolation while allowing access to shared resources is a complex task that requires careful planning [83]. Within this context, S. K. Sood [90], advocates for the adoption of a well-planned defence-in-depth security strategy. The concept of defence-in-depth involves implementing multiple layers of security measures to bolster protection. This approach guarantees that users are granted access based on their assigned level and the specific context of their interaction. Additionally, B. Alouffi [80], recommends the appointment of a mediator responsible for granting access to shared resources.

5. **Data leakages and losses**

Another security threat presented by B. Alouffi [80], is the loss of cloud data due to server crashes, accidental deletion or inadequate backups. To prevent such issues S. Aldossary et al. [54], counsel for the utilisation of appropriate backup procedures and data encryption both when stored and transmitted.

6. **Attacks from external individuals**

Man-in-the-Middle attacks, backdoor channel attacks, zombie and DoS attacks are amongst the common external attacks highlighted by D. Amalarethinam et al. [85]. Section 2.8.3 highlights additional common attacks typically associated with DBMSs.

7. **Failure in the security of the provider**

The integrity of the CSP is a critical security concern that should not be overlooked. It is essential to have a thorough understanding of the potential consequences

in the event of a server crash or temporary data unavailability. Thus, comprehensive planning of mitigation techniques is vital to address failures in security policies [88].

2.8.4 Database Security in Multi-Tenant Cloud-Based Systems

As emphasised in the preceding sections, security is an indispensable aspect of any system. In a notable security incident which occurred in May 2019, a DBMS-related security breach resulted in the exposure of sixteen years' worth of insurance holders' data. The incident involved First American Financial Corporation, a real estate insurance company, which unintentionally exposed 1.3 billion sensitive records including bank account numbers, statements, tax records, social security numbers, and photos. Shockingly, these records were accessible to anyone with an Internet connection, completely bypassing the need for authentication [91]. Thus, from this accident and other similar ones, one can assert that the inclusion of security policies from the initial stages of database design is not only beneficial but indispensable.

Prior to the incorporation of security measures during the database system's design phase, it is crucial to identify security risks associated with the DBMS. These may originate from individuals within the organisation as well as external sources. S. Kulkarni et al. [92] and M. Mousa [93] elicit commonly cited DBMS threats, including:

1. **SQL Injection Attack** - This involves the insertion or manipulation of SQL code within input fields with the intention of exploiting vulnerabilities to gain unauthorised access to databases or execute unauthorised database operations.
2. **Misuse of Authority** – This refers to authorised users taking advantage of their privileges to maliciously access or destroy data in a database.
3. **Bypassing Controls** – This involves bypassing security measures and controls meant to limit unauthorised database access, often through tactics like exploiting software vulnerabilities, misconfigurations, or unauthorised methods to access data or perform restricted actions.
4. **Trojan Horse** – This represents malicious logic that, once installed, carries out harmful actions like data theft or corruption while pretending to be a legitimate database tool.
5. **Masquerading** - This describes a deceptive tactic, where an unauthorised entity or user impersonates a legitimate user or system to illicitly access the database. This

may encompass the use of stolen credentials or the exploitation of system vulnerabilities to mimic authorised access.

6. **Hardware and Media Attacks** - These encompass physical breaches and manipulations of hardware or storage media, which can lead to data theft, tampering, corruption or unauthorised utilisation of resources.

The database security threats discussed prior remain relevant when transitioning to cloud databases. Moreover, they might have a more significant impact, particularly within multi-tenant environments where multiple clients or tenants collaboratively utilise the same database infrastructure. The shared nature of these environments increases the complexity of security concerns related to data isolation, privacy, and the potential for one tenant's vulnerabilities to affect others.

Within the context of cloud-based databases, the key security risks identified in literature encompass concerns regarding availability, unreliable access controls, inadequate monitoring and auditing tools, absence of proper encryption, isolation shortcomings, and the lack of effective data sanitisation mechanisms [94], [10]. Additionally, as highlighted by Jangjou and Sohrabi [12], the prominence of hardware and media attacks gains increased significance within the domain of cloud databases, echoing concerns associated with traditional DBMS. This heightened risk stems from the potential accumulation of unused resources, often triggered by an increased number of tenants and expanded system complexity. These underutilised resources pose security risks as potential entry points for attackers and also contribute to the challenge of pinpointing the source of a security breach.

Each of the threats discussed in this section can be mitigated through the implementation of various mechanisms, including robust access controls, encryption, regular backups, and intrusion detection systems. Comprehensive consideration of these threats and the implementation of a holistic security framework are essential for minimising the risk of security breaches and ensuring the continuous protection of data. This proactive approach guarantees the protection of sensitive data while capitalising on the advantages offered by database systems, both locally or on the cloud.

The security considerations outlined within this section are pivotal in constructing a resilient system. Among these considerations, access control stands out as one of the most critical aspects. Beyond the fundamental task of safeguarding data

and resources, a comprehensive security strategy necessitates precise control over access permissions. Within this context, the subsequent section examines various access control mechanisms and their practical applications.

2.9 Access Controls

Access controls play a critical role in addressing security concerns and mitigating threats, particularly within the context of multi-tenant cloud-based systems. They serve as gatekeepers, managing access rights, which represent the fundamental permissions allocated to users and privileges, which encompass the authority to execute specific actions on objects. Their primary objective is that of preventing unauthorised users from utilising system resources while ensuring that authorised entities can make legitimate use of these resources [95]. Refined access control strategies are capable of granting access not only to specific resources but also to distinct segments within those resources. Within database systems, this level of granularity is achieved through vertical and horizontal partitioning. The former involves dividing a database table into smaller subsets of columns. This allows users to access specific columns within a database object based on their access rights. For example, in a customer database, vertical partitioning might enable certain users to access only the customer's name and contact information, while restricting access to sensitive financial data.

Horizontal partitioning, on the other hand, divides a database table into smaller subsets of rows. This facilitates users' access to specific rows within a database object according to their designated access privileges. For instance, in a sales database, horizontal partitioning might permit certain users to access sales data for specific regions or time periods, while preventing access to data from other regions or time periods.One of the prevalent methods for visualising access controls in databases is through the use of a CRUD matrix. This matrix serves as a structured framework for defining and managing permissions related to creating, reading, updating, and deleting data or resources within a system. It delineates the specific individuals or roles authorised to execute each of these actions on various objects, offering a well-defined and organised approach to oversee and enforce access permissions. L. Cavique and M. Cavique [96], elucidate that the CRUD matrix consists of a table featuring all database subjects vertically and objects horizontally. It assigns Create (C), Read (R), Update (U),

and Delete (D) permissions to each subject based on their access requirements to specific objects. Table 2.1 shows a CRUD matrix associated with an ordering system.

Table 2.1: CRUD Matrix for a simple ordering system

	Customer	**Order**	**Account**
Register Customer	C		
Create New Order		C	
Open Account			C
Update Customer Details	U		
Remove Customer	D		
Maintain Customer Order		U	
Cancel Order		D	
Modify Account Details			U
Terminate Customer Account			D
Generate Customer Report	R		
View Orders		R	
View Account Details			R

It can be noted that different processes exhibit distinct permission requisites. For instance, the 'Generate Customer Report' process, necessitates a SELECT (Read) privilege on the 'Customer' table whilst the 'Terminate Customer Account' requires a DELETE privilege on 'Account'. On the other hand, 'Create New Order' mandates a CREATE privilege on the 'Order,' table whilst 'Modify Account Details' involves an UPDATE privilege on 'Account'. The same logic can be applied to the remaining entries.

In addition to the conventional structures commonly associated with access control, such as the previously discussed CRUD matrix, scholars like J. Qiu et al. [95] offer further insight by categorising access controls into two distinct groups: traditional and enhanced. The former group refers to discretionary and mandatory access controls whilst the latter refers to role-based and attribute-based access controls.

2.9.1 Discretionary Access Control

In Discretionary Access Control (DAC), access to resources is determined by the subject's identity, which can be either the user or the group to which the user is affiliated (Figure 2.6a) [97], [98]. The typical method for implementing this access control model involves the use of access control lists (ACLs), access control matrices (ACMs), or capability lists (CLs). These lists or matrices consist of a compilation of users and groups,

along with the corresponding level of access. Users can manage permissions for objects that they own, as the data owner possesses the authority to approve or deny access to any object under their ownership.

Despite its capacity for fine-grained control over access permissions, this widely adopted and flexible access control model has its limitations. M. Wang [99] argues that in addition to being burdensome to manage because of the need to maintain access control matrices or lists, the process of granting and revoking permissions also requires ongoing maintenance. N. Kashmar et al. [100], further add that DAC possesses inherent vulnerabilities, such as the susceptibility to Trojan horse attacks.

2.9.2 Mandatory Access Control

Mandatory Access Control (MAC) is a structural model in which the determination of access to specific objects is made by a running automaton instead of the object owner. This model relies on security labels assigned to users and objects as its primary component. Objects are typically classified as ‘confidential’, ‘private’, or ‘public’, while users are categorised as ‘top secret’, ‘secret’, or ‘unclassified’ [101]. J. Qiu [95] elaborates that user access is determined based on the security level of both the user and the object they intend to read from or write to. The access control mechanism follows the "write up read down" protocol, granting read access to users when the object's security level is equal to or lower than theirs, and granting write access when the object's security level is equal to or higher [102] (Figure 2.6b). This model is often depicted as providing higher security but exhibiting less flexibility when compared to the DAC model. Additionally, it comes with limitations, notably a lack of fine-grained control, which can make it challenging to manage effectively.

2.9.3 Role-Based Access Control

Role-Based Access Control (RBAC) was devised in response to the need for an access control model that accommodates emerging computing paradigms particularly cloud computing and multi-tenancy [103]. RBAC is a straightforward model that relies on the association between users and roles to determine the permissions and privileges assigned to each role for every object. Administrators assign users roles based on the concept of least privilege, where users are granted the minimal set of permissions required to perform their tasks effectively (Figure 2.6c). According to N. Meghanathan

[104], this approach proves beneficial in cloud computing environments where roles are frequently created, modified, or customised to suit individual requirements. RBAC combines elements of both DAC and MAC, enabling flexibility while simplifying user management by assigning permissions to roles rather than individual users [105].

2.9.4 Attribute-Based Access Control

The Attribute-Based Access Control (ABAC) model enables the granting of access rights to individuals by utilising security policies that incorporate different attributes (Figure 2.6d). Typically, three attributes are defined in this model: subject, resource, and environment attributes. The subject attribute pertains to users, the resource attribute to objects or resources such as databases or servers, while the environment attributes encompass other factors that may impact overall performance [106].

2.9.5 Comparisons of Access Control Models

Figure 2.6, which has been referenced in the previous subsections, illustrates the distinction among the four primary types of access controls mentioned earlier.

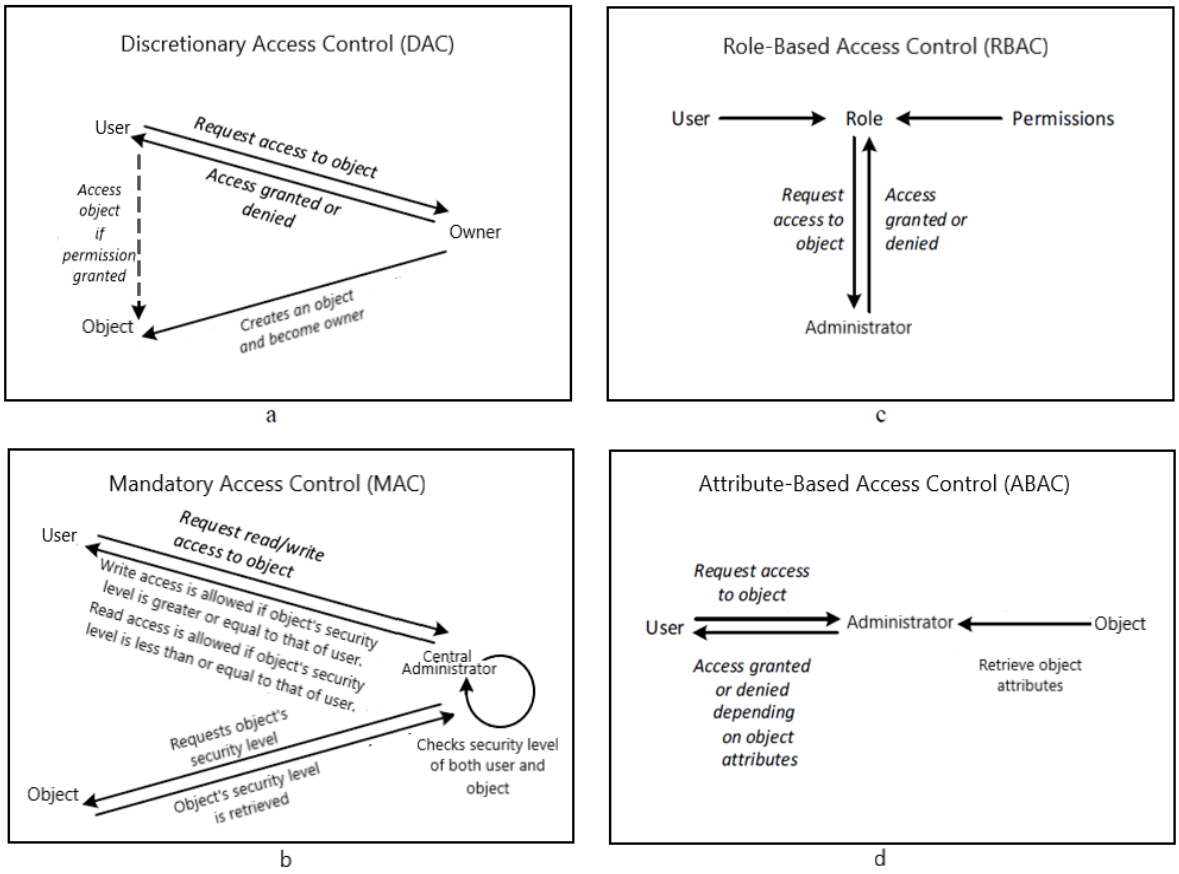


Figure 2.6: Access Control Types. Adopted from [107].

Several leading scholars have attempted to compare the aforementioned access control models. G. Pernul [108], introduced a comprehensive security framework to facilitate the comparison of such models. Initially, the author highlighted a prevalent issue regarding the lack of inclusion of these controls during the design phase in nearly all database systems. This emphasises the unfortunate perception that security is still regarded as a separate issue. The author also highlighted some drawbacks related to the DAC model. Firstly, this model places the onus of security enforcement solely on the tenant DBA. Moreover, its heavy reliance on subject-based access decisions renders DAC systems highly vulnerable to Trojan Horse Attacks. This introduces a potential vulnerability where a malicious user could exploit a trojan horse to gain the same access privileges and infect the system. In contrast, K. Almarhabi [109], highlighted the contrasting nature of MAC by emphasising that users are not burdened with the responsibility of implementing security policies and that Trojan Horse Attacks are limited due to the disallowance of write-downs.

S. Hasani and N. Modiri [110], attempted to identify ways of comparing the multiple access control models using two main criteria - qualitative and basic. The former criterion facilitates a comparative evaluation of models, guiding users to choose the one that best aligns with the specific needs of the organisation. The latter category, mentioned also by N. Zhang [111], establishes six criteria upon which all access models can be compared so that the most suitable model or combination of models is chosen to abate threats. The six criteria identified are:

1. **Administrative Requirements** which identify the type of policy utilised in the access control model i.e., whether centralised or not, easy to manage or not.
2. **Constraints** which refer to whether separation of duty and the least privilege concept can be implemented.
3. **Delegation** which indicates whether assignment of tasks is supported or not.
4. **Implementation Mechanisms** which might differ between different access control models. These include security labels, roles, access control matrix, access control list and capability list.
5. **Positive or Negative Authentication** which either specify those who are authorised to log in to the system or on the contrary specify those who are not allowed.

6. **Authorisation** which might be implemented differently in different access controls i.e., whether fine grained or not

In conclusion, it can be noted that the attainment of a safe and secure system hinges upon the implementation of the most suitable access control model tailored to the specific scenario at hand. This critical step can be achieved by thoroughly considering the range of measures presented in this section and making a decision based on that assessment.

2.9.6 Implementation of Access Control Models in DBMS

Access control models are indispensable components across several systems, including DBMS, which is the topic of interest throughout this dissertation. Understanding the implementation of these models is crucial for data security considerations; thus, this section delves into the typical implementation of each access control model within the DBMS domain.

DAC is a prevalent access control model commonly utilised in DBMSs. It operates on the principle of discretion, entrusting data owners or administrators with the authority to establish access permissions for individual database objects. The implementation process involves defining permissions, such as read, write, and execute actions, tailored specifically to tables, views, stored procedures, and other pertinent database elements. Subsequently, these permissions are assigned to users, determining who can interact with the database objects according to their designated rights.

RBAC, which builds upon DAC, diverges in its approach by granting or revoking permissions, via GRANT and REVOKE statements, not directly to individual users for specific objects but to roles, which represent groups of users. This streamlined approach simplifies access control management significantly. Implementation of RBAC includes defining roles, associating permissions with these roles, and assigning users to appropriate roles. To enhance RBAC implementation, many database systems, like PostgreSQL, offer the capability of implementing Row-Level Security (RLS) policies whereby horizontal partitioning is enabled to achieve a finer level of access control.

On the contrary, the implementation of Mandatory Access Control (MAC) in DBMS necessitates the categorising of data and users into security levels and establishing access policies based on these classifications. Many database systems incorporate specialised extensions to support MAC. For instance, PostgreSQL utilises

'sepgsql,' a loadable module that integrates label-based mandatory access control in accordance with pre-established security policies. Security labels, assigned through the 'SECURITY LABEL' statement, lay the groundwork for access control determinations in this approach. Access determinations depend on two key labels: one indicating the user's actions and the other denoting the object under examination. Both labels undergo evaluation, leading to either access authorisation or refusal based on the assessment's result.

Conclusively, albeit less commonly used than other access control methods, many DBMS platforms also support ABAC. ABAC implementation follows a systematic approach, commencing with the definition of attributes essential for access control policies. These attributes encompass a range of characteristics linked to users, objects, and the environment. Access control policies are then formulated, specifying access permissions based on attribute values. For instance, policies may dictate that individuals with a specific role or department can access data with particular classifications. These policies are defined and expressed as rules or conditions. The next step involves assigning attributes to users, objects, and the environment, based on their inherent properties. Subsequently, when a user requests access to a database object, the DBMS evaluates the request against the predefined policies, comparing the user's attributes to the conditions in the policies. Depending on the outcome of this evaluation, the DBMS grants or denies access, possibly requiring additional authentication or authorisation steps.

Each of the access control mechanisms discussed in this section is connected to overarching security models, which serve as the theoretical basis for shaping access control strategies. The subsequent section provides a thorough overview of the main security models, together with the access controls typically related to each model.

2.10 Security Models

Access controls are foundational elements in establishing robust security protocols within systems, functioning within the framework of diverse security models. Y. A. Younis et al. [112] suggest caution when choosing the appropriate model to address specific threats. This is especially critical in the context of multi-tenant cloud-

based systems, where multiple configurations and tenant requirements must be considered to ensure the integrity of the overall security architecture. Implementing a comprehensive access control framework not only safeguards sensitive data but also fosters trust among tenants, reinforcing security of the whole system.

N. Meghanathan [104] concurs and emphasises the pivotal role of selecting the most suitable model to effectively minimise risks and mitigate threats to the lowest conceivable level. He highlights the need for a thorough understanding of the system's intricacies and the potential vulnerabilities that may arise. By doing so, informed decisions regarding security model selection and implementation strategies can be carried out. Consequently, this section examines the four most prevalent security models, offering comprehensive insights into the ideal scenarios for the implementation of each model.

2.10.1 Bell – La Padula Model

This security model offers a mathematical representation of the implemented security policies. Since the 1980s, it has been recognised as the most robust model in terms of security. M. Cristia and G. Rossi [113], explain that the model governs users' access to objects through security labels associated with both the user and the object. These security labels typically range from highly sensitive ones, such as 'Top Secret', to public ones, like 'Unclassified'. According to G. D. Wurster et al. [114] an illustrative example of employing this model can be seen in a medical context, where certain sensitive medical information is classified as 'Top Secret' and can only be accessed by authorised medical professionals and patients. To determine whether access to an object is allowed or denied, the security labels are compared, and a decision is made based on three essential properties and their inference [110], [115]:

1. **Simple Security Property** – a subject at a particular security level can only access an object at a lower or equal security level.
2. **Star Property** – a subject at a particular security level can only write to an object with an equal or higher security level.
3. **Discretionary Security Property** – an access control matrix is utilised to define discretionary access control (explained in Section 2.9.1).

These properties serve to mitigate several of the prevalent DBMS threats outlined in Section 2.8.3, particularly those associated with misuse of authority and bypassing of controls. This is mainly due to the above-mentioned rules serving as a meticulously structured and tightly controlled approach for granting access. S. M. Hasani et al. [110], elaborate that the model commonly linked to this approach is Mandatory Access Control (MAC), which is known for its strong emphasis on the confidentiality aspect. However, the author concludes that this model does not adequately address the requirement of integrity.

2.10.2 BIBA Integrity Model

According to the pioneer of this model [116], the BIBA Integrity model ensures the secure transfer of information between resources with a high integrity level. This is accomplished by assigning integrity-level labels to objects and users, and adhering to the following three rules:

1. **No Read Down rule (Simple Integrity Property)** - A subject can only access data from an object with an equal or higher integrity level.
2. **No Write-Up rule (Star Integrity Property)** - A subject can only modify data in objects with an integrity level that is lower or equal to the object's current level.
3. **Invocation rule** - A subject is restricted from requesting access to resources with higher integrity levels.

M. Toapanta et al. [117], note that the first two rules of the BIBA model are the inverse of those used in the Bell-La Padula security model. In the BIBA model, the principle is described as "read up, write down," while the Bell-La Padula model follows the opposite principle of "read down, write up". This distinction arises from the fact that the BIBA model primarily focuses on ensuring integrity requirements, whereas the Bell-La Padula model is designed to fulfil confidentiality requirements. The Biba model proves invaluable when applied to healthcare systems, where preserving the integrity of patient data is key for ensuring patient safety and regulatory compliance.

2.10.3 Clark-Wilson Model

T. Tsegaye [118], elucidates that the Clark-Wilson model was developed after Biba's model and employs distinct strategies to safeguard information integrity. According to

the author, this model places emphasis on maintaining system-wide consistency, encompassing security, both before and after a transaction occurs. It is particularly effective in mitigating threats related to data tampering, unauthorised changes, and data corruption, which are all critical aspects of data integrity. This objective is achieved by adhering to the following three key principles:

1. A well-formed transaction consists of a series of operations that transition the system from one consistent state to another.
2. The integrity of each transaction is guaranteed through the inception of an integrity policy, which restricts access only to authorised personnel.
3. The principle of separation of duty is applied, wherein different users are assigned distinct roles based on specific requirements.

S. M. Hasani et al. [110], further add that the Clark-Wilson model also considers external information received by the system. To detect security breaches effectively, this model requires the implementation of a comprehensive set of measures, including robust audit trails, alongside reliable identification, authentication, and authorisation mechanisms. Access controls used in implementing this model should clearly define the authorised data access operations for each subject. According to the author, the Clark-Wilson model can be applied in banking systems, where safeguarding the integrity of financial transactions and deterring fraudulent activities is critical.

2.10.4 Chinese Wall Model

D. F. Brewer et al. [119], the developers of this model, clarify that this model addresses both the confidentiality and integrity requirements by incorporating elements of both mandatory and discretionary access controls. This model relies on various components, including users, objects, data, and labels, to effectively prevent any information flow between objects and subjects that could potentially give rise to conflicts of interest. By leveraging these components, and implementing the data isolation principle, this model ensures a secure environment free from conflicting information exchanges. F. K. Parast et al. [120], further add that the Chinese Wall Model expands upon the foundations of the Clark-Wilson Integrity Model to establish its security rules. The primary objective of this security model is to prohibit unauthorised users from accessing confidential data pertaining to other users. It is commonly utilised by accounting and consulting organisations to safeguard against situations where

consultants may encounter conflicts of interest by potentially accessing data belonging to competitors.

Prior to the utilisation of any of the security models mentioned throughout this section, it is essential to establish well-defined software requirements. Aligning security models with project-specific requirements enables organisations to proactively detect potential vulnerabilities and design solutions that accentuate data protection and threat mitigation, resulting in the creation of a robust and secure system.

2.11 Requirements

System development is rooted in the fulfilment of critical requirements. In the context of multi-tenant cloud-based systems and their security challenges, defining these requirements becomes intricate due to their ever-evolving nature. However, the process of gathering requirements should centre on answering three core questions: **what**, **why**, and **who**. Prior to the formulation of requirements, it's pivotal to establish the purpose driving their development. Furthermore, a clear grasp of the necessity for system development and the identification of key stakeholders are vital in ensuring a comprehensive requirement definition process.

M. A. da Silva et al. [121], shed light on the significance of requirements. They acknowledged the criticality of early-stage requirement elicitation, a notion which was concurred also by S. Lim et al. [122]. However, both acknowledged the lack of literature that clearly outlines a comprehensive set of requirements for secure systems. One relevant work in this regard is G. R. Tsochev and R. I. Trifonov's [123] research, which presents a structured methodology for extracting security requirements in cloud-based systems, focusing on external threats and suggesting ways to mitigate them. R. Kumar and R. Goyal [124] propose an alternative approach by incorporating common threats into case diagrams. Additionally, M. Irshad et al. [125] highlight the systematic literature review as a prominent method for requirement elicitation, offering a comprehensive overview of existing research to identify areas for improvement.

The main methodology utilised for requirements elicitation when utilising a systematic literature review is based on the guidelines of B. Kitchenham [126]. The phases involved when utilising this methodology for requirements elicitation are visible in Figure 2.7 and described hereunder:

Steps 1 and 2 involve defining the study's scope and rationale. In Step 3, relevant literature papers are identified and selected using predefined inclusion and exclusion criteria. Both snowball sampling and database searching are frequently used methodologies for assembling the initial set of literature works [127], with a slight advantage attributed to snowball sampling, as noted by Badampudi et al. [128].

The snowball sampling process is composed of the following steps [127]:

- Insertion of keywords related to the topic of interest as the search criteria in Google Scholar Semantic Scholar, IEEE Xplore and Academia.eu.
- Examination of the title of the literature works returned from the search.
- If the title is vague, the abstract is examined. In cases of inconclusive abstract examination, the conclusion is reviewed. In rare instances of further uncertainty, the entire literature paper is analysed.

Once the starting set of literature papers is established, Step 4 is executed where the suitability of the initial set is assessed using Wohlin's [127] recommended technique. This technique offers several recommendations to facilitate the selection process of candidate papers namely that selected papers should:

- Appertain to different authors.
- Include both recent and long-standing papers spanning a broad time frame.
- Contain findings and analysis related to the research area at hand.
- Include similar attempts at tackling the main research objectives although tackling distinctive niches or utilising different techniques.

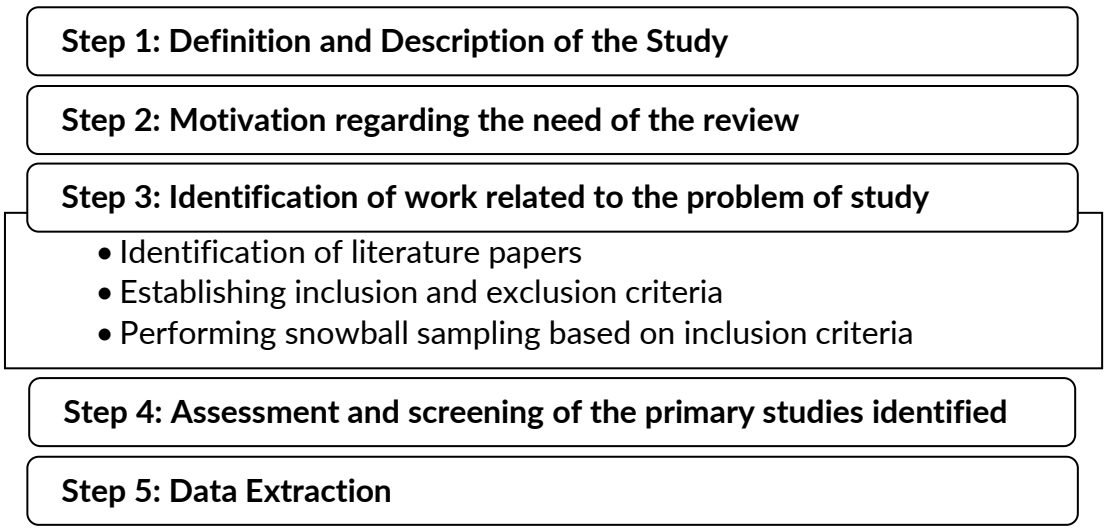


Figure 2.7: Methodology for systematic review. Adopted from [126].

2.12 Computer Aided Software Engineering (CASE) Tools

The landscape of emerging applications is characterised by a constant and progressive increase in complexity and variety. In response to these challenges, a pragmatic solution, known as Computer Aided Software Engineering (CASE) tools has emerged. These tools are carefully designed to make the software development process more efficient and less prone to human errors, playing a crucial role in automating tasks during the design, development, and implementation stages.

N. Yousaf et al. [129] point out that CASE tools can significantly assist in the domain of metaprogramming, amplifying their utility in software development. Metaprogramming, involves dynamically generating and manipulating code during runtime, a process that can be inherently intricate. In this context, CASE tools step in as invaluable aids, offering developers an intuitive and systematic interface for harnessing the potential of metaprogramming, while also enabling them to automate repetitive tasks and fine-tune code generation templates. This not only simplifies the complex process of metaprogramming but also introduces a level of flexibility and adaptability that is crucial in today's fast-paced software development landscape. By facilitating metaprogramming, CASE tools enable developers to make real-time modifications to class structures, create customised code templates, effortlessly adjust to changing project needs, and seamlessly adapt to evolving project requirements. They act as a bridge between the developer's intent and the dynamic world of metaprogramming, making it more accessible and efficient for achieving enhanced productivity and code quality [129].

S. S. Patel [130] advocates the adoption of such tools, citing their numerous advantages. They assert that by automating diverse tasks, these tools not only diminish errors stemming from human involvement but also save time and money while enhancing overall quality. Additionally, the standardised presentation of information generated by these tools enhances communication and streamlines the entire workflow. Another notable advantage lies in the insulation from intricate implementation details, allowing developers to focus on higher-level design and functionality without getting burdened by the nitty-gritty of coding. Additionally, S. S. Patel [130] discusses best practices linked to their utilisation. These practices include a thorough grasp of project objectives and requirements, underlining the pivotal role of selecting a tool

aligned with project needs and team expertise, alongside the implementation of standardised processes to promote consistency and enhance code quality.

2.12.1 Core Functions of CASE Tools

Figure 2.8 offers a schematic representation of a CASE tool, providing a concise overview of its fundamental components and core functions.

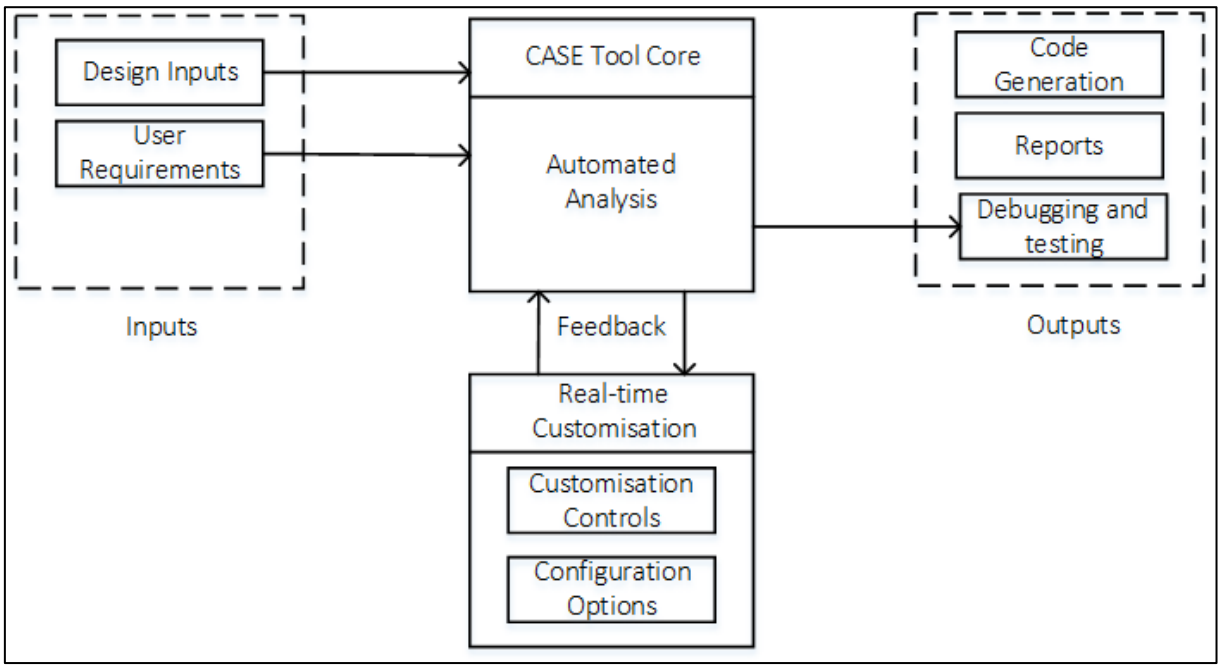


Figure 2.8: Schematic Representation of CASE Tool

The CASE tool's inputs encompass user requirements and design data, with its core functionality centred around automated analysis capabilities. In addition to these inputs, developers can perform real-time customisations, allowing them to fine-tune and adapt the tool's output to meet specific project needs. Lastly, the outputs generated by the tool yield tangible results, including code generation and the identification of potential error detection.

[130][130]Several studies including [129] and [131] were conducted to compare software production with and without CASE tool integration. The findings indicate that employing CASE tools results in fewer errors during the analysis and design phases of system development. Furthermore, these tools prove advantageous during testing and maintenance, primarily due to the simplified modifications available during the design phase, as opposed to post-implementation alterations

While CASE tools offer various advantages, they also come with some drawbacks, as noted by G. Dias [131]. One notable challenge is determining the extent to which requirements should be incorporated into the design diagrams. If certain requirements are not encoded, it necessitates ad hoc programming alongside the generation of CASE tool outputs. Additionally, the CASE tool should be capable of interpreting design diagrams and producing suitable programming code. The author concludes that excessively intricate diagrams, when interpreted by the CASE tool, can result in an environment that is impractical and inflexible. Conversely, diagrams that necessitate additional coding contradict the purpose of using the CASE tool. Nevertheless, the advantages highlighted earlier far outweigh these potential drawbacks. In fact, CASE tools continue to maintain momentum, particularly as they aid organisations in gaining a competitive edge [129].

2.12.2 Security-Oriented CASE Tools for Database Systems

Since CASE tools are utilised to support automated application development, they have also proven to be beneficial in the development of data-centric information systems [129]. Consideration and integration of security requirements during the initial design phase is fundamental. Employing database design diagrams during the early stages can be instrumental in implementing security measures. Three primary diagrams, namely the Entity Relationship Diagram (ERM), Entity Life History (ELH), and Data Flow Diagram (DFD), offer a comprehensive perspective of the application system. [132]

ERMs provide a representation of the entities within a system and the relationships between them. They are widely used in database design to model the structure of a database and the connections between different entities. They aid in understanding the logical structure of data and identifying key relationships that exist within the system.

ELHs extend the concept of ERMs by incorporating the life cycle of entities and the events that affect them over time. Such diagrams are valuable for capturing temporal aspects of a system and depicting how entities evolve and interact with each other throughout their life cycle. This perspective enhances the understanding of system dynamics and facilitates the identification of security vulnerabilities that may arise at different stages of entity life cycles.

DFDs provide a graphical representation of the flow of data within a system. They depict the processes that manipulate data, the data stores where information is stored, and the data flows between them. These help in understanding the flow of information within a system and identifying potential points of vulnerability where security measures need to be implemented.

Thus, by comprehensively integrating access rights and privileges into these three diagrams, security considerations are seamlessly integrated into the system design, eliminating the necessity for separate post-design security assessments [132].

Several CASE tools have been created to aid in the database design procedure. One such tool is the Artiso Visual CASE tool, designed to support software development, database design, and maintenance [133]. It enables users to employ data flow and entity relationship diagrams, streamlining the automation of new software development. Alternatively, MetaEdit+ is a tool that can be employed to design modelling languages and generate diagrams [134]. Another valuable CASE tool tailored specifically for database systems is DB-Main [135]. The primary objective of this data modelling tool is to assist developers and database administrators (DBAs) throughout the system design process. This includes aiding in requirements elicitation, conceptual and physical design, as well as schema modelling.

Despite a widely acknowledged consensus in both the research community and industry that security requirements should be seamlessly integrated with functional requirements from the earliest stages of design, the work of H. Mouratidis et al. [136] highlights a significant gap in existing literature. There is a lack of references that identify CASE tools intentionally developed to integrate security requirements into design diagrams such as ERM, ELH, and DFD. This is regrettable, as these diagrams not only define the overall system but also are capable of uncovering potential bottlenecks. Such diagrams are capable of bridging the gap between system design and data analysis, rendering them indispensable for effectively addressing the challenge of incorporating security requirements into design diagrams.

Consequently, further exploration in the field of CASE tools that integrate security requirements right from the initial design stages, generating full system code while highlighting vulnerabilities, is essential. This endeavour plays a critical role in

minimising conflicts between security and other functional requirements, ultimately leading to the development of more secure and comprehensive systems [129].

2.13 Related Work

As highlighted in the earlier sections of this chapter, security emerges as a paramount concern that hinders the adoption of multi-tenant systems. Although numerous authors, including [137], [138] and [139], have conducted extensive research in this domain, addressing security in multi-tenant cloud systems remains an enduring challenge. Most research papers tend to concentrate on specific facets of security within this context. For instance, some papers such as [140] and [141] direct their attention on access control mechanisms, related to the multi-tenant cloud landscape. Others focus their efforts on evaluating data separation [142] and encryption algorithms [143], to bolster data security in multi-tenant cloud databases. On the other hand, separate researchers such as [144] and [145], focus on advanced monitoring and auditing tools tailored for multi-tenant cloud environments. These examples illustrate the diverse facets of security that researchers address, each contributing to the overarching goal of enhancing security in multi-tenant cloud systems.

However, a significant gap exists in research regarding the development of CASE tools that systematically incorporate security considerations from the initial stages of database design. Treating security as a standalone concern and attempting to introduce security measures at a later stage can lead to challenges, as the database may not have been originally designed to accommodate these measures. Therefore, the implementation of a unified automated system that integrates a comprehensive set of security requirements through design diagrams during the initial stages of database design can effectively reduce the risk of security breaches.

Despite the availability of numerous database design CASE tools that assist system designers during the design phases, as mentioned in Section 2.12, there is a notable scarcity of frameworks that incorporate security modelling into these design platforms. Pioneering work in this domain was undertaken by Guili and Iglio [146], who expanded ERMs with role-based access controls to facilitate the development of secure database designs. The proposed tool incorporated an analysis algorithm capa-

ble of identifying potential security design flaws and a translation procedure that automatically generated SQL specifications aligned with the conceptual model. However, certain aspects of the database system fell outside the scope of their work. These aspects encompass elements like transition mechanisms and process utilisation, which are typically depicted in alternative modelling diagrams such as ELHs and DFDs. Their tool primarily concentrated on security concerns related to the structural and access control facets of database design, without delving into the intricacies of data flow between processes or the execution of specific actions within the system. These additional factors are pivotal for a comprehensive approach to addressing database system security, leaving room for potential future research and tool development in these areas.

In a related study entitled "A Pattern-Based Approach for Secure Database Design," [147], the authors introduce an innovative method aimed at assisting database designers in creating schema designs that align with organisational security policies, particularly those related to authorisation. This approach diverges from the previous work since database design diagrams are not used. Instead, it relies on the early integration of security policies [146]. The process begins by defining organisational security policies in the form of security patterns. These patterns serve as guiding principles during application development, ensuring the proper implementation of security requirements. It then concludes by enabling the automated generation of a secure database schema. However, it is worth noting that the primary emphasis of this research is directed specifically at the 'authorisation' aspect of security. Thus, the main objective is to address and enhance the mechanisms related to granting or denying access to resources within the system, rather than providing an all-encompassing or holistic security solution that covers multiple security concerns. Similarly, Wazid et al's paper entitled "Design of Secure Key Management and User Authentication Scheme for Fog Computing Services" [148], follows a comparable path. It also focuses on one security aspect, specifically authentication mechanisms without taking a holistic approach.

It can be noted that the studies discussed throughout this section, primarily centre around the design of secure database systems, either locally or within a cloud-based context. To the best of our knowledge, no existing papers were identified that

delve into security considerations from the outset of multi-tenant system development. This absence underscores the novelty and significance of the proposed work, which seeks to address this substantial gap in current knowledge.

2.14 Summary

This chapter lays the foundation for a comprehensive understanding of the key concepts explored in this dissertation. It serves as a comprehensive foundation, offering an in-depth exploration and elucidation of key concepts relevant to the dissertation's subject matter. It encompasses a detailed background discussion on various topics, including cloud computing, multi-tenancy, access control mechanisms and database systems, along with a focus on security considerations within these domains. Furthermore, a review of relevant literature closely aligned with the study's approach is also included in this chapter, providing valuable insights for the research conducted.

3 Requirements and Analysis

3.1 Introduction

In this chapter, Kitchenham's method [126] is employed to compile a cohesive list of security requirements. Eliciting requirements prior to the initiation of the design and development of software products, including the one proposed for this dissertation, is of utmost importance. This list lays the essential foundation upon which the framework developed in this dissertation, targeted towards software houses implementing multi-tenant cloud-based solutions, is constructed. Apart from delineating the fundamental requirements that underlie the proposed tool, this chapter also emphasises the pivotal techniques employed throughout this dissertation to achieve the specified requirements. The rationale behind the selection of these techniques is substantiated.

The findings from this chapter were presented at the International Conference on Network Security and Cloud Computing and published in the paper titled "Specification of Requirements to Ensure Proper Implementation of Security Policies in Cloud-Based Multi-Tenant Systems" [149].

3.2 Requirements Elicitation

As stated in Section 3.1, prior to the design and construction of the framework developed throughout this dissertation, it's crucial to pinpoint the foundational requirements that will guide its development. While striving for a comprehensive compilation of requirements is optimal, it's important to acknowledge that security is a non-bullet proof area. Nevertheless the ultimate goal should be to support software houses in targeting and achieving the highest possible level of security.

The methodology employed for gathering requirements is grounded in a systematic review of existing literature, following the guidelines established by B. Kitchenham [126]. The deliberate choice of this methodology stems from its suitability in gaining insights into the current state of the field, as well as enabling the identification and subsequent addressing of gaps present in the literature. The phases encompassed by this methodology for requirements elicitation are outlined and explained in Section

2.11.1. The subsequent paragraphs delineate these steps with respect to the tool developed throughout this thesis.

It is worth noting that, in addition to the guidelines established by B. Kitchenham [126], two extra sub-steps were incorporated into Step 3 to accommodate the distinctive nature of this research. The additional sub-steps are:

- Analysis of security features in current cloud software.
- Analysis of existing threats and risks.

3.2.1 Definition of study and motivation

The initial two stages of the methodology, as outlined in Figure 2.7, encompass defining and describing the study, while also establishing the motivation for conducting the review. These steps are closely aligned with addressing the essential questions in requirement elicitation, namely, the **what**, **why**, and **who**. Prior to the process of gathering and formulating requirements, it is imperative to delineate the purpose for which these requirements are being developed. Additionally, it's important to ascertain the necessity behind system development, to ensure that any needs the system should fulfil are identified whilst identifying the individuals or entities who will either be involved in or impacted by the proposed system (key stakeholders).

What: Throughout this dissertation, the focus is on developing a framework for software houses to be able to provide security multi-tenant solution for cloud-based solutions. This in fact, represents the central aim of the dissertation, outlined in Section 1.3, which emerged from a thorough examination of existing literature and the revelation of a research gap discussed in Chapter 2.

Consequently, the purpose of conducting this systematic review is the purpose of conducting this systematic review is to identify various security requirements relevant to multi-tenant based CDBMS. Although the primary emphasis is placed on security-related CDBMS requirements, it is equally imperative to consider security measures concerning the application program itself, Cloud Service Providers (CSP), multi-tenancy, and Software as a Service (SaaS). This all-encompassing approach guarantees the establishment of a secure and comprehensive system.

Why: The ‘why’ should be grounded in addressing the ‘what’, which, in this context, pertains to the development of a secure multi-tenant system for cloud-based solutions. This necessity stems from the growing demand for such systems due to the

array of benefits presented by multi-tenant cloud-based solutions. However, this demand is tempered by reservations stemming from security concerns.

Developing a multi-tenant cloud-based solution is indispensable due to its multitude of advantages, outlined in Section 2.7, including enhanced resource efficiency, scalability, and cost savings. By enabling multiple users or tenants to utilise the same infrastructure, this approach optimises resource consumption and reduces operational expenses. The inherent scalability of this architectural model ensures adaptable resource allocation to cater to varying demands whilst streamlining essential tasks such as updates and maintenance. This approach aligns with the modern emphasis on agility, innovation, and effective resource utilisation, making it a valuable strategy for delivering services in a dynamic business landscape. Nonetheless, a certain level of hesitance persists in fully adopting this approach, primarily due to its associated security risks. Thus, the development of a framework which targets this domain is important.

Who: The intended recipients of the proposed framework, are software houses and database administrators (DBAs). These entities require systems, capable of creating secure multi-tenant solutions, which can be customised to address the unique requirements of individual tenants.

3.2.2 Identification of work related to the problem of the study

As emphasised by C. Wohlin [127], literature constitutes the primary avenue for conducting a systematic review. Nevertheless, given the dynamic nature of advancements in the multi-tenant CDBMS domain, two additional sources have been integrated into the study. As depicted in Figure 3.1, the elicitation of security requirements relies on two key sources:

1. **Academic papers**

   This category encompasses scientific publications addressing security requirements in multi-tenant cloud-based databases, along with those highlighting prevalent threats within the domain. Papers discussing threats were particularly valuable for the formulation of requirements to counteract or mitigate such risks.

2. **Documentation of prominent cloud computing products**

   Beyond literature work, the study extends to incorporate existing software products capable of delivering cloud-based multi-tenant solutions. By analysing their security features, valuable insights into security requirements can be derived.

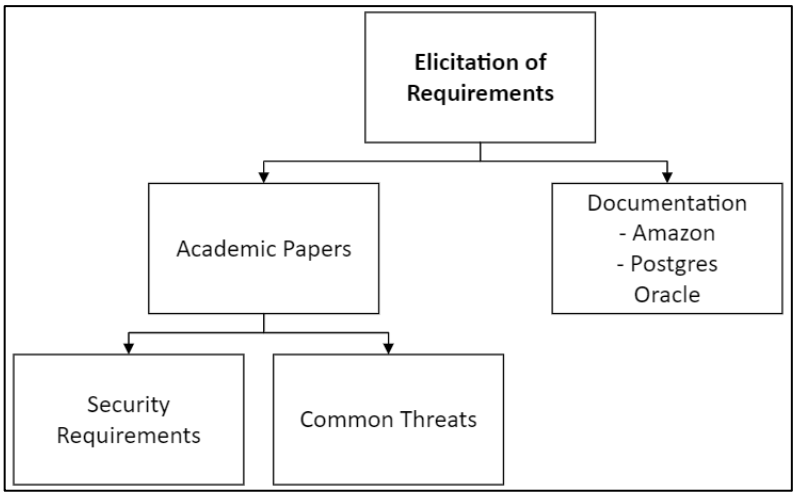


Figure 3.1: Identification of resources for requirement elicitation

The sequence of activities involved in identifying and selecting relevant research is documented hereunder:

3.2.2.1 Identification of academic papers

1. **Security Requirements**

Several works from Google Scholar, IEEE Xplore, Academia.eu and Springer Scholar were identified as the initial set of literature papers from which requirements are to be extracted. Before the gathering of information, inclusion and exclusion criteria were established so that only the most vital sources concerning the topic were retained. The sources which were included for this dissertation were those which specifically discussed requirements elicitation in cloud-based multi-tenant database systems. Exclusions encompassed materials that exhibited vagueness, centred exclusively on a particular scenario, or concentrated on a limited number of requirements. With such criteria in place, the snowball sampling technique was executed. Subsequently, the initial collection of papers was refined, retaining those deemed most fitting according to the suitability method established within the same section.

In the end, the following ten papers were selected [12], [75], [137], [150]-[156]. Although the selection process was conducted in a very rigorous manner, it was observed that most of the papers either tackle one specific example of cloud-based systems or produce a limited number of requirements. For example, A.Elmonem et al. [157], discussed the elicitation of cloud security requirements solely concerning ERP systems. Thus, it was clear that a cohesive set of general security requirements applicable to diverse multi-tenant cloud-based scenarios is lacking.

2. **Identification of existent threats and risks**

The review was also utilised to identify the most common security threats and risks so that mitigation or prevention requirements can be formulated. A similar procedure was conducted in this section and the literature works considered were namely [40], [80], [81], [84], [86], [89], [90], [93], [117], [158]. The works included were those dealing with threats concerning data at rest (i.e., at a database system level), particularly those related to multi-tenant CDBMSs. The suitability mechanism, established in the prior paragraph was once again utilised to filter the literature works. Through analysis revealed that among the frequently cited threats in the literature are SQL Injection attacks, Denial of Service (DoS) incidents, Information Disclosure and Trojans, as well as assaults from both external and internal entities, among others.

3.2.2.2 Analysis of security features in current cloud software

An examination was also conducted on prevalent cloud-based multi-tenant products to evaluate the security attributes they incorporate. PostgreSQL, Amazon, and Oracle [159]-[161], are the most frequently mentioned software in literature when considering multi-tenant CDBMSs. The primary security features of each software are listed in Table 3.1 hereunder. These features are more comprehensively defined and elaborated upon in the Requirements Analysis section (Section 3.3).

Table 3.1: Comparison of Security Features in major CDBMSs

PostgreSQL (version 15) [159]	**Amazon AWS** [160]	**Oracle Database 21c** [161]
Data Encryption	Data Encryption	Data Encryption
Authentication	User Authentication	User Authentication
Access Controls	Access Controls	Access Controls
Security Label	DoS Attack Mitigation	Password Protection
Superuser account with specific privileges	Recording of Data Configuration	Audit by Session
Allowing connection between clients to take place only if sockets are present	Logging of data alteration activities	Automatic Secure Configurations

3.2.3 Assessment and screening of the primary studies identified

Upon concluding the aforementioned stages, the gathered information was analysed and refined. Certain materials were excluded, leaving only those with significant points aligned with the predefined objective. The retained scholarly works were chosen based on their popularity, the principal inquiries posed and outcomes achieved.

3.2.4 Data Extraction

The final phase involves generating a conclusive outcome, namely a coherent collection of crucial security prerequisites for multi-tenant cloud database systems. This information is extracted from the set or relevant works identified in previous works. It was noted that multiple literature works exist which mention requirements elicitation concerning cloud database systems. Nonetheless, most of them either target one specific scenario or contain very few requirements. Moreover, a cohesive list of requirements which takes into consideration literature work, current systems security implementations and mitigation to threats were not found. The requirements elicited from both techniques can be divided into three main categories: *Base*, *Functional* and *Non-Functional* requirements.

1. **Base Requirements**

Among the key requirements highlighted in various literature sources, a notable emphasis was placed on employing suitable *design diagrams* capable of comprehensively representing multi-tenant systems. These diagrams serve as a robust foundation for implementing secure policies that encompass the entirety of such setups. Furthermore, another essential requirement consistently highlighted across the literature works under consideration pertained to the integration of *access controls*. These should not only be well in place, but it should also be ensured that the *current state is kept* once access controls are assigned. Furthermore, [75] and [151] emphasised the fact that *reviewing of code utilised in DBMS* should also be considered as a basic security requirement. This is because, code review does not only ensure that all specified requirements are implemented, but it also guarantees that best practices are utilised to avoid possible threats and risks.

Brown et al. [75], stated that the utilisation of *open standards* is another aspect which should be considered as fundamental. The use of openly available specifications

for the accomplishment of particular jobs diminishes the risk of vulnerabilities since the code would have been scrutinised by multiple individuals. The last base requirement extracted from literature was related to the utilisation of *parameterisation, configuration and additional security requirements* [75], [137]. Although DBMSs have inherent security policies, clients should be able to augment their security profile by including further security configurations.

2. **Functional Requirements**

Confidentiality, *Integrity* and *Availability*, commonly known as the CIA triad, constitute three essential security prerequisites that demand meticulous fulfilment in any given system. Virtually, all literature works scrutinised, allude in varying ways, to these three foundational security principles. Apart from the prior three requirements; M. Ragi [155] stated that *non-repudiation* is a vital security requirement. Non-repudiation refers to eliminating the risk of having untraceable operations. This notion was strengthened by, multiple authors such as [80], [137], [147], [151], [152], who emphasised the importance of having suitable *attack prevention, detection and mitigation procedures* together with adequate *auditing and accountability for security breaches*.

3. **Non-Functional Requirements**

Following the identification of the base and functional requirements, other non-functional criteria which might directly impact security were highlighted. One such consideration is *performance* mentioned by P. M. Hoener [153] and Liu et al. [156]. It should be ensured that each of the applied security techniques, effect minimally the overall application performance.

Furthermore, *tenant isolation* was another non-functional requirement brought forward by [80], [137], [147]. Although tenants would have moved their data from local storage onto multi-tenant cloud database systems, it should still be ensured that they are the sole owners of their data. Conclusively, [80] and [137], consider *complexity* as a non-functional requirement. Although, at first glance, complexity might seem a distinct concept from security policies and procedures, a good balance needs to be established between the inclusion of security requirements and the impact of this on the overall complexity of the system management. Security procedures must be implemented in the simplest way possible to minimise any possible disruptions in the overall functioning of the system.

3.3 Requirements Analysis

After identifying the requirements, outlined in Section 3.2, a thorough analysis was undertaken to ascertain the primary techniques that the proposed framework will employ in order to meet the specified requirements. This analysis encompassed an evaluation of the diverse alternatives at hand, and the selection and justification of the most suitable approach. The primary aim behind this analysis was to not only identify the fitting solutions for the established requirements but also to provide a thorough rationale for the chosen course of action.

3.3.1 System Design Diagrams

A pivotal requirement that significantly influences the entire process is the accurate depiction of a system in its entirety. This entails taking into account various angles including functionality, structure, and evolution over time [162], [163]. Employing diagrams such as the ERM, ELH, and DFD is an effective approach to meet these criteria, allowing for a holistic representation that encompasses all three previously mentioned elements. A comprehensive analysis in the subsequent paragraphs illustrates the primary strengths and weaknesses of each diagram, while also drawing comparisons with alternative methodologies that may have been utilised.

ERMs offer advantages in representing complex data scenarios with simplicity and can be easily transformed into relational database systems, which is beneficial for our tool. Notwithstanding such advantages, the proposed tool may not be suitable for software developers working with advanced database concepts like aggregation, composition, specialisation, or generalisation since these require a modified version of ERM. ELHs have the strength of selectively reflecting database system modifications without affecting other ELHs. Together with DFDs, they are capable of validating system requirements and retrieving business rules, but do not model time-dependent behaviour or triggers which might affect the system's primary course of action.

An alternative to the mentioned ERM, ELH, and DFD approaches is the utilisation of Unified Modelling Language (UML) diagrams [164]. UML can be adapted for database design, with the Class Diagram serving as a substitute for ERM. However, these are focused on classes and associations in a software program, while ERM diagrams depict entities and connections in a database. Given the objective of this

dissertation to build a CASE tool for database systems, ERM diagrams are more suitable. UML sequence diagrams are a suitable alternative to ELH diagrams, focusing on process execution order whilst use case diagrams in UML are comparable to DFDs but lack a holistic overview and do not include data stores [165]. Therefore, DFDs are considered a more suitable formal model for the system, while Use Case diagrams may be useful as initial drafts during analyst and client discussions.

3.3.2 Database Engine Techniques for Meeting Requirements

The selection of the DBMS for implementing the proposed framework was based on a rigorous analysis of how each requirement outlined in Section 3.2 could be effectively realised. Taking this into account, an implementation bias was adopted since PostgreSQL version 15 [159] was used as the central database engine, supported by SQL and PL/pgSQL for script development. Considering other DBMS alternatives as well, particularly those indicated in Table 3.1, the rationale for favouring PostgreSQL is explained hereunder.

One primary factor in its selection is its increasing prominence within the cloud computing sector, being consistently utilised by industry leaders such as Microsoft, and other competitors [32]. This is mostly attributable to its numerous benefits including rapid transfer of local applications to a cloud-based ones, straightforward setup and management of database systems as well as scalability and availability which are two very important characteristics of cloud systems. Moreover, it has the advantage of being an open source project which can be downloaded freely from its website with a maximum table size of 32TB and a maximum row size 1.6TB [159] .

Apart from the prior-mentioned benefits, PostgreSQL possesses the capability to function as a CDBMS, with a multitude of built-in features designed to enhance security measures. Some of these features, closely align with and impact the implementation of a majority of the requirements discussed in Section 3.2 and are analysed in the subsequent paragraphs:

One of the specified base requirements was related to the review of DBMS code to address various security threats. This involves thorough evaluations of the code's structure to ensure its ability to handle security risks like data leaks, improper configurations and vulnerability to SQL injection attacks. In PostgreSQL, specific tech-

niques such as using the ‘format’ options (%L or %s), or parameterised queries can effectively mitigate these threats, making them key points to focus on during code reviews [166]. Another notable aspect involved embracing open standards. Despite opting for PostgreSQL in the proposed tool, the code remains highly adaptable, primarily due to its alignment with SQL standard code. This inherent compatibility facilitates effortless modifications and utilisation across diverse cloud-based database management systems and various development environments.

When analysing the selected database engine in relation to the functional requirements, it's essential to emphasise that PostgreSQL encompasses a variety of features which can be used to ensure confidentiality, integrity and availability. Several authentication mechanisms can be employed including passwords, unique security codes, mobile authentication, or biometrics. For the purposes of this dissertation, the chosen method was password authentication, as it stands as the most prevalent approach [167]. However, should tenants prefer alternative authentication methods, the related scripts can easily be adjusted accordingly. The other DBMSs under consideration also provide authentication mechanisms. However, PostgreSQL stands out by offering the most extensive variety [168].

Within the scope of confidentiality, authorisation is another pivotal factor that can be implemented via all database engines under consideration. The implementation mechanism differs across PostgreSQL, Amazon AWS, and Oracle. PostgreSQL employs a role-based model that enables the association of roles with users and the assignment of privileges to these roles [159]. On the other hand, Amazon AWS offers database instance authorisation through the management of security groups and database user accounts. Lastly, Oracle employs role-based and privilege-based models, encompassing roles, together with system privileges and object privileges [169]. In line with the discussion in the following section, the framework's use of role-based access control mechanisms makes PostgreSQL a suitable candidate.

Moreover, PostgreSQL also includes encryption mechanisms, a significant element contributing to data integrity. Encryption can be applied to both passwords and other columns using 'pgcrypto' [170], an integrated PostgreSQL extension that provides symmetric and asymmetric encryption, hash functions, and digital signatures.

The selection of the most appropriate type of encryption for a specific system is dependent on multiple factors. These include data sensitivity, required level of security, performance considerations, and other regulatory compliance constraints to name a few. Comparable to PostgreSQL, encryption possibilities for data in transit and at rest are present in both Amazon AWS [160] and Oracle [171].

Other requirements specified in Section 3.2, which can be addressed effectively in PostgreSQL, are integrity and availability. Notably, an important aspect of these requirements involves the implementation of redundancy measures to mitigate data losses. PostgreSQL provides three primary modes of data backup which can be used when backing up database systems or schemas: SQL dump, file system-level backup, and continuous archiving [172]. The choice of backup mode depends on the specific needs and requirements of the user. However, a constraint regarding backups arises due to their exclusivity to separate databases and separate schemas in multi-tenant approaches. This limitation becomes particularly evident when addressing multi-tenant scenarios that involve the same table. In such instances, a tailored procedure needs to be crafted to address this gap. Similar issues persist in both Oracle and Amazon AWS albeit offering different backup procedures. Amazon AWS offers the possibility of Backup-as-a-Service [173], whereas Oracle offers Recovery Manager and user-managed backup and recovery options, capable of safeguarding the entire database, tablespaces, or all data files, yet the core problem remains unresolved [174].

The final, yet equally crucial, functional requirement revolves around maintaining audit trails to guarantee the absence of untraceable events. In PostgreSQL, this need can be fulfilled by establishing dedicated tables for audit purposes, coupled with the option of triggers. Oracle, on the other hand, offers auditing capabilities targeted at individual pluggable databases or the multitenant container, determined by the specific policy, achieved through the CREATE AUDIT POLICY and AUDIT statements. Each pluggable database, including the root, maintains its own unified audit trail. In the case of Amazon AWS, the AWS Audit Manager provides a sophisticated solution for this purpose, albeit with associated costs [175].

Table 3.2 summarises the security features offered by PostgreSQL, AWS, and Oracle, as identified in this section. It also evaluates their alignment with some of the established requirements outlined in the thesis

Table 3.2: Comparison of Security Features and Requirements Alignment

Requirements	PostgreSQL	AWS	Oracle
Review of code to address security threats	Format options and parameterised queries	Parameterised queries	Parameterised queries
Authentication	Offers the widest range of authentication mechanisms, including passwords and unique codes	Provides several authentication mechanisms but fewer than those offered by PostgreSQL	Offers various authentication mechanisms but with a fewer range when compared to PostgreSQL
Confidentiality	Implements role-based authorisation	Provides database instance authorisation through security groups and user accounts	Incorporates role-based and privilege-based models
Integrity	Supports encryption for data in transit and at rest	Supports encryption for data in transit and at rest	Implements encryption for data in transit and at rest
Availability	Features three redundancy options - SQL Dump, File System Backup, and Continuous Archiving	Offers Backup-as-a-Service	Provides the possibility of a Recovery Manager and user-managed backup
Monitoring audit trails	Utilises dedicated audit tables and triggers	Utilises AWS Audit Manager	Implements audit policies and ensures each pluggable database (even the root) maintains its own audit trail

PostgreSQL is also capable of implementing and assessing the non-functional requirements highlighted. One key consideration is finding the right equilibrium between security measures and their impact on the framework's performance. PostgreSQL allows performance assessment through accessible benchmarks like ‘pg_bench’ [176]. This is optimal for evaluating the performance of the proposed framework, taking into account the presence or absence of implemented security features. Likewise, performance benchmarks like 'Amazon CloudWatch metrics' for Amazon AWS [177], and Oracle's dedicated extensions such as 'Database Performance Analyzer for Oracle' [178], are accessible within their respective ecosystems. Complexity is another non-functional requirement, and in this regard, PostgreSQL aligns well as it effectively addresses this aspect. The code implementation for security features like Row-Level Security (RLS) policies in PostgreSQL is straightforward and devoid of excessive complexity.

3.3.3 Access Control

As evident from Section 3.2, a paramount security requirement for multi-tenant cloud-based solutions, consistently emphasised across all reviewed papers, is the implementation of robust access control mechanisms.

The role-based model was selected as the principal approach for the proposed tool. This decision was based primarily on the fact that RBAC is the most prevalent access control mechanism used in cloud-based systems [182]. Albeit similar in nature to discretionary access control (DAC), RBAC streamlines access management in intricate cloud databases by assigning permissions to roles rather than users. Thus, management of large user groups with comparable rights are easier to handle and inconsistencies are reduced. Furthermore, this mechanism aligns well with the dynamic nature of the cloud, where constant changes in resources and users should be reflected in access control policies.

Moreover, the decision to opt for RBAC was also influenced by the manifold benefits associated with it, particularly in relation to the least privilege and separation of duties security concepts. The former concept is adhered to, since roles are only granted access to objects which are mandatory for them to be able to perform their job. On the other hand, the latter is ensured via the assignment of distinctive roles to distinct users, each having its own permission set [52]. Users are assigned roles based

on their job functions or responsibilities, and their access to resources is determined by the permissions associated with those roles.

The RBAC approach provides a better alternative to DAC and MAC, particularly in multi-tenant cloud-based systems. This is mostly attributable to its capability to overcome some of its primary limitations. In DAC, users can grant or revoke privileges on any objects they own. Despite providing a certain level of flexibility, this approach might augment security risks if some of the users are inept in taking informed access control decisions. In contrast, MAC restricts access based on rigid security labels which might be difficult to administer in dynamic environments such as the cloud. RBAC provides a balanced alternative by mitigating security issues arising from DAC's flexibility while still being flexible enough to overcome MAC's constraints of inadaptability and unscalability. In conclusion, based on the conducted analysis and the extracted features from Section 3.3, PostgreSQL has emerged as the favoured option for implementation of the proposed framework.

3.4 Summary

In this section, a cohesive list of security requirements was elicited, establishing the foundation on which the proposed framework, aimed at providing secure multi-tenant cloud-based solutions, was constructed. This endeavour was undertaken based on the recognition that identifying requirements serves as the initial and fundamental step in any system's development. Furthermore, it was noted that the existing literature lacks a comprehensive compilation of security requirements for such systems. Following this, an in-depth analysis was performed to identify the techniques capable of fulfilling the requirements among a range of available options, substantiating the rationale for selecting specific approaches.

4 Design overview and system diagrams

4.1 Introduction

The design phase represents a crucial stage in the development of any software tool or product as it serves as the bridge between the research's theoretical framework and its real-world implementation. During this phase, systematic planning and construction of the framework are undertaken to address the research questions and attain the defined objectives.

This section outlines the core components involved in the design of the proposed CASE tool, targeted towards the automated generation of a secure multi-tenant, cloud-centric database system. It provides a comprehensive outline of the entire CASE tool construction process but centres its attention on the pivotal role played by system design diagrams as the foundation for constructing security assertions unique to each tenant. Their formulation, encoding, and initial validation is described. Comprehensive explanations of the remaining stages in the construction process can be found in Chapters 5 and 6, respectively. The work described in this section was presented at the 4th International Conference on Information Systems & Management in the paper entitled 'Incorporating Security Features in System Design Documents utilized for Cloud-Based Databases' [179].

4.2 Overall design of the proposed CASE Tool

In this section, the primary procedure (Figure 4.1) employed in the development of the proposed framework is presented. The proposed tool is aimed at aiding software developers in creating secure multi-tenant database systems effectively and effortlessly. It plays a crucial role in streamlining the distribution and maintenance of access rights, ensuring consistency throughout.

Moreover, it provides continuous feedback on the database design and security profile, enabling prompt detection of potential errors and breaches. The valuable insights obtained from this feedback mechanism allow for appropriate adjustments to be made, resulting in ongoing system improvement and refinement, thereby contributing to the overall enhancement of the database environment.

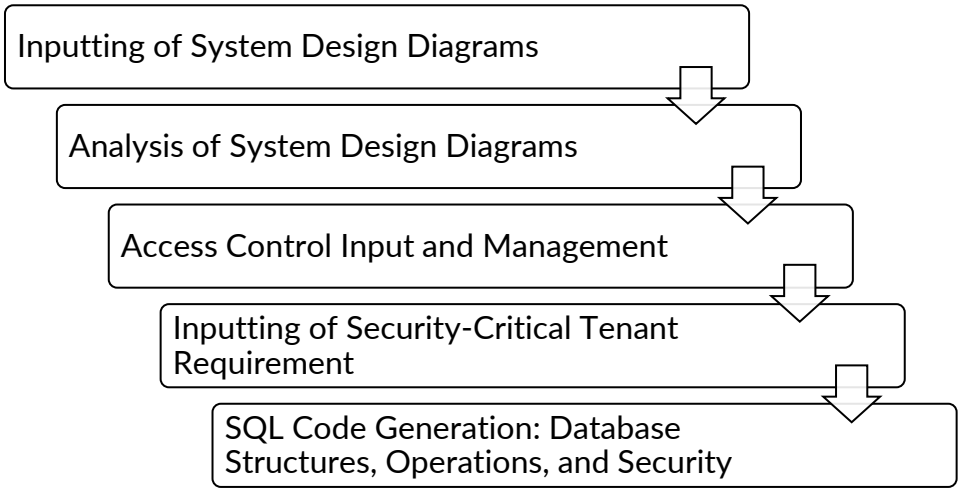


Figure 4.1: Proposed CASE Tool Workflow

The proposed CASE tool initiates its operation by accepting three types of system design diagrams (ERM, DFD, and multiple ELHs) that provide a comprehensive representation of the application. These diagrams are encoded and structured, in accordance with predefined rules, and then presented to the tool in JavaScript Object Notation (JSON) format. Subsequently, the aforementioned design diagrams undergo a comprehensive analysis, involving both individual and collective reviews, to identify and rectify any missing information. This scrutiny is vital for pre-emptively addressing gaps that could potentially result in vulnerabilities. Information presented in subsequent stages will offer additional insight to better evaluate the potential risks associated with these vulnerabilities.

Following completion of such analysis, access control mechanisms are defined to govern the allocation of rights and privileges for each role and database object. This stage primarily involves the automated generation of a comprehensive CRUD matrix, which depicts the requisite table access for each function. Subsequently, this matrix is expanded into a three-dimensional model, taking into account the various roles within the system. To ensure comprehensive access control, further enhancements are introduced, including views and other related components, which cover not only general access but also specific permissions related to individual rows and columns. Conclusively, this stage encompasses the execution of checks conducted on the access control management system created, to validate its proper functionality, ensuring that only authorised transactions are permitted.

The proposed CASE tool provides users with the capability of specifying individual tenant preferences, encompassing factors such as preferred multi-tenant configurations and encryption methods, which may vary from one tenant to another. Upon addressing the security feedback provided by the tool after each stage, the final phase commences, involving the creation of a secure database system. In addition to the SQL statements defining the database system's core architecture (Section 2.5.1), it addresses security concerns like SQL injection and aligns with the primary security requirements detailed in Chapter 3.

In summary, the tool relies on three primary inputs: system design diagrams, partition specifications, and tenant-specific security preferences. These inputs are then analysed and transformed, to result in the generation of SQL code that represents a secure multi-tenant database system. This approach offers substantial advantages by automating the creation of code that incorporates security assertions for each tenant. By eliminating the need for manual changes, the risk of errors resulting from possible human interactions is greatly diminished. Chapter 7 offers a comprehensive real-world example using a 'Banking System' CASE study, depicting typical banking operations, particularly those related to customer transactions and their connection to accounts. The ‘Scott’ system [180], containing information about employees, their affiliations with departments, and salary details, serves as the consistent reference point for explanations throughout Chapters 4 to 6. Appendix C.2 presents an additional CASE study featuring the School Management System, which showcases intricate relationships, offering valuable insights into the complex interactions among students, lecturers, departments, units, and transcripts.

4.3 System Design Diagrams

As can be noted from Figure 4.1, the initial stage of the proposed tool requires the inputting of an appropriate set of system design diagrams capable of representing the application. Considering the proposed tool's approach of prioritising security from the outset of database design, the first stage plays a pivotal role in shaping the overall outcome. Subsequent processes in the tool, including security checks and implementations, as well as the final result in the form of generated SQL code, are all based on

the presented set of diagrams. The following three primary structured diagrams were used to portray a database system in its entirety:

1. **Entity Relationship Management (ERM):** such diagrams are used to represent the primary tables comprising a database system and their relationships.
2. **Entity-Life History (ELH):** which depicts all conceivable states of the entities presented in the ERM, along with their transitions from one state to another.
3. **Data Flow Diagram (DFD):** which is used to portray the exchange of information among various components presented in both ERM and ELH, including entities, processes, and datastores.

Such diagrams provide not only a graphical representation of the entire system but also prove to be highly beneficial in implementing security policies at the design stage. They serve as a foundation for annotating security aspects, allowing privileges and access rights to be directly co-related from such diagrams and subsequently implemented through access controls. Attaining an appropriate balance during the implementation of security annotations is crucial. Overly generic annotations can dilute the overall system security, whereas excessively specific annotations may render the system impractical, hindering its management and user-friendliness to the point where end users are unable to effectively utilise it.

The following sections in this chapter provide a detailed discussion of how to construct each diagram, the essential components that must be incorporated, and the format which must be used, to ensure clear comprehension of the proposed tool.

4.4 Assumptions

During the development of the proposed CASE tool, a few assumptions related to the expected system design diagrams were made, namely:

- Only binary and ternary relationships are considered in the ERM. This decision was based on the knowledge that any other n-ary connection may be transformed into such relationships [181].
- A system is denoted by one DFD, one ERM and multiple ELHs. The basis for this assumption was its reflection of typical system representations in real life. While a system is represented by a single ERM and DFD, each entity within the

system undergoes a life cycle, moving through diverse phases, resulting in a distinct ELH associated with each entity.

- In ELH representations, processes are listed in sequential order of execution.
- Partial keys in weak entities of ERMs are treated as primary keys i.e., they are distinct for each tuple in a weak entity. This approach avoids the need for generating an auto-generated primary key or constructing a primary key set that includes both the partial key and the foreign key of the owner entity.
- Each primary key attribute associated with the entities found in the ERM, is simple, non-derived, and single-valued. These characteristics are assumed since they are the advised settings for primary keys in database systems to ensure efficiency and data integrity [182].
- In cases where entities within the system lack a dedicated ELH, a rudimentary ELH is generated, covering the essential operations of insertion, updating, and deletion.

4.5 Encoding and representation of design diagrams

Three sets of Extended Backus-Naur Form (EBNF) rules were developed to define the syntax and grammar for all ERMs, ELHs, and DFDs submitted to the proposed CASE tool. Adherence to these rules is mandatory for each diagram. Within these rules, italics text represents keywords that must be precisely replicated in the actual system diagram. Given the adherence to EBNF notation, the use of square brackets ([]) signifies optional text, curly braces ({}) represent repetitions, and the vertical bar (|) indicates a choice [183].

All system diagrams presented to the tool are encoded as JSON files with extensions .elh, .erm or .dfd depending on the type of diagram being represented. The preference for JSON format stems from its widespread adoption as a standard in cloud-based database management systems, ensuring compatibility and facilitating seamless integration with existing technologies [184].

4.5.1 Encoding of ERM

Thirty-eight (38) rules were developed for the representation of ERM diagrams. The ERM encoding is hierarchical in nature and starts with the definition of the main entities comprising of the entity's name and attributes. Each attribute is defined by means

of its name, data type, and other properties indicating whether the attribute is compulsory or optional, derived or non-derived, single or multi-valued, simple or composite. If required, these are then followed by the listing of weak entities with a similar set of characteristics and properties.

Subsequently, all relationships present in the system are indicated and categorised under one of five main categories – 'SE (Strong entity)-SR (Strong Relationship)-SE', 'SE-WR (Weak Relationship)-WE (Weak Entity)', 'WE-WR-WE', 'WE-SR-SE' and ternary. 'SE-WR-SE' and 'WE-SR-WE' are not considered due to their illogical nature. Apart from the indication of such category, the relationship name, attributes, and list of participating entities together with their participation and cardinality are also specified. In Listing 4.1 below, a subset of these rules is illustrated, specifically those concerning the definition of strong entities. The complete set of rules can be found in Appendix A.1.

Listing 4.1: A subset of EBNF rules for ERM

1: STRONG_ENTITY := *"strong entity name"*: string [{, *"attributes"*: *[*[{lstPK_ATT}] [{,}] [{ lstATT }] [{, lstCOMPOSITE_ATT }]*]* }]
2: lstATT := *{*ATTRIBUTE*}* [{, *{*ATTRIBUTE*}*}]
3: lstPK_ATT := *{*PK_ATT*}* [{, *{*PK_ATT*}*}]
4: ATTRIBUTE := *"attribute name"*: string, ATT_TYPE, REQUIRED, ATT_MODE
5: PK_ATTR := *"attribute name"*: string, ATT_TYPE, *"required"*: *"yes"*, *"attribute mode"*:*["non-derived", "single", "primary key simple"]*
6: ATT_TYPE := *"attribute type"*: *"integer"* | *"boolean"* | *"character varying* [(numberOfChars)]*"* | *"date"* | *"double precision"*
7: REQUIRED := *"required"*: *"yes"* | *"no"*
8: lstCOMPOSITE_ATT := *{*COMPOSITE_ATT*}* [{, *{*COMPOSITE_ATT*}*}]
9: COMPOSITE_ATT := *"main attribute name"*: string, *"composition"*: *"composite"* [{, lstCOMPOSITE_ATT }], *"secondary attributes"*:*[*lstATT*]*
10: numberOfChars := integer
11: ATT_MODE := *"attribute mode"*: *[*DERIVE, VALUED, *"simple"]*
12: DERIVE := *"derived"* | *"non-derived"*
13: VALUED := *"single"* | *"multi"*

These EBNF rules serve as the basis upon which all ERMs presented to the system in JSON format are based on. Figure 4.2 below displays the JSON representation

of a segment of the 'Scott' schema's ERM, particularly the 'Department' entity, one of its attributes, and a singular corresponding relationship with the 'Employee' table. The presence of ellipsis (...) implies that the complete ERM diagram representing the system comprises further components of the same type.

```
{
 "name": "Scott",
 "strong entities": [{
   "strong entity name": "Department",
   "attributes": [{
     "attribute name": "departmentNum",
     "attribute type": "integer",
     "required": "yes",
     "attribute mode":["non-derived", "single", "primary key simple"]
   }, ......
   ]}],
 "SE-SR-SE relationships": [{
   "relationship name": "Works in",
   "strong participating entity 1": {
     "strong participating entity name": "Employee",
     "participation in relationship": "total",
     "cardinality": "M"
   },
   "strong participating entity 2": {
     "strong participating entity name": "Department",
     "participation in relationship": "partial",
     "cardinality": "1"
   }}, ......
 ], ......
}
```

Figure 4.2: Subset of JSON encoding for 'Scott' ERM

Figure 4.2 begins with the definition of the 'name' component, indicating the system's name ('Scott' in this case). It is followed by the definition of the 'Employee' entity and its 'attributes'. The 'departmentNum' attribute is linked to an integer datatype and designated as mandatory. Additionally, the nested 'attribute mode' array specifies that 'departmentNum' is the primary key attribute and is single-valued and non-derived. The JSON file concludes by defining the 'SE-SR-SE relationship' between the ‘Department’ and ‘Employee’ tables. The entities' participation in the relationship (‘Department’ is partial, ‘Employee’ is total) is conveyed by the value following the 'participation in relationship' component, while the degree of association (many-to-one) is indicated through the 'cardinality' component.

4.5.2 Encoding of an ELH

Every entity stated in the ERM has a corresponding ELH that identifies sequentially, the processes responsible for the entity's change in state. Consequently, the five (5) EBNF rules (Listing 4.2) developed for ELH representation are directed towards the specification of entity names and their associated processes. Processes can encapsulate other sub-processes and are comprised of a name and type - asterisk (*) or circle (o), identifying whether the process is iterative or selective. Iterative processes are those which can be repeated indefinitely, whilst selective ones imply OR occurrences.

Listing 4.2: EBNF Rules for ELH

1: ELH := {ENTITY, *"processes"*:[lstPROCESS]}
2: ENTITY := *"entity"*: string
3: lstPROCESS := {PROCESS} [{, {PROCESS}}]
4: PROCESS := *"processname"*: string, OPTION [, *"processes"*: [lstPROCESS]]
5: OPTION := *"option": "0"* | *"*"* | *"none"*

Figure 4.3 below presents an example of an ELH, utilising the rules outlined in Listing 4.2. This example focuses on the 'Department' entity within the 'Scott' system. It demonstrates that three processes ('newDepartment,' 'modifyDepartmentInformation,' 'closeDepartment') directly affect the entity's state. The 'newDepartment' adds new tuples to the 'Department' table, 'modifyDepartmentInformation' updates existing department attributes, and 'closeDepartment' removes tuples representing departments that are no longer operational. The 'option' parameter is consistently set to 'none,' indicating that none of these processes are iterative or selective.

```
{"name": "Scott",
"entity": "Department",
"processes": [
 {"processname": "newDepartment",
  "option": "none"
 },
 {"processname": "modifyDepartmentInformation",
  "option": "none"
 },
 {"processname": "closeDepartment",
  "option": "none"
 }
]}
```

Figure 4.3: JSON encoding for 'Scott' ELH – 'Department' table

4.5.3 Encoding of a DFD

The DFD encoding is represented by a set of eleven (11) rules, illustrated in Listing 4.3, which contain definitions for entities, data stores, processes, and dataflows specified in that order. The former three categories, define the main elements of a DFD and are defined solely by the names of the corresponding component. The latter category delineates the connection between all components mentioned in the prior three categories. Its definition encapsulates the dataflow's name together with the names and types of both its inward and outward connected entities. Connected entity types can be set solely to data store, processes or external entities.

Listing 4.3: EBNF Rules for DFD

1:	DFD	:=	{NAME , *"external entities"*: *[*lstEXT_ENTITY*]}* [, *"outside external entities"*:*[*lstOUTSIDE_EXTERNAL_ENTITY*]}*] , *"data stores"*:*[* lstDATA_STORE *]}* , *"processes"*:*[* lstPROCESS *]}*, *"dataflows"*:*[* lstDATAFLOW *]}*
2:	NAME	:=	*"name"*: string
3:	lstOUTSIDE_EXT_ENTITY	:=	*{*OUTSIDE_EXT_ENTITY*}* [{,*{*OUTSIDE_ EXT_ENTITY*}*}]
4:	lstEXT_ENTITY	:=	*{*EXTERNAL_ENTITY*}* [{,*{*EXTERNAL_ENTITY*}*}]
5:	lstDATA_STORE	:=	*{*DATA_STORE*}* [{,*{* DATA_STORE *}*}]
6:	lstPROCESS	:=	*{*PROCESS*}* [{,*{*PROCESS*}*}]
7:	EXT_ENTITY	:=	*"external entity name"*: string
8:	OUTSIDE_EXT_ ENTITY	:=	*"outside external entity name"*: string
9:	DATA_STORE	:=	*"data store name"*: string
10:	PROCESS	:=	*"process name"*: string , *"level"*: integer
11:	lstDATAFLOW	:=	*{*DATAFLOW*}* [{, *{* DATAFLOW *}*}]
12:	DATAFLOW	:=	*"dataflow name"*: string , *"inward connected entity name"*: string , *"inward entity type"*: TYPE *", outward connected entity name"*: string ,*"outward entity type"*: TYPE
13:	TYPE	:=	"external entity" | "outside external entity" | "data store" | "process"

Similar to the previous two diagrams, these rules serve as the foundation for constructing JSON-encoded DFDs when presented to the proposed tool. Figure 4.4 displays a segment of the JSON DFD representing the 'Scott' system, including the 'Manager' external entity, 'Department' data store, 'closeDepartment' process, and the data flow connecting them. As detailed in Figure 4.2, the ellipsis (...) notation is used when additional objects are present in the complete DFD.

```
{
 "name": "Scott",
 "external entities": [
  {"external entity name": "Manager"}, ......
  ],
 "data stores": [
  {"data store name": "Department"}, ......
  ],
 "processes": [
  {"process name": "closeDepartment"}, ......
  }
 ],
 "dataflows": [
  {"dataflow name": "Department Information",
   "inward connected entity": "closeDepartment",
   "inward entity type": "process",
   "outward connected entity": "Manager",
   "outward entity type": "external entity" }, ......
 ]}
```

Figure 4.4: JSON encoding for 'Scott' DFD

4.6 Designing System Diagram-to-Data Dictionary Conversion

To streamline the secure SQL code generation process, a method was designed to organise the three system diagrams as relational tables, effectively acting as customised data dictionaries for each system. Through the process of streamlining, simplification and enhancement of efficiency can be achieved. Representing system diagrams as relational tables provides a clear and structured framework for data manipulation. It simplifies the process of generating secure SQL code by offering a readily accessible and organised format for data handling, contributing to the overall efficiency and security of the development process.

This section provides an overview of the process's design starting with the importing of three JSON files, representing the distinct system diagrams, into the TEXT column of 'import_elh,' 'import_erm,' or 'import_dfd' relation. After ensuring correctness through the 'is_json' function, the data is transferred to a JSON column of the same table. Following this, distinct relations are established to represent the components of each of the three diagrams. Several processes are then utilised to transform each JSON-formatted system diagram into relational tuples, which are then stored in appropriate tables. The ERM diagram below (Figure 4.5) illustrates the two relations used for the relational storage of ELH diagrams.

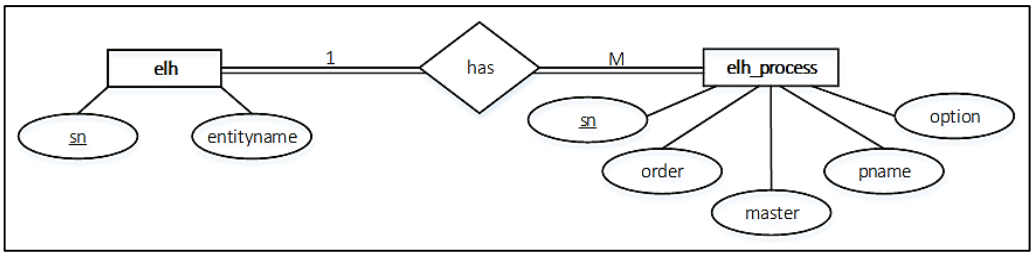


Figure 4.5: Relations used for the relational storage of ELH Diagrams

The ‘elh’ table lists entity names linked to an ELH, while the ‘elh_process’ catalogs processes directly affecting each entity's state. The former relation is filled using 'elh_import2predicates' (Pseudocode 4.1), and the latter is populated via Pseudocode 4.2. In pseudocode, '<-' indicates assignment or variable update, and '->' denotes data retrieval. For JSON columns, '->' returns JSON data, and '->>' returns text.

Pseudocode 4.1 Conversion of ELH to relational data structures

```
1:   Input: sysdiag_name text
2:   Output: void
3:
4:   Algorithm:
5:   If sysdiag_name is NOT NULL then
6:     sysDiag ←SELECT sysdiag_sn
7:                  FROM systemdiagrams
8:                  WHERE sysdiag_name = sysdiag_nm
9:     If sysDiag == 0 then
10:    For elh_curs in
11:      SELECT ie_serno, ie_json → entity
12:      FROM import_elh
13:      WHERE ie_sysdiag_serno = sysDiag
14:    Do
15:      elh ← SELECT ie_json
16:     FROM import_elh
17:     WHERE ie_serno = elh_curs.ie_serno
18:                                        >get ELH JSON to convert into predicates
19:     INSERT INTO elh
20:     VALUES elh_curs.iename, elh_curs.ie_sysdiag_serno
21:     RETURNING elh_e_sn INTO current_e_sn
22:
23:     idx ← json_array_length(elh → processes)      >extract process subtree, if any
24:     If idx == 0 then
25:     elh proc ← function json_extract(elh, processes)
26:     p_rtn ← function elh_import2predicates_processes(current_e_sn, null, proc)
27:     End if
28:   End for
29:   End if  End if
```

Pseudocode 4.2 Conversion of ELH processes to relational data structures

```
1:   Input: elh_sn integer, elh_master integer, json_fragment JSON
2:   Output: void
3:
4:   Algorithm:
5:   For rec_curs in
6:    SELECT t.proc → processname, t.proc → option, t.proc → processes,
7:        json_array_length(t.proc→ processes)
8:    FROM json_fragment → processes AS t(proc)
9:    Do
10:
11:   INSERT INTO elh_process
12:   VALUES (rec_curs.key, elh_sn, elh_master, rec_curs.pn, rec_curs.opt)
13:   RETURNING elh_p_sn INTO current_p_sn
14:
15:   If rec_curs.nsubp → 0 then
16:    p_trn←function elh_import2predicates_processes(elh_sn, current_p_sn,
17:    processlist)                                    >Recursive calling of function
18:   End if
19:   End for
```

The process of converting JSON-based DFDs into relational tables involves a greater number of tables compared to the ELH transformation process.

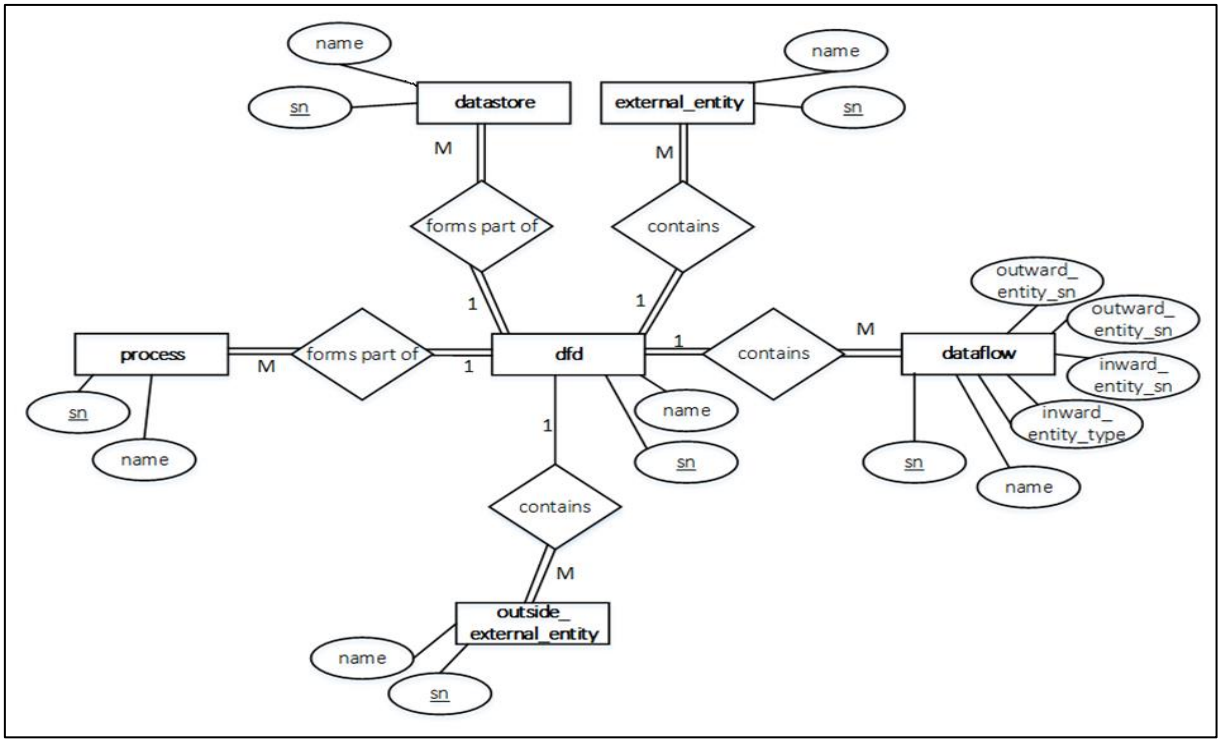


Figure 4.6: Relations used for the relational storage of DFD Diagrams

As can be noted from Figure 4.6, in addition to the main ‘dfd’ relation, distinct tables are created for each DFD component. Each table shares three common attributes, including a serial number, component name, and a foreign key link to the main table. Additionally, the 'process' and 'dataflow' tables require specific additional attributes, as highlighted in the previous figure.

Pseudocode 4.3 Conversion of DFD to relational data structures (dataflow segment)

```
1:    Input: sysdiag_name text
2:    Output: void
3:
4:    Algorithm:                                   >extract the dataflows fragment
5:     dfcounter ← json_array_length(dfd → dataflows)
6:      If dfcounter == 0 then
7:       dfd_extent_df ← function json_extract(dfd, dataflows)
8:       For rec_curs in
9:      t.rel -» dataflow_name, t.rel -» inward_entity_type,
10:     CASE t.rel -» inward_entity_type
11:      external_entity then
12:     SELECT dfd_ee_sn
13:      FROM external_entity
14:      WHERE dfd_ee_name = t.rel-»'inward connected entity'
15:     AND dfd_ee_h_sn = current_d_sn)
16:      data_store then
17:            SELECT dfd_d_sn
18:      FROM data_store
19:      WHERE dfd_ee_name = t.rel -» 'inward connected entity'
20:      AND dfd_d_h_sn = current_d_sn
21:           process then
22:           SELECT dfd_p_sn
23:      FROM process
24:      WHERE dfd_p_name = t.rel -» 'inward connected entity'
25:      AND dfd_p_h_sn = current_d_sn)
26:          END CASE,
27:          t.rel -» 'outward entity type' ,
28:          CASE t.rel -» outward entity type …
29:          END CASE,
30:          FROM dfd_extent_df→'dataflows' AS t(rel)
31:
32:     Do                                        >insert each dataflow as a tuple
33:     INSERT INTO dfd_dataflow
34:     VALUES (current _d_sn, rec_curs.dfn, rec_curs.iet, rec_curs.iesn,
35:      rec_curs.oet, rec_curs.oesn)
36:     RETURNING dfd_df_sn INTO current_df_sn
37:     End for
38:      End if
```

The 'dfd_import2predicates' function converts the JSON-formatted DFD into relational tuples organised into five sections, each corresponding to a specific component. For each section, a FOR-loop iterates through the elements of the related component type, extracting and saving the component names as tuples in the appropriate table, with additional processing for the 'dfd_dataflow' table. Pseudocode 4.3 maps dataflow components to tuples, with the remaining sections detailed in Appendix A.2.

JSON-based ERMs are also converted into relational tables. The ERM transformation process is similar to that of the other diagrams but requires significantly more tables. Apart from the main ‘erm’ relation, separate tables are used for the storage of strong and weak entities, their corresponding attributes, and all relevant relationships. The full list of relations required with their attributes are displayed in Figure 4.7.

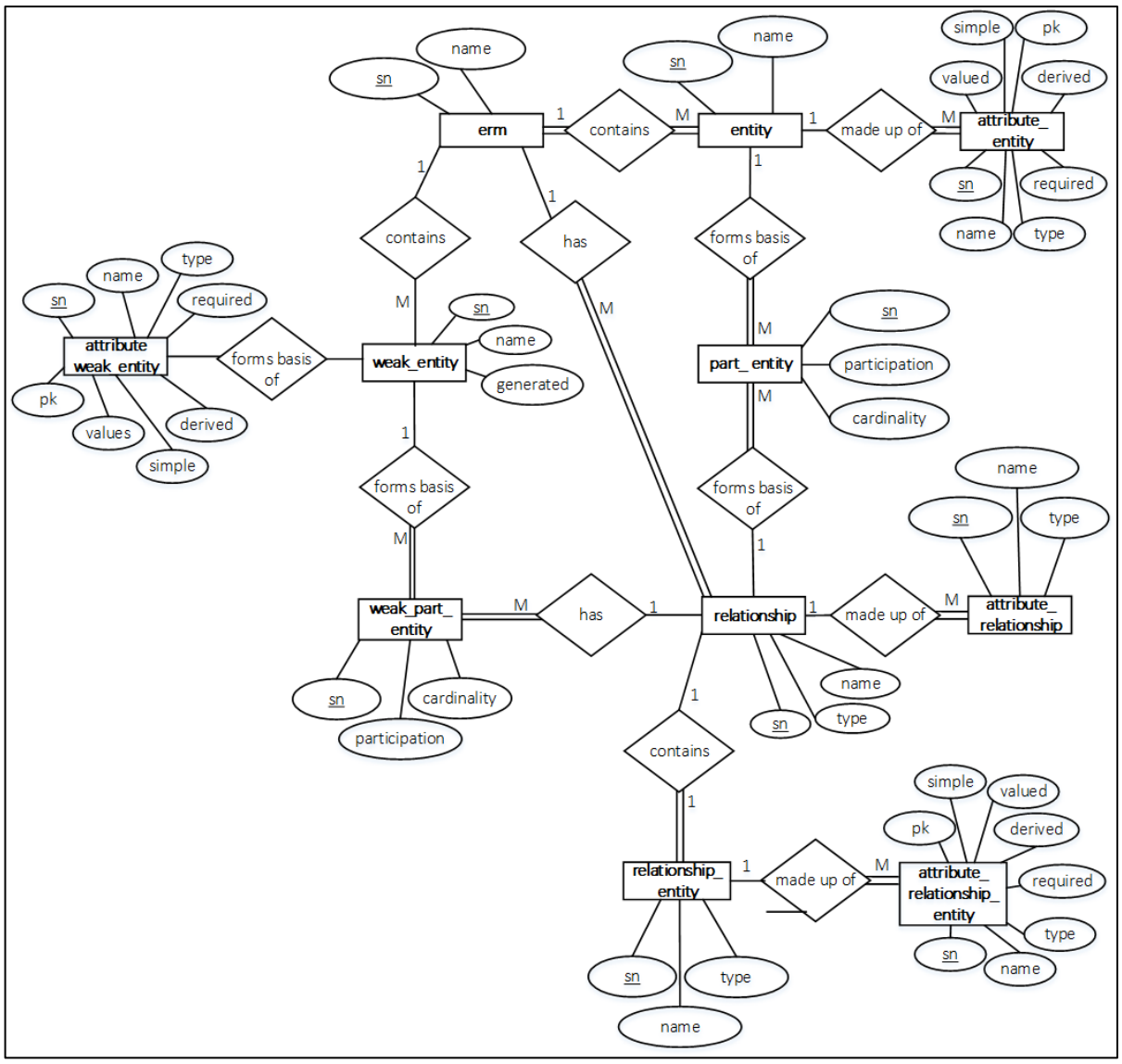


Figure 4.7: Relations used for the relational storage of ERM Diagrams

The procedure for converting JSON-formatted ERM into relational tuples is 'erm_import2predicates'. It retrieves the JSON ERM from the 'erm_import' table and inserts a new tuple into the main 'erm' table. A FOR-loop iterates through strong and weak entities in the JSON file, creating tuples for each entity in appropriate relations and linking them to the main table. Tuples for entity attributes are generated and stored in 'erm_strong_attribute' or 'erm_weak_attribute' tables as needed, reflecting attribute characteristics from the JSON file. For boolean values, IF-statements are used to convert text information to true or false values. Subsequently, the 'erm_relationships_import2predicates' function converts specified relationships into tuples. This function is divided into sections for each of the five allowable relationships, with Pseudocode 4.4 providing an algorithmic representation of the section for strong entity relationships. Similar procedures (Appendix A.3) handle the conversion of other relationships.

Pseudocode 4.4 Conversion of 'SE-SR-SE' relationships to relational data structures

```
1:  Algorithm:
2:    rc ← json_array_length(json_fragment → SE-SR-SE rel)
3:    If rc > 0 then
4:    erm_extent_sss ← function json_extract(json_fragment, SE-SR-SE rel)
5:    For rec_curs_sss in
6:    (erm_extent_sss → SE-SR-SE rel)
7:    Do
8:    INSERT INTO erm_relationship
9:    VALUES (erm_sn, rec_curs_sss.rn, 'strong')
10:   RETURNING erm_r_sn INTO current_r_sn
11:
12:   INSERT INTO erm_part_entity
13:   VALUES (current_r_sn, rec_curs_sss.particEnt, rec_curs_sss.cardEnt,
14:    rec_curs_sss.partEnt)
15:   RETURNING erm_pe_sn INTO current_pe_sn
16:   For rec_curs_rel_att in
17:    SELECT rel→'relationship attribute' -» 'attribute name', rel →
18:    'relationship attribute' -» 'attribute type'
19:    FROM erm_extent_sss → SE-SR-SE rel
20:    WHERE rel → 'relationship attribute' -» 'attribute name' IS NOT NULL
21:    AND rel -» 'relationship name' = current_r_sn.name
22:   Do
23:        INSERT INTO erm_attribute_relationship
24:    VALUES (current_r_sn, rec_curs_rel_att.attName, rec_curs_rel_att.attType)
25:    RETURNING erm_ra_sn INTO current_ra_sn
26:   End for
27:   End for
28:  End if
```

4.7 System diagrams suite of checks

Following the competition of all preceding sections in the chapter, a suite of checks was developed to systematically analyse each diagram presented to the CASE tool. This analysis acts as the initial gauge, in determining system adequateness prior to the implementation of any security features.

A text file, entitled ‘PreAndPostChecks.txt’, documenting the complete list of all checks is created and provided to the CASE tool. Every line in the aforementioned file corresponds to the names of the procedures responsible for executing each individual check. This approach is advantageous since it enables the effortless integration of any newly developed checks in the future and ensures that their results are included alongside the existing ones. The outcome of all checks is exhibited in tabular format made up of three columns indicating the check name, any data which violates the check criteria and the final status of the check, either ‘Good’, ‘Bad’ or ‘Warning’.

4.7.1 Checks on individual system diagrams (Pre-checks)

The first set of checks is targeted towards the assessment of ERMs, ELHs and DFDs in an independent manner. The cluster of checks developed for each diagram encapsulates syntactic, lexical, and semantic aspects to ensure the analysis of each diagram in a holistic manner. Whilst syntactic and semantic checks necessitated the development of specific functions or stored procedures, lexical checks were in most cases implemented via relational constraints. A detailed explanation of the system diagram checks developed for the proposed CASE is explained hereunder.

4.7.1.1 Checks conducted on ELH

1. **Every ELH should be made up of at least two processes.**

ELHs, which illustrate an entity's lifecycle, play a crucial role in determining the required GRANT privileges for relations based on the executed process. Having less than two processes associated with a particular entity impedes an entity from being either inserted to or deleted from depending on which process is missing. While some entities are intentionally set up as such, serving as predefined values that are not meant to be further deleted or inserted, the majority do not fall into this category. Consequently, this check reviews the ELHs within the system, triggering a 'Warning' if

any of them contain fewer than two processes and providing clear identification for these ELHs. Conversely, it yields a 'Good' result for ELHs with two or more processes.

2. **The set of ELHs forming a system cannot include duplicate processes.**

Permitting duplicate processes within the same application introduces the risk of ambiguity regarding which entities these processes affect. This ambiguity can, in turn, create unexpected security vulnerabilities. Specifically, it may lead to incorrect associations between processes and entities, as well as misconfigured permission grants. In practical terms, this means that certain entities could unintentionally inherit permissions and actions from duplicated processes, potentially compromising data security and overall system integrity. This check produces a 'Bad' result if a duplicate process is detected, accompanied by the name of the duplicate process. Otherwise, the outcome produced from this check is 'Good'.

3. **A system having two ELHs for the same entity is prohibited.**

When multiple ELHs are associated with a single entity, it can introduce confusion regarding which ELH to follow, leading to potential data inconsistencies and operational challenges. By enforcing the rule that all processes associated with an entity must be clustered within a single ELH, it ensures a streamlined and unambiguous approach to tracking and managing that entity's lifecycle. This helps maintain data integrity and system stability. This check verifies that all processes linked to a specific entity are consolidated within a single ELH. It yields a 'Good' result if this condition is met; otherwise, it returns 'Bad'.

4.7.1.2 Checks conducted on ERM

1. **Entities specified in the EBNF's relationship section of the ERM must correspond to those inputted in the entities section.**

This validation check inspects the content of the presented ERM JSON encoding, with a specific focus on identifying entities listed in the 'relationship' section that are not specified in the 'entities' section. It yields a 'Good' result when all entities found in the relationship section are consistent with those already documented in the entities section. Conversely, a 'Bad' result is issued if any entities fail to meet this criterion, highlighting the specific entities that require attention.

2. **Every recursive relationship has at least one partial participation.**

Recursive relationships represent self-referencing connections i.e., entities participating more than once in the same relationship with different roles. Having both entity instances of a recursive relationship in total participation implies that each entity's instance should be associated with at least one other entity in the set. This might indicate a possible flaw in the system's design since, in most circumstances, the entity instance representing the parent component is exempt from having children associations. Therefore, having at least one partial participation in a recursive relationship provides more flexibility and better reflects the reality of the situation being modelled. Thus, the objective of this check is to notify the designer about a potential issue by generating a 'Warning' when a recursive relationship lacks at least one partial participation.

3. **Ternary relationships are only possible with strong entities.**

This check was formed to conform with the assumption listed in Section 4.4.

4. **Each entity should ideally be connected to at least one relationship.**

The purpose of this check is to identify instances of stand-alone entities in the original database design. This helps mitigate the risk of potential security vulnerabilities, including privilege escalation, data integrity issues, and data leaks that could be exploited by attackers. This check issues a 'Warning' along with the name of tables lacking connections to at least one relationship. Nevertheless, it doesn't forbid such instances, as in exceptional cases, tables like information or reference tables may not require any relationships.

5. **A weak entity should have a direct or indirect connection to a strong entity parent.**

A weak entity's existence relies on another entity, which implies that while weak entities may connect to other weak ones, the final entity in the chain must invariably be a strong one. In the context of an ERM, the participation of a weak entity in a relationship with a strong entity is always 'total' since a weak entity cannot exist independently, a crucial element in preserving data integrity.

Utilising a recursive algorithm, this check identifies all weak entities within the system and verifies that they are linked, either directly or indirectly, to at least one

strong entity. If this condition is not met, the check outputs a 'Bad' result and identifies the specific weak entity causing the issue.

6. **Entity and attribute names are unique.**

The UNIQUE relational constraint is employed to perform this syntactic check, ensuring the uniqueness of all entities and their attributes. Each entity in a database represents a distinct object characterised by its unique attributes. Therefore, the presence of duplicate entities or attributes within the same entity can compromise the data integrity of the database.

4.7.1.3 Checks conducted on DFD

Unless otherwise specified, the majority of the checks in this section are structural ones, targeted towards ensuring that the DFD notation's rules are adhered to.

1. **Inputs and outputs specified in the EBNF's dataflow section of the DFD, must be consistent with those defined as entities, processes or data stores.**

This validation check scrutinises the provided DFD file within the CASE tool for accuracy and consistency. Its purpose is that of detecting data objects utilised within dataflows, whether as inputs or outputs, that are not defined in any of the four other DFD sections: entities, processes, or data store components.

2. **At least one of the connecting elements in a dataflow is a process.**

The following connections are not permitted in a DFD: 'data store-data store', 'entity-entity', 'data store-entity'. This implies that only connections involving at least one process - 'process-process', 'process-entity', and 'process-data store' - are permitted. This restriction is in place since processes are central to data manipulation within the system, requiring data to flow through them for transformation.

By SELECTing all connecting elements in each dataflow and ensuring that at least one of them is of type 'process', this validation ensures the enforcement of the rule. It returns either 'Good,' indicating compliance with the rule, or 'Bad,' signifying non-compliance. If the latter is the case, it provides a list of the elements that fail to adhere to this criterion.

3. **Every process and data store should have at least one input and one output.**

The prior rule is stipulated by the DFD requirements to ensure that the system's functionality is appropriately presented, and all relevant data is being captured and processed. This check guarantees that some sort of input is supplied to all data

stores and processes prior to their execution. Moreover, since functions and data stores without output are useless, it also ensures that an effective result is delivered.

4. **Every external entity should be related to at least one dataflow.**

External entities in DFD are equivalent to roles in real-life. Arguably, each role must be able to execute at least one process, and thus, must be connected to a minimum of one dataflow. The opposite of the aforementioned statement might be indicative of a flaw in the database design, suggesting that a role present in the system is not required. This might pose a security risk since unused roles might be utilised to access the database without authorisation, increasing the system's susceptibility to attacks. Displayed in SQL Code 4.1 below is a PostgreSQL code snippet of this check, which accepts the DFD under review as its input argument.

```
PERFORM dfd_df_name
FROM "spdba".dfd_dataflow
WHERE dfd_df_outentitytype = 'external entity'
AND  dfd_df_outentitysn =
                (SELECT dfd_ee_sn
          FROM "spdba".dfd_external_entity
          WHERE dfd_ee_name = rec_curs
          AND dfd_ee_h_sn =
                        (SELECT dfd_h_sn
    FROM  "spdba".dfd
    WHERE dfd_h_sysdiag =
                                (SELECT sysdiag_sn
      FROM "spdba".systemdiagrams
      WHERE sysdiag_name = sysdiag_nm)))
)
```

SQL Code 4.1: Code snippet of the function implementing Check 4

5. **Non-leaf nodes should not have data store connections whilst leave nodes should.**

Non-leaf nodes denote parent processes having at least one or more child processes linked to them. On the other hand, leaf nodes serve as an endpoint for data, hence, they must be linked to a data store which indicates the data's origin or storage location. If the prior statement is not in the affirmative, data stored in relations can never be reached. This is because direct access to tables is prohibited due to the security risk of having unmonitored and unauthorised data access. Provided that is the case, the system's overall integrity and accuracy is compromised since data in tables is not updateable.

6. **External entities must be exclusively linked to first-level parent processes.**

Roles engaging with the database system being portrayed are indicated by external entities. Naturally, these roles ought to have access to a set of processes that they may use. To preserve the security concept of least privilege, roles are only granted access to first-level processes and are never granted direct access to any child process. The latter processes are invoked by their parent processes according to the data inputted by the users. This safeguard is implemented via SELECT statements encapsulated in two different check routines - ‘dfd_mainprocess_connectedto_entity’ and ‘dfd_processnot main_notconnectedto_entity’. The first function guarantees that at least one of the dataflows connected to each process has the inward or outward type set to 'external entity,' while the second process ensures that the inverse is true for the remaining processes.

7. **Processes which are not root or leafs must be linked solely to other processes.**

According to the DFD specifications, any dataflows connected to processes that aren't roots or leafs must have both their inward and outward components assigned to processes. This is due to the connections that these processes must have with both their children and their parents. Check 7 ensures that all such DFD processes comply with this requirement.

8. **Data stores, external entities, processes and the combination of inward entity name and type–dataflow–outward entity name and type are unique.**

The above-specified syntactic checks are implemented through UNIQUE relational constraints. These restrictions are necessary to prevent the granting of unsolicited privileges and access rights to objects.

4.7.2 Checks on combined system diagrams (post-checks)

The three system diagrams required by the proposed tool are used interchangeably when verifying and asserting security policies, since the same element might feature in multiple diagrams. For this reason, another set of checks was developed. Figure 4.8 graphically highlights the main interlinking processes between an ELH, ERM and a DFD and serves as the foundation upon which the checks listed hereunder were based up.

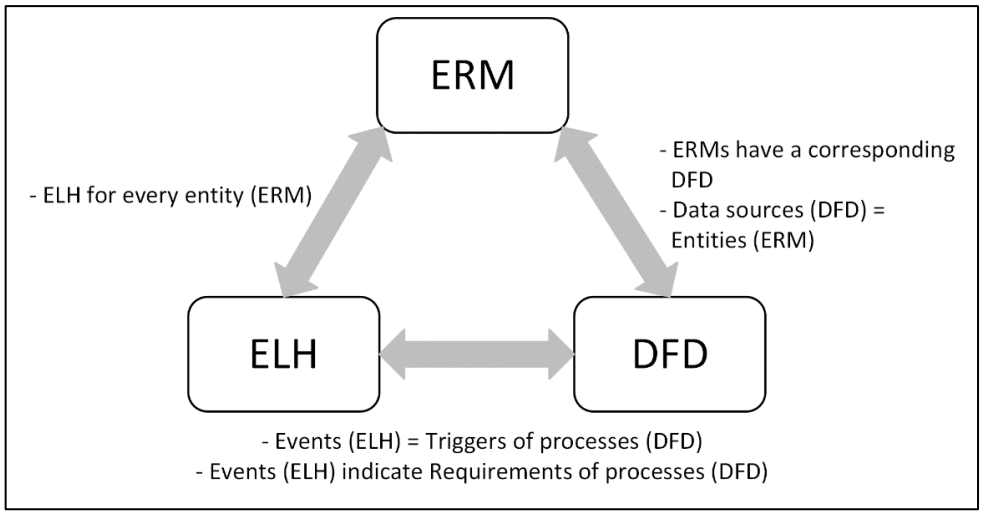


Figure 4.8: Inter-relationships between ERM, ELH, DFD

1. **Every process listed in any ELH of a particular system must be present in the DFD of the same system.**

This check makes sure that every process that has the potential to change an entity's state has a defined mode for how data enters and exits the process. Such a check is vital for accountability purposes, particularly in relation to any actions that might result in data transformation. The converse statement might not always hold (i.e., not all processes found in DFDs are present in ELHs). Some processes, including the creation of reports based on entities, do not feature in ELHs since there is no direct impact on the entity's state but rather serve as a source of information.

2. **All entities listed in the ERM must be specified as a data store in the DFD and vice versa.**

ERMs highlight the connections between the data stores identified in the DFD that are otherwise only apparent during the specification phase. Such entities correspond to database objects, including base tables and views, upon which security rights and privileges are granted. A symmetric difference operator is utilised in the implementation of this check to ensure that the same entities are contained in both the ERM and DFD diagrams.

3. **Every entity listed in the ERM must have a corresponding ELH.**

This check is required, to avoid redundant entities which might undermine the security of the whole system. All different states each entity can experience during its lifetime should be accounted for in its corresponding entity-life history. Apart from

avoiding direct data access, processes associated with entities also serve as a safeguard to circumvent data breaches. The likelihood of being the target of a cyberattack increases with the amount of inaccessible data an organisation possesses. The difference operator, utilised in the implementation of such a check retrieves all ERM entities which do not have a corresponding ELH. If this operation yields a positive result, the ‘elh_any’ function is invoked, an ELH is generated and the three rudimentary processes (Insert, Update, Delete) are created and associated with the entity.

4.8 Design of setup related to other key users

The design of the proposed tool, discussed in this chapter, primarily centres on system diagrams depicting tenant-provided information. This is due to the fact that the CASE tool developed for this thesis is specifically customised to prioritise tenant needs, and to create tenant-specific security profiles. Nevertheless, as outlined in Section 2.8.2, multi-tenant CDBMS environments inherently encompass diverse user roles. For the purpose of this dissertation, the basic setup of the other key users involved, namely the CSP and software provider, was designed.

This encompassed the design of a CSP role along with an associated authorised user. It also involved a dedicated database responsible for the storage of data related to the CSP's operations. This database is designed such that access to it is exclusively granted to the authorised CSP role, with all permissions associated with the public role deliberately revoked to ensure strict control and safeguarding of data. The operational details and contents of this database, although important, were deemed beyond the scope of this dissertation.

Subsequently, the setup for the software provider was designed, made up of a dedicated database comprising three distinct schemas: 'softwareproviderdba,' 'softwareproviderdeveloper,' and 'softwareprovidersecurity.' Access was intentionally restricted to just two specific roles: the software provider DBA and the developer. Additionally, usage privileges on these schemas were assigned to these roles in accordance with their respective access level. Following that, the essential foundation for the proposed tool's effective operation was established. This involved designing critical database objects related to overseeing ERMs, ELHs, and DFDs, managing tenant-specific tables and functions, executing CRUD operations, supporting role management and

strategically situating them within the previously mentioned three schemas. The implementation scripts corresponding to this design can be found in (Appendix A.4).

4.9 Summary

This chapter focuses on the design phase of the proposed tool, particularly emphasising the importance of system design diagrams as the foundation for all subsequent security assertions. Customised EBNF rules were developed to suit the unique requirements of three primary diagram types: Entity-Relationship Models (ERM), Entity-Life History (ELH), and Data Flow Diagrams (DFD). Thorough checks were subsequently carried out on each diagram, both in isolation and as an integrated whole, to determine the system's initial security status and identify any potential security vulnerabilities. Lastly, the design features related to other key users in a multi-tenant cloud-based environment were discussed providing a holistic perspective.

After successfully creating and validating design diagrams, as well as designing the infrastructure for all key users involved, the next pivotal phase in developing the proposed tool centres on the implementation of efficient access controls, a topic that will be explored in more depth in the subsequent chapter.

5 Access Control and Security Considerations

5.1 Introduction

This chapter focuses on the security access control framework for the proposed tool, building upon the foundation established through the reading and analysis of the application description, conveyed by the presented design diagrams. It delves into the implementation of RBAC mechanisms and explains the automatic generation of the CRUD matrix, essential for defining access permissions. Taking a step further, the chapter discusses the augmentation of the CRUD matrix by incorporating roles, enabling a more refined and granular approach to security management. Moreover, it explores the considerations for vertical and horizontal partitioning, ensuring the implementation of robust security-grained access control to safeguard critical resources effectively.

5.2 Initial stages of Access Control

Granting access privileges to database objects is one of the most critical processes during development and runtime, particularly due to its direct bearing on the overall system's security. To assist the implementation of this process, an access control or CRUD matrix is usually constructed. Additionally, it also assists in detecting possible security weaknesses and in identifying areas where permissions must be tightened up.

As explained in Section 2.9, these privileges are subsequently assigned to roles based on the least-privilege notion, thus ensuring that roles only have access to the least information necessary to accomplish their tasks.

Henceforth, following the reading and validation of the three system diagrams (described in Chapter 4), the proposed tool automatically generates a first level CRUD matrix. By extracting information from the presented system diagrams, the matrix summarises the table permissions required by each process, in particular SELECT, INSERT, UPDATE and DELETE. Such permissions are later updated to encapsulate any views (indicated as third level artefacts in Figure 2.2), present in the system. As dis-

cussed in the following sections, the determination of the least specific privilege required is based on the positioning of the process with respect to the ELH and DFD models.

5.3 Parent-child process associations

Parent-child process associations are a prevalent occurrence in most systems. A 'parent' refers to a process which assumes an overarching role, bearing higher-level responsibilities and the authority to delegate tasks or responsibilities to 'child' processes. The latter processes are situated lower in the hierarchy and serve to fulfil specific functions or tasks within the broader context defined by the ‘parent’ process.

Consequently, retrieving these process associations is a prerequisite prior to the creation of the CRUD matrix. The 'crud_matrixPP' procedure streamlines this data extraction process, commencing with the creation of a 'crudMatrixPP' relation consisting of three columns. The first column designates the parent process's name, the second identifies the child process, while the last column signifies their associated permission, typically 'EXECUTE.' This 'crudMatrixPP' relation is populated with all DFD processes connected to other processes. Subsequent to the retrieval of these relationships, an analysis is conducted to confirm that each parent process effectively encompasses the execution permissions of its associated child processes.

While the 'crudMatrixPP', is typically required for most systems, there are instances where its generation is superfluous. For instance, the 'Scott' system, which serves as a recurring example throughout this dissertation, represents a straightforward system where processes do not exhibit parent-child associations. Instead, all processes in this system are directly linked to data stores.

5.4 First level CRUD matrix

Following the successful creation and population of the 'crudMatrixPP' (explained in Section 5.3), the CRUD matrix is generated based primarily on the data present within the system design diagrams. The following sections provide a detailed explanation of this generation process, highlighting the significance of the system design diagrams presented within this context. Additionally, they detail the correlation of these diagrams with regards to the privileges granted to roles on database objects.

5.4.1 Relationship between diagrams and granting of privileges for security purposes

The first level CRUD matrix indicates the SELECT, INSERT, UPDATE and DELETE privileges granted on the base tables indicated in the ERM. This is later revisited to include privileges granted on views (discussed thoroughly in Section 5.6). Including views circumvents unnecessary data exposure, as granting privileges directly on the base table implies unrestricted access to all its attributes and rows. Figure 5.1 below depicts the relationship between an ERM and its utilisation in the implementation of security access control privileges.

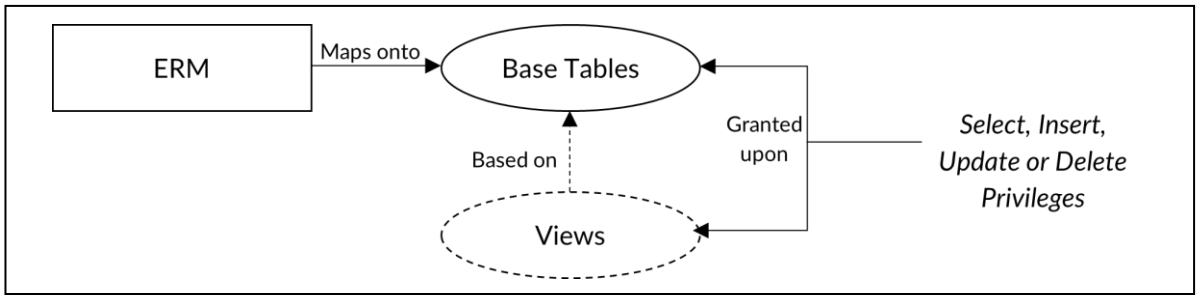


Figure 5.1: Relationship between ERM and security implications

Determining the type of privilege to be granted, is largely dependent on what is required from the entity. The various states and transitions of each entity are determined by the functions found in the ELH diagrams which in turn map to the majority of processes found in the DFD. The identification of the type of privilege required is established by considering first the position of the process within the DFD. Any outgoing dataflow from a data store to a process corresponds to a SELECT privilege on the data store (base table or view) required by the function. This is because it signifies that the operation requires data from the table.

Outgoing dataflows from processes to data stores correspond to INSERT, UPDATE or DELETE privileges on the data store required by the function. However, the challenge lies in selecting the required privilege (from INSERT, UPDATE or DELETE) and that is where the ELH proves to be important. The ELH depicts sequentially each state an entity goes through from its inception until its conclusion. Thus, if the process being considered, is at the start of the ELH, the INSERT privilege is required since there is the creation of a new entity. On the other hand, functions occurring on the

middle states of the ELH tree (i.e. not first or last), are those requiring UPDATE privileges since they are the ones responsible for progressing the entity from one state to another. Lastly, if a function is related to the final state of an entity, a DELETE privilege is required since it indicates the removal of a record in an entity. Figure 5.2 illustrates the relationship between the assignment of privileges upon the DFD and ELH diagrams.

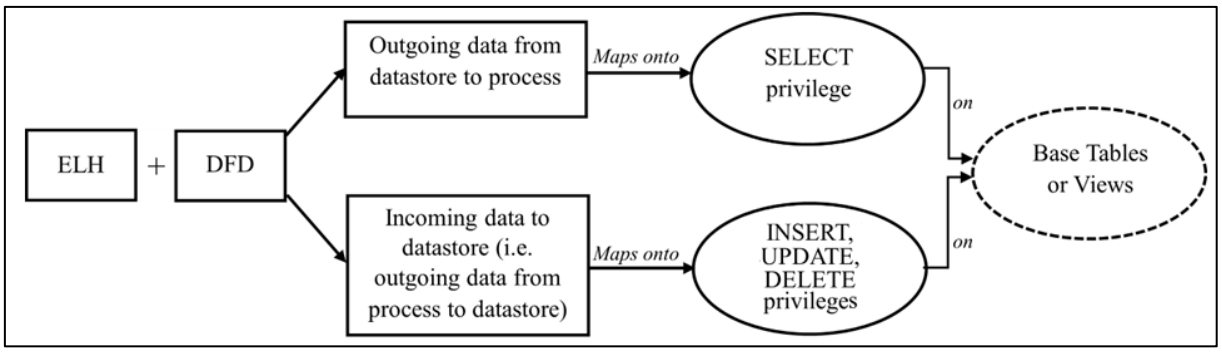


Figure 5.2: Relationship between ELH, DFD and security implications

5.4.2 CRUD Matrix generation and implementation

The methodology outlined in Section 5.4.1, is implemented via three procedures, with the principal one being ‘crud_matrix’. This method is responsible for creating the 'crudMatrix' table, comprising six (6) columns indicating the process name, and its related entity name and type, together with the associated permission, system and tenant identification numbers. It then identifies SELECT access privileges, equivalent to ‘Read’ operations, by retrieving the full list of processes and entities associated with a DFD dataflow having an inward process and an outward data store.

Subsequently, the ‘crud_matrix’ method, detects INSERT, UPDATE or DELETE permissions by extracting all processes with an inverse dataflow, i.e., process outward, data store inward. Since the associated permissions, in this case, are slightly more intricate because they may fall into one of three categories, the 'IUD' function is invoked. This function, which accepts an input parameter indicating the process name, is capable of determining which of the three privileges is required, based on the process defined in Section 5.4.1 and according to the procedure discussed hereunder. The process passed as an input parameter is analysed to establish whether it has a

parent process. If the prior assertion is true, then the 'IUD' function is recursively invoked, until the topmost parent (i.e., the one connected to a data store) is reached. Otherwise, an 'if-then-else' statement is used to determine the required privilege type. If the process's order number is 1 i.e., it is located within the first sub-tree of the ELH diagram, then the associated permission is set to INSERT. Alternatively, the permission is set to DELETE, if the process's order number is the largest one in the current ELH. Such a value indicates that the process is located at the end of the ELH sub-tree and thus, deals with the termination state of an entity. Otherwise, if the process is positioned somewhere in between, the associated permission is set to 'UPDATE' to cater for any alterations which might be performed on the entity's state. The implementation of this part of the procedure is demonstrated in Pseudocode 5.1 below.

Pseudocode 5.1 Algorithm for part of the 'IUD' Process

```
1:    Input: process_name text
2:    Output: text
3:
4:    Algorithm:
5:        >Non-parent process
6:    If masternum is NOT NULL then
7:      mProcessName ←SELECT DISTINCT elh_p_name
8:          FROM spdba.elh_process
9:         WHERE elh_p_sn = masternum
10:
11:     function spsecurity.IUD(mProcessName, systemNum);
12:                    >IUD function is called recursively until topmost parent is reached
13:   Else
14:     ordnum ←SELECT elh_p_order
15:      FROM spdba.elh_process
16:      WHERE elh_p_sn = sn
17:
18:     If ordnum == 1 then
19:     retValue = 'Insert'
20:     Elseif ordnum == MAX(elh_p_order) then
21:      retValue = 'Delete'
22:     Else
23:      retValue = 'Delete'
24:    End if
25:   End if
```

Following execution of the preceding process, the 'insertIntoCrud' procedure is invoked for each tuple in the result set. Apart from ensuring that the data present in

each tuple is inserted into the ‘crudMatrix’ table, the function invokes itself recursively. This recursion occurs for each tuple’s corresponding parent process, identified from the ‘crudMatrixPP’ table. The same parameters, associated with the child process, are passed in each iteration, except for the process name which is altered to reflect the parent process name.

Since only base tables are considered for the matrix generated in this section, the value of the CRUD matrix’s ‘crud_ds_type’ column is always set to ‘Table’. Permissions on other database objects such as views are considered subsequently in Section 5.6. Similarly, since for the time being, the permissions considered are generic i.e., they are directly extracted from the presented system design diagrams, the ‘tenant_id’ column is set to 0. Once, the matrix is expanded to capture vertical or horizontally partitioned roles, specific for each tenant, the ‘tenant_id’ value is updated to reflect the identification number of the tenant, associated with that partition. Table 5.1 illustrates the first-level CRUD Matrix for the ‘Scott’ case study (Appendix C.1).

Table 5.1: CRUD Matrix for 'Scott'

crud_ds_name	crud_ds_type	crud_p_name	crud_perm
Department	Table	closeDepartment	DELETE
Department	Table	generateDepartmentReport	READ
Department	Table	modifyDepartmentInformation	UPDATE
Department	Table	modifyEmployeeDetails	READ
Department	Table	newDepartment	INSERT
Department	Table	startWork	READ
Employee	Table	generateEmployeeReport	READ
Employee	Table	leave	DELETE
Employee	Table	modifyEmployeeDetails	UPDATE
Employee	Table	startWork	INSERT
Salgrade	Table	generateGradeReport	READ
Salgrade	Table	insertGrade	INSERT
Salgrade	Table	modifyGradeSalary	UPDATE
Salgrade	Table	removeGrade	DELETE

5.4.3 CRUD Checks

Following the automated creation of the CRUD matrix, a series of checks were developed to examine and verify the veracity of all permissions. These are essential to ensure that all relational objects are accessible and prevent security vulnerabilities associated with inaccessible objects. Similar to the system diagram checks outlined in Section 4.7, the output for each check is presented in a tabular format, encompassing the check's name, the result (which may include a list of objects in violation, when applicable), and its classification, which can be 'Good,' 'Warning,' or 'Bad'.

The full set of checks developed is explained hereunder:

1. **Every object in the CRUD matrix should have at least one SELECT privilege associated with it.**

A functional database system is one whereby all objects are accessible and readable. Thus, the presence of at least one SELECT privilege per object is vital. A lack of this implies that data cannot be queried or retrieved, rendering the database item completely unreachable. Moreover, the absence of such privilege augments the risk of having incomplete security controls, thereby making it easier for attackers to access sensitive data or conceal their footprints following a breach.

Pseudocode 5.2 illustrates part of the ‘crud_checkselect’ procedure utilised for the implementation of this check whereby an EXCEPT operator is utilised to identify entities which are present in the matrix but do not have at least one SELECT privilege.

Pseudocode 5.2 Algorithm for part of the 'crud_checkselect' process

```
1:   Input: sysdiag_name text
2:   Output: function_name text, result text, classification text
3:
4:   Algorithm:
5:   Difference between:                    >All distinct process in CRUD matrix
6:     SELECT DISTINCT crud_ds_name
7:     FROM spsecurity.crudmatrix
8:     WHERE crud_sysdiag = sn of sysdiag_name
9:   and                                    >All processes with ‘Read’ permission
10:    SELECT DISTINCT crud_ds_name
11:    FROM spsecurity.crudmatrix
12:    WHERE crud_sysdiag = sn of sysdiag_name
13:    AND crud_perm = ‘Read‘
```

2. **Ideally, every relation in the CRUD matrix has at least an INSERT, UPDATE and DELETE privilege.**

These checks, implemented by means of the methods ‘crud_checkinsert’, ‘crud_checkupdate’ and ‘crud_checkdelete’ respectively, are designed to ensure that tuples in database relations may be added, modified, or removed through a security profile privilege. Although the same set operator ‘EXCEPT’ is used to implement these checks, unlike the preceding one, a warning is issued instead of an error message if one of these privileges is missing. This is because in certain circumstances such as in cases where tables are set to be read-only, the lack of one of the permissions does not imply a mistake. Nonetheless issuing a warning to the software developer is still essential since in other cases such permissions are accidentally left out, potentially opening the door to security flaws.

3. **All database tables should be mentioned at least once, in the CRUD matrix.**

This aim of this check is that of verifying that the CRUD matrix encapsulates all relations present in the database. This is a fundamental aspect of the security regime since it ensures that all database objects are accessible in a controlled manner and permissions are granted as required according to the least-privilege principle. This check is conducted via the 'crud_checktables' method, which employs an EXCEPT operator to extract all DFD data stores (or ERM entities) not included in the generated CRUD matrix.

4. **Processes found in the DFD should be included in the CRUD matrix.**

The ‘crud_checkprocesses’ method is responsible for confirming or refuting the prior assertion. In general, database processes are related to database relations, since they either retrieve or store data in them. Nonetheless, in rare cases this might not always hold true; for instance, there could be a scenario where a process simply performs printing tasks. Therefore, a warning is issued to the user, if such processes are found to indicate any spurious processes which might potentially provide loopholes for security vulnerabilities.

Following completion of these checks, the CRUD matrix is expanded to accommodate roles, consistent with the preferred RBAC mode used throughout this dissertation.

5.4.4 Identification of roles and associated privileges from the presented system diagrams

Roles interacting with the system can be deduced from the information present in the system design diagrams, particularly in the DFD. External entities depicted in the latter diagram, represent units that interact with the system of interest but are external to it. Such units might represent individuals, companies, or other systems which either transmit data to the system or receive information from it.

Access to the system's data is exclusively granted to the external entities accounted for in the DFD, allowing for their direct mapping to roles. The sole limitation of such an association lies in its consideration of roles from a generic viewpoint. Despite occupying the same role, distinct users may be permitted access to only a subset of the data. The proposed tool addresses this limitation by allowing each tenant to indicate specific role-based partitions (described in Section 5.6). This approach does not only provide customisation possibilities for tenants, but it also increases the overall security by effectively implementing the least privilege concept at partitioning level. Figure 5.3 depicts the mapping process.

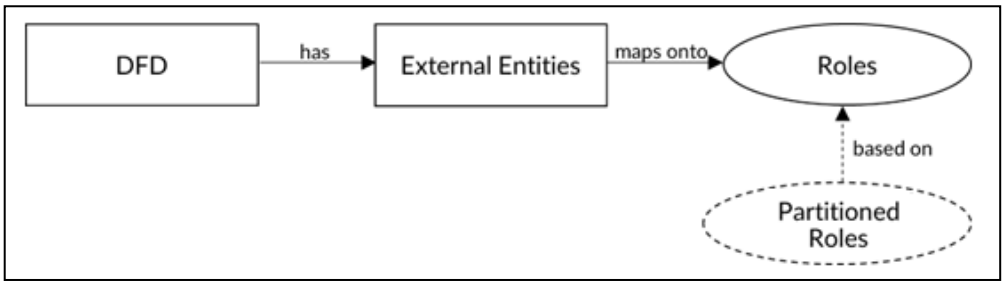


Figure 5.3: Relationship between a DFD's external entities and roles

Following the identification of roles, distinct privileges are assigned to each role. This ensures that access is granted solely to the resources required for each role to execute its job whilst guaranteeing that unauthorised access to sensitive data is prevented. The determination of permissions associated with each role is also dependent primarily on the DFD. As indicated in Check 1 (Section 5.4.3), external entities in a DFD can be connected solely to processes. This implies that roles interact with the system exclusively via processes. Such interaction is desirable not only for

encapsulation purposes but also since allowing direct access to base tables can make the system more vulnerable to unauthorised access and data breaches.

The proposed tool prohibits the use of role hierarchies. The hierarchy mechanism, involves inheriting all attributes and permissions from the parent role. For instance, a ‘Manager’ role might inherit certain access rights from an ‘Admin’ role, allowing the manager to perform administrative tasks. The potential impact of hierarchical changes on multiple users was the primary factor influencing this decision. Moreover, the introduction of additional factors which might negatively impact the overall security of the system also served as a deterrent. Low-level roles, capable of granting permissions to high-level roles, might potentially bypass the normal access control mechanisms via role escalation. Additionally, although role inheritance might simplify the granting of permissions, it can also provide user access to data or capabilities they shouldn't have, based purely on their position in the job hierarchy.

Thus, the proposed tool grants EXECUTE privileges to roles on the processes they are connected to. It is possible to determine whether the role executes or receives data from the process based on the direction of the dataflow linking the external entity with the process. If the dataflow originates from the external entity and goes towards the process, then the role requires execution of the process. Conversely, a dataflow originating from a process and moving towards an external entity, indicates that a role received data from a process.

Consequently, the next stage of the proposed tool re-visits the first level CRUD matrix, developed in Section 5.4. A procedure for expansion is employed, whereby the original matrix specifying the privileges required on tables for each process is expanded to encompass roles. Each process is associated with a set of roles depending on their authorisation level (as per the procedure outlined in the preceding paragraph). Examining the 3D matrix helps in determining which roles are authorised to perform a specific set of operations. Moreover, it ensures that every role granted EXECUTE access on a function has adequate privileges on the objects required by the function, as specified in the original first-level CRUD matrix.

Since the proposed system is based on RBAC, no negative (deny) permissions are allowed. Thus, the expanded CRUD matrix is sufficient in depicting all access rights required for the appropriate functioning of the system. Apart from increasing

complexity, allowing negative privileges can introduce several security risks. Although they may be used to compensate for errors, such as inadvertently allowing unauthorised access to a resource, the underlying security problem is not solved. Moreover, if deny privileges are allowed, the likelihood of introducing security gaps in the system augments. A user may be granted access to multiple roles, having contradicting privileges on the same resources. Additionally, if such privileges are allowed, auditing becomes more difficult since the rationale behind withholding access might be complex. As a consequence, the investigation of security incidents can become notably more challenging, primarily due to conflicts that can arise when multiple negative permissions intersect or contradict one another. Discerning which permission holds precedence in such cases is complex and can potentially result in unintended outcomes.

5.4.5 Implementation of Roles and Privileges

The process of associating roles with processes, is conducted in an automated manner via the ‘role_process’ function. The process is initiated with the creation of a table named 'rolesProcess' consisting of four (4) columns denoting the role name, process name, associated permission, and system identifier. Following that, a list of all processes related to external entities, regardless of the direction of the dataflow, is retrieved and combined into a single set utilising the UNION set operator. An 'Execute' permission is then associated with all processes having an inward dataflow association. Contrarily, those processes with an outward connected dataflow are set to ‘Receive Information’. Pseudocode 5.3 illustrates the pseudocode for extracting processes linked to entities via an inward-outward dataflow.

Pseudocode 5.3 Extracting processes associated with ‘Execute’

```
1:  Algorithm:
2:    SELECT DISTINCT dfd_ee_name, dfd_p_name, ‘Execute’
3:    FROM spdba.dfd_dataflow, spdba.dfd_process, spdba.dfd_external_entity
4:    WHERE dfd_df_inwardentitytype = 'process'
5:    AND dfd_df_outwardentitytype = 'external entity'
6:    AND dfd_p_sn = dfd_df_inwardentitysn
7:    AND dfd_ee_sn = dfd_df_outwardentitysn
8:    AND dfd_df_h_sn = dfd associated with sn of sysdiag_name
```

Subsequently, the ‘insert_rolesProcess’ function is invoked for each tuple in the resultant return set. This function inserts a new tuple in the ‘rolesProcesses’ matrix based on the values provided as parameters and then creates new tuples for each of its parents. This is accomplished by recursively invoking the function for each parent process (identified via the ‘crudmatrixpp’ table), using the same parameters but replacing the process name with the parent process name.

Conclusively, two validation functions are invoked to check the prospective information of the ‘rolesProcesses’ matrix prior to its storage. These functions are implemented to facilitate ongoing security monitoring, even during runtime, with a focus on preserving the overall integrity and validity of the system

These functions are implemented to ensure continuous security monitoring, even during runtime, particularly with respect to the system’s overall integrity and validity. The underlying security principles related to each function are described below:

1. **All roles are connected to at least one function.**

The procedure entitled ‘roles_atleastonefunction’, ensures that there exists at least one function association per role. Such assurance is pivotal when upholding security standards as it helps guarantee that each user has a specified set of responsibilities. If a role is not coupled up with any functions or permissions, managing access control for that role becomes challenging, and there is a risk of granting unintended or unnecessary access to users assigned to that role. Moreover, the definition of roles and their associations with specific functions, helps in enforcing the principle of least privilege.

A set difference operator is used to implement the above check. All external entities, representing roles are retrieved from the DFD. These are subsequently compared with the roles listed in the prospective 'rolesProcesses' matrix and by utilising the set difference operator, any roles that are not mentioned in the matrix even once are identified.

2. **All functions can be executed by at least one role.**

The second check is the reverse of the previous one, as it verifies that all functions within the system are associated with at least one role. Unused processes pose potential vulnerabilities that can be exploited to gain unauthorised access to sensitive information or perform unauthorised actions in the database. Moreover, these unused

processes can consume system resources, resulting in performance issues and potential denial-of-service attacks. Furthermore, associating functions with roles simplifies tracking access and modifications to specific data for accountability purposes. This is crucial for auditing and helps in ensuring that any unauthorised changes are detected.

Similar to the prior check (i.e. Check 1), the main point of reference for this evaluation is the DFD system diagram. All processes are extracted from the DFD, and the EXCEPT operator is used to exclude the processes listed in the 'rolesProcesses' category. By employing this technique, it becomes possible to identify processes that are not mentioned at least once, signifying their absence of association with any function. Provided that both checks are satisfied, the system proceeds to create the 'roleProcesses' matrix, which outlines the associations between roles and the processes they are capable of executing.

5.5 Expanding the CRUD Matrix: Adding a New Dimension

Following the creation of the 'rolesProcesses' matrix, the first-level CRUD matrix is revisited and transformed into a 3D one. This transformation introduces an additional dimension by incorporating roles alongside processes and tables. This is required so that a holistic overview of the system is achieved and implementation of access controls rights and privileges is facilitated. To achieve this objective, the 'roles_process_table' procedure is employed. This process begins by establishing the 3D CRUD matrix, whereby, each role is associated with its respective processes and table permissions. The 'crudmatrix' is then used to retrieve the tables accessed by each process. At this point, only tables are extracted, since views are considered in the following section. The extracted data is then inserted into the matrix through a designated function, allowing it to be utilised for the subsequent assignment of privileges.

A verification check is then performed via the invocation of the 'roles_roleentity' function to ensure that every entity maintains at least one role association. This requirement is vital because an entity that cannot be accessed by any role serves no practical purpose. It becomes futile, leading to potential security vulnerabilities as actions performed on such entities remain unaccounted for. Additionally, the presence of unused entities consumes valuable space without contributing to the system's functionality.

5.6 Horizontal and Vertical Partitioning

Despite the ability of the proposed tool to automatically extract system roles from the presented system design diagrams (explained in Section 5.4.1), it should be noted that not all roles can be uniformly applied to all tenants. Tenants may require lower-level security restrictions due to their varied working environments and different users accessing the system. Consequently, the proposed tool allows tenants to specify horizontal or vertical partitions for each role, which can be leveraged to assign varying data access coverage to users occupying the same role. These partitions are then realised by means of row-level and column-level security restrictions, which ensure that only authorised information can be accessed.

The diagram below illustrates an example of vertical and horizontal partitioning applied to the 'Department' table within the 'Scott' system, for the 'Clerk' role. In this example, 'Clerks' are granted access exclusively to the 'department name' columns, among the three available (as indicated by the vertical partitioning). Moreover, they are limited to data associated with department number 1 (as indicated by the horizontal partitioning).

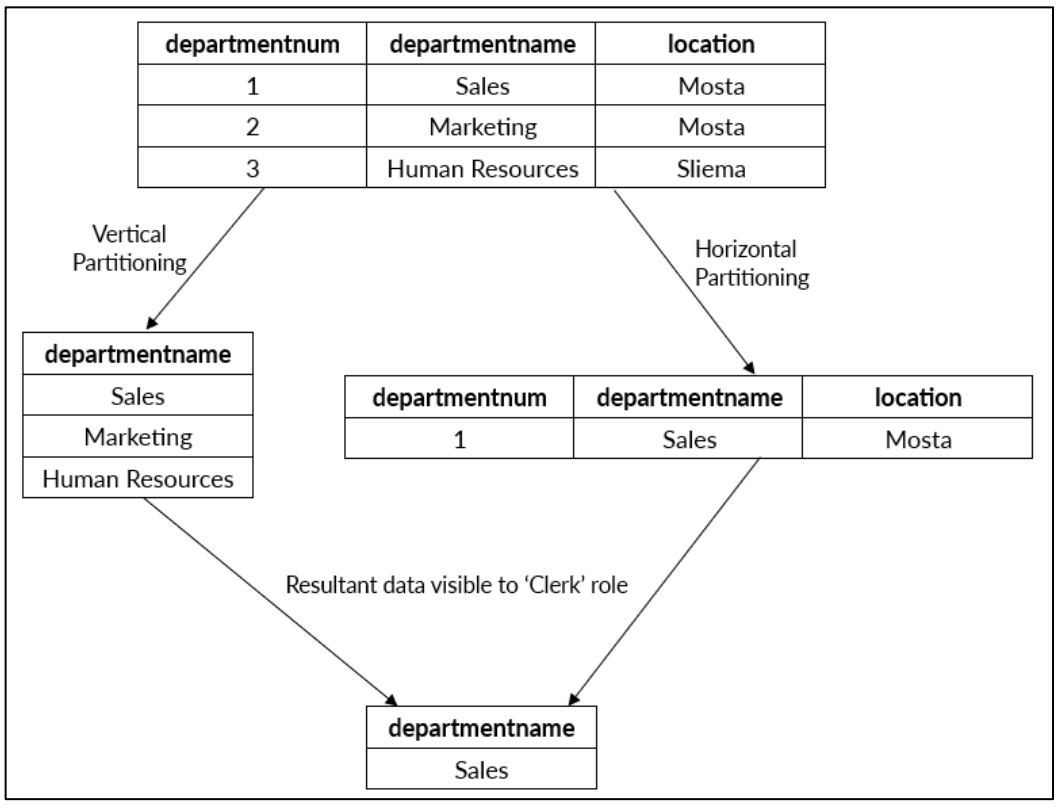


Figure 5.4: Vertical and Horizontal Partitioning on Department Table

Row-level security (RLS) policies are used to implement row-level security in the proposed tool. Such policies are capable of restricting rows based on the currently logged-in user and his role. A supplementary filter associated with a particular role is applied to the table upon which an RLS policy is defined. Once a specific action is conducted by the said user on that table, the defined security filter is implemented prior to returning the expected result set, limiting data accordingly. The direct implementation of RLS policies on base tables, eliminates the need for constructing any additional database objects or structures, thus rendering them advantageous. Moreover, they discretely suppress any row which cannot be viewed or modified by the user, thus ensuring that the user is unaware that such a row exists. Furthermore, RLS access control is centralised in a single location. In most cases, applications developed to anticipate access restrictions only require slight tweaks in response to future access rule changes.

On the other hand, column-level security is implemented by means of views. Such structures provide a straightforward and secure way of achieving security when seeking to exclusively retrieve a predetermined set of columns. Opting for such a technique, does not only obscure inessential information, via the use of ‘INSTEAD OF’ triggers, but it also safeguards data integrity and consistency whilst providing logical data independence. The ensuing section describes how vertical and horizontal partitions on roles are presented to the proposed tool for proper processing.

5.6.1 Encoding and representation of partitioned roles

To maintain consistency with the approach taken in depicting the three system diagrams, EBNF notation is used to define the syntax used during the definition of role partitions. Listing 5.1 depicts the established EBNF for such a representation, consisting of four (4) rules, namely ROLES, ROLE, lstPARTITIONS, and PARTITION.

The first rule, ROLES serves as the overarching structure for role definitions, encompassing both vertical and horizontal partitions. In this notation, a ROLE is defined by the keyword ‘role’ followed by a string representing the role's name. Vertical partitions are defined using the keyword 'verticalPartition,' followed by an array of partition definitions ('lstPARTITIONS' rule). The same is done for the horizontal partitions but these are defined using the keyword ‘partitionBy’. Horizontal partitions fol-

low a comparable structure, involving an array of partitions akin to their vertical counterparts. The key distinction lies in the utilisation of the 'partitionBy' keyword instead of 'verticalPartition' for their definition. Each PARTITION, as indicated by the concluding EBNF rule, is composed of two main elements indicating the table and column name upon which the partition is based. The condition values for each user's horizontal partition are determined during user creation because the rules outlined in Listing 5.1 define the partition structure for generic roles, rather than for individual users. Despite being associated with the same role and partitions, distinct users may possess unique values associated with each partition condition, tailored to their specific needs.

Listing 5.1: EBNF Rules for Partitioned Roles

```
1: ROLES          :=  '{' ROLE ',"verticalPartition":[' lstPARTITIONS '], "partitionBy" :[' lstPARTITIONS ']}'
2: ROLE           :=  '"role":' string
3: lstPARTITIONS  :=  '{' PARTITION '}' [{',{' PARTITION '}'}]
4: PARTITION      :=  '"tablename":' string ', "columnname":' string
```

Similarly, to the encoding of system diagrams, partitioned roles are presented to the proposed tool in JSON format in conformance with the EBNF rules specified in Listing 5.1. The JSON file should include solely those roles with vertical or horizontal partitions associated with them. The rest of the roles in the system (i.e., non-partitioned ones) are extracted from the DFD and retained in their original form. Figure 5.5 provides an example of a JSON file that defines a partitioned role for 'Scott'.

```
{
"name": "Scott",
"roles": [
 {
  "rolename": "Clerk",
  "verticalPartition": [
 { "tablename": "Employee",
  "columnname": "name"
 },
 { "tablename": "Employee",
  "columnname": "job"
 }
  ],
  "partitionBy": [
 { "tablename": "Employee",
  "columnname": "job"
 }
  ]}]
}
```

Figure 5.5: Partitioned role for 'Scott' system

According to the values specified in the vertical partition section, these users' access is restricted to just two columns in the 'Employee' table: 'name' and 'job.' Furthermore, horizontal partitioning based on the 'job' column, enforces additional constraints on user access to specific rows in the 'Employee' table, depending on specific 'job' column values. Further examples can be observed in Chapter 7 and Appendix C respectively.

Each partitioned role file, presented to the tool, is stored in the TEXT column of the ‘import_roles’ table. The ‘is_json’ function is then used to ensure that the text is written in correct JSON format, after which it is stored in a JSON column of the same table. Subsequently, the ‘roles_import2predicates’ function, discussed hereunder is invoked to convert the presented JSON files to relational tables. Before such conversion occurs, a number of several checks are conducted on the data provided. Once the correctness of the submitted information is confirmed, the roles are stored in a separate relation, and the CRUD and roles-processes matrices are updated appropriately, ensuring that the access control system is continually accurate.

5.6.2 Checks conducted on the presented partitioned roles

As indicated in the prior section, the ‘roles_import2predicates’ function commences with the execution of a series of checks on the partitioned roles presented.

1. **All partitioned roles must be present in the DFD as external entities**

The above-mentioned function starts off by invoking another procedure entitled ‘roles_checkRoleExist’. The aim of this function is that of ensuring that all the roles mentioned in the presented JSON file, can be mapped to external entities present in the system. Failure to meet this requirement might compromise the system’s overall security. A DFD specifies role access to particular areas of the database and acts as a blueprint for the security architecture of the system. Thus, if a role specified in the JSON file is not present in the DFD unintentional disclosure of sensitive data is more probable since the data flow to and from the role is not specified. Excessive rights or elevated permissions might be granted, and the system's integrity and confidentiality might be jeopardised.

2. **All presented roles should be associated with at least one partition**

Provided that the first check yields a positive result, every role present in the JSON file is scrutinised to ensure an association with at least one vertical or horizontal

partition. This ensures that the presented JSON adheres to the pre-defined EBNF rules, which indicate that non-partitioned roles should not be included. As indicated earlier, these are extracted directly from the DFD's external entities.

3. **Partitioned table and column name exist in current system**

Subsequently, the third check is performed following the invocation of a separate procedure entitled ‘roles_checkColumnExist’. At the outset, this procedure confirms that all tables specified in the partitioned roles file exist within the current system and are associated with a process which can be executed by the current role. Moreover, it also ensures that the columns listed in both horizontal and vertical partitions are present in the corresponding relation.

The validation checks for user-specific values within each partition, such as those which ensure their compatibility with the data types of the partitioned column, occur at a later stage. This is because, as detailed in Section 5.6.1, these values are inputted after the creation of each user during runtime.

5.6.3 Determining whether required views allow inserts, updates or deletes on the base tables

Not all views specified in the vertical partition segment can be modified. One of the main limitation of views lies in the extent to which they may be altered, which is determined by the structure of the query that composes them. Consequently, in the next phase of this procedure, targets specifically the ‘view-update’ problem. The function performs a series of checks to assess the feasibility of inserts, updates, or deletes based on the view's composition as specified in the ‘vertical partition’ section of the JSON file. The points below, outline the checks performed to determine permissible privileges associated with each view:

Views cannot be modified if [185]:

- The SELECT clause contains aggregates or functions.
- More than one relation is specified in the FROM clause.
- Set operators such as UNION, INTERSECT or EXCEPT are present.
- The following keywords appear: WITH, DISTINCT, HAVING, GROUP BY, LIMIT, OFFSET.

Moreover, a view can only be inserted to, if in addition to fulfilling all the preceding constraints:

- It contains all the fields found in the original table's primary key set.
- The columns not included in the view but present in the main table can be NULL.
- The view definition doesn't have any filters based on attributes not included in the SELECT clause.

While the mentioned constraints can be circumvented with view triggers, the decision was made to refrain from using them in order to prevent undue burden and complexity for tenants. Each check is implemented by means of a specific block of code which analyses the elements associated with each base table in question. A variety of SELECT constructs, union-set operators, and other techniques were employed. When deemed necessary, particularly in cases where the check is more cumbersome, separate procedures are invoked. An example of this is when verifying whether all attributes of the primary key set are found in the view. In this case, the 'erm_pk_attributes' function is invoked as can be noted from the code snippet displayed in SQL Code 5.1. The projected view is then designated as being updateable, insertable, or deleatable based on the outcomes of the previous checks.

```
pkArray = "spdba".erm_pk_attributes(sysdiag_nm, curs.tn);

IF pkarray != ANY(SELECT array_agg(vp ->>'columnname')
         FROM json_array_elements(roles_extent_r->'roles') AS r,
    json_array_elements(r->'verticalPartition') AS vp
  WHERE r->>'rolename' = rec_curs.rn
  AND  vp ->>'tablename' = curs.tn) THEN

canInsertVar = 'false';
canUpdateVar = 'false';
END IF;
```

SQL Code 5.1: Determining whether views can be inserted, updated or deleted

5.6.4 Relationship between roles and views tables

Following successful completion of all checks described in Section 5.6.2, each partitioned role, associated views, and view columns are inserted as tuples into three distinct relations, located in the software provider schema of the tenant database. The 'roles' table, functions as the primary repository for system roles and is linked to the 'views' table through a one-to-many relationship, allowing a single role to be associ-

ated with multiple views based on different tables. This is in turn linked to the 'views-columns' table, which specifies all columns that comprise a single view, together with an indication of whether they are used to split the view horizontally or vertically. In some multi-tenant scenarios, specifically in the ‘separate schema’ and ‘same table’ set-ups, an additional column labelled 'tenant ID' is included in the ‘roles’ and ‘views’ table. The storage of data from multiple tenants within the same relational objects requires the usage of this additional column for security purposes.

Prior to the insertion of roles, views, and view columns, in the appropriate table, a final verification step is carried out on the provided JSON file. This check ensures that when a role has both vertical and horizontal partitioning on the same table, all horizontally partitioned columns are included in the vertical partitioning. Verification of this aspect is crucial since role access is restricted only to the view rather than the underlying base table. Hence, any RLS policies used for horizontal partitioning can be created solely on those columns present in the view. SQL Code 5.2 shows the part of the procedure addressing this issue. The IF NOT EXISTS syntax is used to extract any columns present solely in the horizontal partition section. If the resultant set of this procedure is not null, an exception is raised.

```
FOR clm IN
  SELECT DISTINCT pb ->>'columnname' AS "cnn"
  FROM json_array_elements(roles_extent_r->'roles') AS r,
   json_array_elements(r->'partitionBy') AS pb
  WHERE r->>'rolename' = rec_curs.rn
  AND pb ->>'tablename' = curs.tn

LOOP
  IF NOT EXISTS(SELECT vp ->>'columnname' AS "cn"
     FROM json_array_elements(roles_extent_r->'roles') AS r,
   json_array_elements(r->'verticalPartition') AS vp
     WHERE r->>'rolename' = rec_curs.rn
     AND  vp ->>'tablename' = curs.tn
     AND  vp ->>'columnname' = clm.cnn)
  THEN
  RAISE EXCEPTION 'Columns specified in horizontal partitioning
                   should also be mentioned in vertical partitioning';
  END IF;
END LOOP;
```

SQL Code 5.2: Ensuring that horizontal partitions are present in view

Provided that all the aforementioned checks pass, the roles specified in the presented JSON file are stored in the ‘roles’ relation and the last column of the table,

indicating whether a role is partitioned or otherwise is set to true. Additionally, all distinct tables specified in the partitions section of the JSON file are stored in the 'tablename' column of the 'views' table and linked to their corresponding role using the 'rolename' foreign key column. The remaining details associated with each view tuple, are populated based on the results obtained from the checks explained in Section 5.6.2. Conclusively, new rows are inserted in the ‘viewsConditions’ table, one for every column listed in either the vertical or horizontal partition of the JSON file. These are then associated with the appropriate view. Once all partitioned roles are processed, the ‘roles_import2predicates’ function, incorporates the remaining unspecified external entities in the partitioned roles file into the 'roles' table. Given that these were not included in the JSON file, it is presumed that they are not partitioned, hence the 'partitioned' attribute value is set to false.

5.6.5 Update of CRUD to reflect partitioned roles and views

The inclusion of partitioned roles also impacts the expanded CRUD matrix outlined in Section 5.5. This matrix has previously been updated from a first-level CRUD matrix depicting processes and tables permissions to one with another dimension which incorporates roles. The latter matrix is subsequently re-visited to encapsulate the utilisation of views, presented in this section. The paragraph below outlines the procedure followed for this matrix to be updated.

Since the expanded CRUD matrix (i.e., 'rolesprocessestables') is built upon the first-level matrix (i.e., 'crudmatrix'), the latter table was first updated. The table is iterated over and the processes which require access to the table upon which the view is based, are extracted. Subsequently, the ‘insertintocrud’ function is invoked, and a set of tuples equivalent in number to those extracted are inserted in the ‘crudmatrix’ relation. Despite having the same process name, the newly inserted rows are now linked to the view rather than the underlying base table. Thus, the value inserted in the ‘tableOrViewName’ column reflects the view’s name whilst that inserted in the ‘tableOrView’ column is set to ‘View’. The 'rolesprocessestables' relation is then modified. All tuples having information related to the partitioned role, and the table upon which the role is partitioned, are updated. Specifically, the associated table name is changed to the view name, and the type is updated from 'Table' to 'View'. In Table 5.2 below,

the updated tuples in the 'rolesprocessestable' matrix are displayed, reflecting the insertion of the partitioned role (illustrated in Figure 5.5) within the 'Scott' scenario. In this example, vertical partitions were configured for the 'Employee' relation, granting visibility of only the 'name' and 'job' columns to roles of type 'Clerk'.

Table 5.2: Updated ‘rolesprocessestable’ following partitioned role insertion

rpc_ role_ name	rpc_process_ name	rpc_process _perm	rpc_table_name	rpc_ table- orview	rpc_ table_ perm
Clerk	startWork	Receive Information	Clerkscott_sd_Employee	View	INSERT
Clerk	modifyEmployeeDetails	Execute	Clerkscott_sd_Employee	View	UPDATE
Clerk	leave	Execute	Clerkscott_sd_Employee	View	DELETE
Clerk	generateEmployeeRport	Execute	Clerkscott_sd_Employee	View	SELECT

It can be observed that four (4) rows were impacted, specifically those indicating the privileges the ‘Clerk’ has on the ‘Employee’ table. In these cases, the name was changed from ‘Employee’ to ‘Clerkscott_sd_Employee’ (i.e., the view name) whilst the type was changed from ‘Table’ to ‘View’. The rest of the information was left intact.

5.7 Software Provider Information

While the majority of the information used to construct the operational security profile is provided by the tenant, there is a crucial set of information supplied by the software provider. This specific dataset includes a collection of tenant tables, functions, and stored procedure codes that are common to all database systems created using the proposed tool. These encompass essential relationships and functions necessary for the accurate operation of the tool's automated mechanisms, such as the 'insert_user' function. Some of these database elements, particularly those related to views, roles, and other aspects within the software provider's domain, are stored in the 'software provider' schema within the tenant database. The remaining components, such as those associated with users and user conditions, are stored in the 'tenant' schema of the tenant database, to allow for direct access by the database administrator of the respective tenant.

Even though the information mentioned in the prior paragraph is provided by the software provider and not by the tenant, for consistency purposes, the same specification approach used in other sections of the tool is applied. Therefore, a set of EBNF rules were developed to delineate the structure of tenant tables (Listing 5.2) and functions (Listing 5.3).

Listing 5.2: EBNF Rules for Tenant Tables

```
 1:  TENANTTABLES   :=  '{' NAME ',"attributes":[' lstATTRIBUTE '], primarykeys":[' lstPRIMARYKEY '],"unique":[' lstUNIQUE '],"notnull":[' lstNOTNULL '],"foreignkeys":[' lstFOREIGNKEY '],' TENANT_OR_SP '}'
 2:  NAME           :=  '"name":' string
 3:  lstATTRIBUTE   :=  '{' ATTRIBUTE '}' [{',{' ATTRIBUTE '}'}]
 4:  ATTRIBUTE      :=  '"attributename": ' string ', "attributetype": ' string
 5:  lstPRIMARYKEY  :=  '{' PRIMARYKEY '}' [{',{' PRIMARYKEY '}'}]
 6:  PRIMARYKEY     :=  '"primarykey":' string
 7:  lstUNIQUE      :=  '{' UNIQUE '}' [{',{' UNIQUE '}'}]
 8:  UNIQUE         :=  '"uniqueattribute":' string
 9:  lstNOTNULL     :=  '{' NOTNULL '}' [{',{' NOTNULL '}'}]
10:  NOTNULL        :=  '"nnattribute":' string
11:  FOREIGNKEYS    :=  '{' FOREIGNKEY '}' [{',{' FOREIGNKEY '}'}]
12:  FOREIGNKEY     :=  '"foreignkey":' string '-' string '.' string '(' string ')'
13:  TENANTORSP     :=  '"tenantorsoftprov":' ' "tenant" | "softwareprovider"'
```

As can be noted from Listing 5.2, a total of thirteen (13) EBNF rules were developed to specify the tables' structure. Curly brackets are used in the topmost rule to enclose the elements needed when constructing a table. These encompass the table name, attributes, primary keys, foreign keys, as well as column-specific constraints pertaining to uniqueness and nullability within the tenant's tables. The attributes are then defined in a separate list rule, composed of multiple elements separated by commas. Each attribute element is composed of both the attribute's name and its type.

Additionally, a separate rule is used to specify the primary key of the table. Given that the primary key can be made up of more than one attribute, it is constructed as a list of column names, of type string, enclosed within curly braces and separated by commas. Unique and required i.e. not null, attributes are specified analogously to primary key ones. The only distinction is in the tag which precedes the elements of that category. As opposed to the 'primarykey' tag, used in the specification of primary keys, string values denoting unique or not null attributes are preceded by

‘uniqueattribute’ and ‘nnattribute’ respectively. Following the definition of the aforementioned rules, another rule specifying the foreign key list is constructed. This list is composed of a string which encapsulates the name of the foreign key column in the current table together with the referenced table and column. The last rule is composed of two alternative options, divided by the vertical bar ‘|’ which indicates whether the table is related to ‘softwareprovider’ or ‘tenant’ schema.

Listing 5.3: EBNF Rules for Tenant Functions

```
1:  TENANTFUNCTIONS   :=  ‘{‘ NAME ‘,“parameters”:[‘ lstATTRIBUTE ‘],’ RETURN ‘,
                          “declare”:[‘lstDECLARE‘],“body”:[‘lstBODY‘],’ TEN-
                          ANT_OR_SP ‘}’
2:  NAME              :=  ‘‘“name”:’ string
3:  lstPARAMETERS     :=  ‘{’ PARAMETER ‘}’ [{‘,{’ PARAMETER ‘}’}]
4:  PARAMETER         :=  ‘‘“parametername”:’ string ‘, “parametertype”:’ string
5:  RETURN            :=  ‘‘“return”:’ string
6:  lstDECLARE        :=  ‘{’ DECLARE ‘}’ [{‘,{’ DECLARE ‘}’}]
7:  DECLARE           :=  ‘‘“variablename”:’ string ‘, “variabletype”:’ string
8:  lstBODY           :=  ‘{’ BODY ‘}’ [{‘,{’ BODY ‘}’}]
9:  BODY              :=  ‘‘“line”:’ string
10: TENANTORSP        :=  ‘‘“tenantorsoftprov”:’ ‘“tenant” | “softwareprovider”’
```

The first rule displayed in Listing 5.3 outlines the overall structure of a function, which comprises its name, input parameters, return value, declare section variables, a multi-line body, and an indication of the schema to which the function is related. Separate rules are then used to define the structure of the function’s name and return types, consisting of a single element of type string. Additionally, two rules are developed to define the structure of input parameters and variables declared within the function. Each parameter or variable is comprised of two string elements, indicating its name and type. The sole differentiation between the two is the utilisation of distinct tag descriptors. For parameters, the tags 'parametername' and 'parametertype' are employed, whereas for variables, the tags 'variablename' and 'variabletype' are used. Both rules allow the definition of several components of the same kind, as can be noted from the utilisation of square brackets '[]'.

Another rule is then created to define the function's body as a string of multiple lines, concluding with the last rule which identifies the potential storage location (schema) of the function. The 'tenantorsoftprov' descriptor precedes the schema

name, which can be one of two options (indicated by the '|' symbol), representing the software provider or tenant schema.

Following development of the aforementioned rules, the generic tenant tables and functions required for the proposed tool were formulated in adherence with these rules and stored in two JSON files named 'ListOfTenantTables' and 'ListOf-TenantFunctions' respectively. In addition to complying with these rules, all functions prioritise adherence to the procedures outlined later on in Section 5.9 to mitigate potential threats. The files are then imported into the TEXT column of either the 'import_tenanttables' or 'import_tenantfunctions' relation and transferred to a JSON column of the same table following validation. Two processes, namely 'import_tables2predicates' and 'import_functions2predicates,' are utilised to convert JSON-formatted files into relational tuples. These tuples are then stored in two distinct relations: 'tenant_tables' and 'tenant_functions.' 'tenant_tables' contains information specific to tenant tables, including serial numbers, table names, attributes, primary keys, indication of non-null columns, foreign keys, and a field indicating whether the table resides in the software provider or tenant schema. 'tenant_functions,' on the other hand, holds data pertaining to tenant-specific functions, featuring columns for serial numbers, function names, function parameters, return values, declarations, function bodies, and an indicator denoting whether the function is located in the software provider or tenant schema.

Similar concepts are employed in both functions used for the population of the above-mentioned relations. The presented tables or functions are processed using a FOR-loop. During each iteration, a series of SELECT statements, combined with the '->' JSON structure, are utilised to extract the individual elements comprising each table or function. These are then merged into a tuple and inserted into the appropriate relation among the two previously mentioned options.

5.7.1 Tenant Tables and Functions

This section provides an in-depth explanation of these crucial database objects, highlighting their purpose and relevance concerning the tool's functionality. Appendix B.1 and B.2 present a tabulated list of all tenant tables and functions utilised by the proposed tool, along with their respective storage schemas.

The tenant tables include 'roles', 'views', and 'viewscolumn', which specifically support the process outlined in Section 5.6.4. Additionally, the 'user' and 'userconditions' tables are used to store information about users associated with a particular tenant. They also store any conditions relevant to the allowed partitions on the roles they are linked to. Further details regarding the attributes associated with each relation and the interconnections between these tables are available in the ERM presented in Figure 5.6 below.

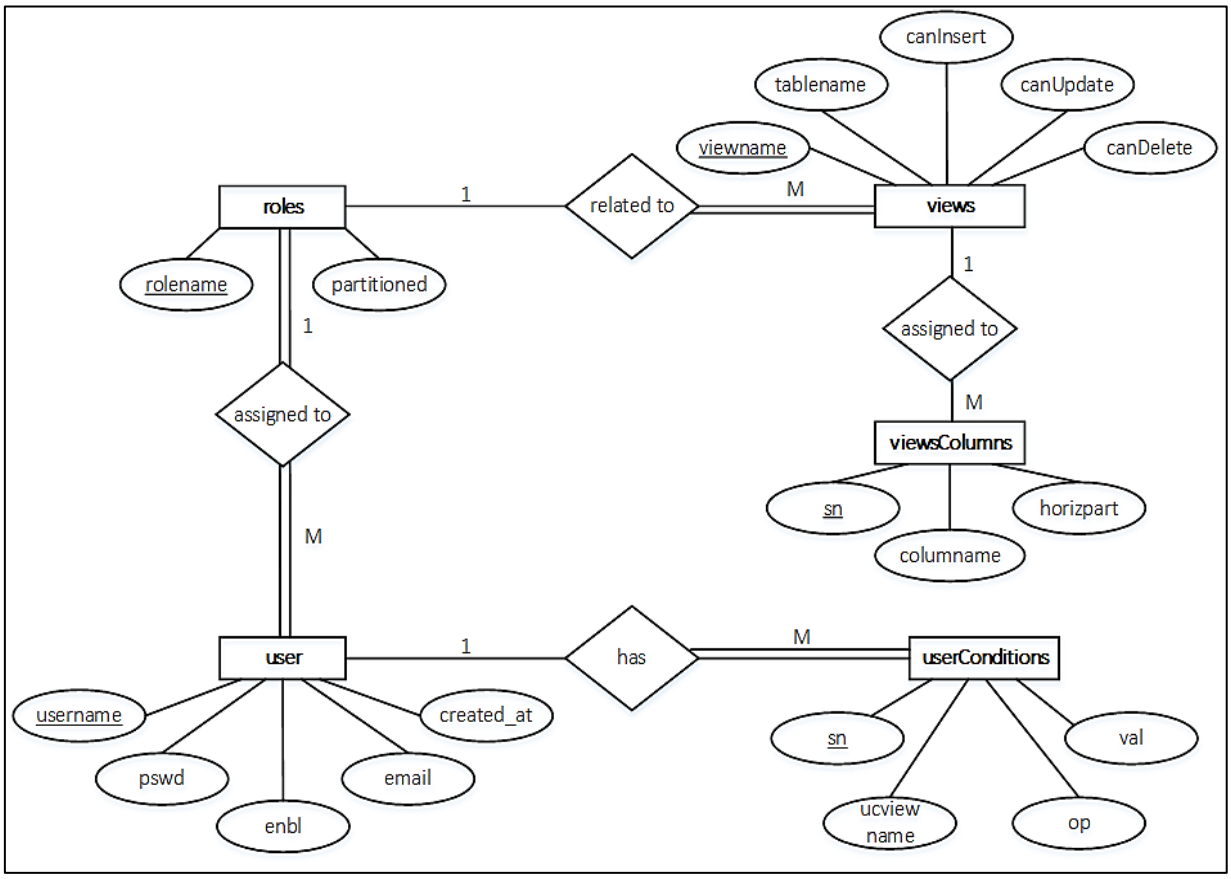


Figure 5.6: ERM depicting interrelations among tenant tables

On the other hand, the tenant functions intended for the proposed tool can be classified into three primary categories. Firstly, there are functions associated with inserting, updating, and deleting data in the prior-defined tables. Secondly, there are 'getter' functions employed for convenient retrieval purposes, aiming to circumvent direct table access and facilitating GUI functionality. Lastly, there are functions directly linked to security-related operations. The categories and descriptions of each function are outlined below.

5.7.1.1 Functions directly related to the table

1. **user_creation** and **user_deletion**

The task of the ‘user_creation’ function is that of creating users and assigning them to roles. Since users in PostgreSQL are implemented as roles with login, the CREATE ROLE statement with login option is used, and the IN-ROLE option is added, indicating the user's membership in a certain role. Conversely, the latter function is responsible for the deletion of a particular user. Such a process is performed by means of the DROP ROLE SQL construct.

2. **insert_user, update_user** and **delete_user**

These functions are responsible for inserting, updating, and deleting user records respectively from the ‘user’ table. While each of the three functions' input arguments contain the username and matching role of the user in question, the 'insert user' and 'update user' methods additionally include other input parameters specifying the actual values to be entered or modified. Prior to every modification or deletion, a query is performed to ensure that the username and role name supplied match those of an existent database user. In the case, of the ‘insert_user’ method, a different query is required to ensure that prior to associating a new user to a particular role, the role in question has already been created. If the result of the SELECT statement is an affirmative one, a user row is inserted, modified or deleted accordingly. Additionally, in the case of the ‘insert_user’ and ‘delete_user’ methods, the ‘user_creation’ and ‘user_deletion’ methods, described prior are invoked so that the actual user instance is created or deleted.

3. **insert_usercondition, update_usercondition** and **delete_usercondition**

These three functions are responsible for managing the insertion, updating, and deletion of tuples associated with user-specific conditions. Prior to the insertion or updating of a condition, several validations are performed to ensure data integrity. These include verifying that the associated user already exists, confirming the existence of the specified 'tablename' and 'columnname' in relation to the user, and validating the compatibility of the operator and value with the specified column’s data type. If all three checks evaluate to true, then a new record is inserted or updated accordingly in the ‘userconditions’ relation. In contrast, the 'delete_usercondition'

method is designed to exclusively delete the tuple linked to the inputted tenant identification from the 'userconditions' table, provided that it exists.

5.7.1.2 Functions used for easier access

1. **insert_csvdata**

   This procedure is implemented to simplify operations and enhance accessibility. It allows tenants to input data in tables in batch, rather than individually inserting each row of data. The COPY command is utilised to extract the data from the .csv file, specified by the tenant and insert it into the relevant relation.

2. **getRoles**

   This function is in charge of retrieving all roles found in a particular database system. It is integrated into the tool's graphical user interface (GUI) and serves as a valuable tool for the tenant’s DBA when managing user-role assignments.

3. **getPartitions**

   This function is responsible for retrieving all horizontal partitions (if any) associated with a particular role. It is particularly useful when assigning row-level security conditions to individual users and requires only a single input parameter, representing the specific role being considered. The provided argument serves as the basis for the WHERE condition in the SELECT query, enabling the retrieval of all corresponding partitions (table and column) from the 'views' and 'viewsColumns' tables respectively.

4. **getProcesses**

   The aim of this function is to retrieve all functions which can be executed by a particular user. The tool's GUI serves the purpose of limiting a specific tenant user's visibility and selection of processes to those they have authorised access to. Moreover, it plays a broader role in the overall security verification process, ensuring that only permissible processes are linked to each user.

5. **getProcessParams**

   This function is responsible for the retrieval of all parameters (both name and data type) associated with a particular process. Implemented mainly as a SELECT query, this function enhances the accessibility of the GUI by ensuring that tenant users are presented with limited options for data entry, restricted to the allowed values within the selected process parameters.

5.7.1.3 Functions required for security purposes

1. **login**

Tenants' users must be able to log in to the system, to ensure that they are granted access solely to the essential resources required for their work. In the proposed tool, a password authentication mechanism is utilised for this purpose, where users are identified by means of a corresponding username and password.

Consequently, the aim of this function is to validate the correctness of the provided username and password for a specific tenant user by checking whether an enabled corresponding record exists in the tenant's 'user' table.

2. **enable_tenantusers** and **disable_tenantusers**

Enabling and disabling tenants is another crucial aspect of multi-tenant security. Tenant DBAs should possess the capability of performing this task for all users associated with their respective tenant. Maintaining an enabled status for a user who has opted out of the system leaves a backdoor open for any attackers trying to get access and thus, should be avoided at all costs.

In this context, these functions hold the responsibility of enabling or disabling tenant users selectively. After verifying the existence of the user provided in the function's input parameter, the 'enbl' column in the 'user' table associated with the specified user is updated to true or false, as appropriate.

5.8 Tenant Specifications

To enhance the tool's adaptability and accommodate tenant-specific customisations in the security profiles of individual tenants, the proposed tool requests the provisioning of an additional file titled 'Specifications'. This file gives tenants the ability of identifying and accentuating specific security-related requirements, relevant to their individual needs. The identified requirements are subsequently integrated during the generation process of the final SQL code. To maintain consistency with other inputted information, particularly that contained in the system design diagrams and partition files, the 'Specifications' are structured in JSON format and developed in a manner which adheres to a predetermined set of EBNF rules (displayed in Listing 5.4).

The software provider retains the authority to modify the aforementioned set of rules, as deemed fit. As discussed in previous sections, the main advantage of this

approach, stems from its ability to easily incorporate additional rules which might be required with future updates. A total of four directives, were considered for this dissertation. In fact, the initial EBNF rule, which outlines the overall structure of the requirements, is made up of four key elements. A comprehensive discussion of each requirement and its relationship to security is presented hereunder:

Listing 5.4: EBNF Rules for Specifications

```
1: REQUIREMENTS     :=  '{' MTSCENARIO ',' DBENCODING ',' AUTHENTICATION ','
                        PSWDVALID '}'
2: MTSCENARIO       :=  '"multitenantscenario":' ' "separate database" | "separate
                        schema" | "same table"'
3: DBENCODING       :=  '"databaseencoding":' string
4: AUTHENTICATION   :=  '"authentication":' string
5: PSWDVALID        :=  '"passwordvalidation":' dateandtime
```

5.8.1 Type of multi-tenancy required

Tenants are allowed to choose the specific type of multi-tenancy approach to be adopted. Section 2.7 provides a detailed discussion of the three primary multi-tenancy approaches, namely separate database, separate schema, and same table. The selection of one approach over the other has a direct impact on the functions invoked whilst generating the final SQL code. This is because, as explained previously, the variations in the mode of data storage employed by each approach, necessitate distinct security considerations. Specifying the preferred type of multi-tenancy approach is mandatory and should be done in accordance with the second EBNF rule displayed in Listing 5.4. According to this rule, the preferred approach for implementing multi-tenancy should be specified using the 'multitenantscenario' tag descriptor and must be one of three possible strings: "separate database", "separate schema", or "same table".

5.8.2 Database encoding

Tenants opting for a 'separate database' multi-tenant scenario are also given the option of specifying the database encoding utilised during the database creation. In the case of 'separate schema' or 'same table' approaches, the database encoding cannot be specified. In these scenarios, the database already exists since it is created once at the beginning by the software provider and shared with other tenants.

The database's encoding type is responsible for determining how text data is stored and managed. Consequently, albeit indirectly, the value of this variable has a bearing, on the overall security of the database system. Unless proper sanitisation is performed, some encoding types such as ‘SQL_ASCII’ are more prone to SQL Injection attacks due to their lack of input restriction or validation. Conversely, other encoding types, such as 'UTF8' are generally considered more secure due to their ability to resist buffer overflow attacks and handle a wider range of characters. Another security aspect impacted by the type of database encoding selected, is encryption. Some encoding types are incompatible with specific encryption techniques, whilst others might produce weak encrypted values due to their limited capabilities when handling a wide range of characters [186].

In PostgreSQL, numerous databases encoding formats are available [187]. Those opting for a ‘separate database’ scenario have the option of selecting any encoding type based on their preference.

The preferred type of database encoding should be specified as a string following the ‘databaseencoding’ tag descriptor, in accordance with the third EBNF rule displayed in Listing 5.4.

5.8.3 Password Authentication and Expiration

Another essential aspect of security, indicated in Chapter 3 is authentication. Although PostgreSQL provides multiple authentication mechanisms, such as Trust, GSSAPI, Ident, Peer, PAM, and BSD [168], the one utilised in the proposed CASE tool is password authentication. The decision to opt for this mechanism was primarily based on the fact that password authentication is the most straightforward option when dealing with remote connections. All alternative options, either necessitate some form of external security infrastructure, such as an authentication server or SSL certification, or are tailored to a specific platform [168]. Nonetheless, this decision can be easily modified in future updates to utilise different authentication mechanisms in accordance with the software provider's preferences.

PostgreSQL offers two primary password-based authentication methods for stored passwords - SCRAM-SHA-256' and 'md5' [167]. Albeit having the same objective, these approaches have varying mechanisms related to the transmission and stor-

age of data. The utilisation of ‘SCRAM-SHA-256’ authentication, ensures secure storage of passwords on servers by means of a cryptographically hashed format. Despite being the most secure approach from those present in PostgreSQL, the downside of the approach is its non-compatibility with older client libraries. In comparison, the 'md5' method utilises a customised mechanism that is slightly less secure but still offers sufficient protection. This decrease in security is attributable to the fact that passwords might be compromised if an attacker gains access to the password hash [167].

The proposed tool provides tenants with the ability to choose one of the two previously discussed password authentication modes, selecting the one that is most fitting for their needs. The chosen method should be specified in accordance with Rule 4 outlined in Listing 5.4. This entails utilising an 'authentication' tag followed by the corresponding string identifier of the preferred password authentication method.

In the context of password authentication, it is critical to implement regular password changes to enhance the security of the database system. Passwords can get compromised over time as a result of theft or hacking efforts, leaving data open to unauthorised access. Modifying passwords on a regular basis, helps in diminishing the span of time an attacker has, to exploit a compromised password. Furthermore, it minimises the likelihood of attackers successfully accessing the database systems by means of a stolen password [188].

For this reason, the 'VALID UNTIL' keyword followed by a specific date and time is appended to all passwords used in any of the databases generated by the proposed tool. Upon expiration of this duration, passwords must be changed to ensure continued protection of the database. Tenants are allowed to specify the duration of password validity utilising the final rule stipulated in Listing 5.4. This rule indicates that password expiration dates should be preceded by the ‘passwordvalidation’ tag and specified as ‘dateandtime’ format.

5.8.4 Checks conducted of tenant specifications

A set of checks were subsequently developed to ensure that the specification values imputed are compliant with the guidelines specified in Section 5.8. The initial verification focuses on the 'multitenantscenario' element, confirming that it corresponds to

one of the three available options. If these conditions are not met, an error message is displayed, and the program execution is halted.

Further checks are conducted on the encoding, and authentication elements to ensure that the specified options correspond to the available choices within PostgreSQL. However, if these conditions are not met, the program issues a warning rather than an error. This flexibility accounts for the possibility that the software provider may utilise alternative extensions or different database systems with distinct encryption, encoding, or authentication methods. Additionally, a check is performed on the second requirement, specifically 'encoding', to verify that it is specified only when the preceding requirement has been set to 'separate database'. In the two other types of multi-tenant scenarios, the database is created only once at the initial stage by the software provider and remains constant for all tenants throughout.

Lastly, the final check pertains to the date inputted after the 'valid until' tag. It verifies not only the correct date format (of type 'date') but also ensures that the date is set in the future, as it is not permissible to set a password expiration date in the past. An illustrative example of the requirements file, featuring potential values for each element, is presented in the Case Study Chapter.

5.9 Addressing Threats

Prior to initiating the design and development of the scripts used during the generation of the final SQL report, deliberate analysis was conducted. The objective was to ensure that the construction of these scripts effectively mitigates a wide spectrum of security threats, particularly those directly linked to multi-tenant database systems. This section provides a comprehensive overview of the key mechanisms employed to counteract each of these threats.

1. **Compromised authentication**

The inclusion of the 'VALID UNTIL' syntax in all password-handling scripts ensured the establishment of a robust and effective mechanism that enhances authentication security. The PostgreSQL syntax (depicted in SQL Code 5.3) mitigates compromised authentication risks by allowing tenants to set expiration dates for user ac-

counts or passwords. Once the specified expiration date is reached, access is automatically revoked, thereby diminishing the risk of unauthorised access and misuse due to the constant password updates.

```
CREATE ROLE clerk
LOGIN
ENCRYPTED PASSWORD 'clerk_pswd'
VALID UNTIL '2023-10-12';
```

SQL Code 5.3: Password VALID UNTIL syntax

2. **SQL Injection (SQLi)**

SQL injection attacks involve the malicious insertion of data into SQL queries using destructive commands or unescaped parameters [189]. Hackers exploit this method to illicitly gain unauthorised access to sensitive data and potentially manipulate or delete it. During the development of SQL scripts, several steps were undertaken to counteract or mitigate such attacks.

As a primary measure, inputs from SQL statements underwent a validation or sanitisation process before being transmitted to the database. This was accomplished by utilising stored procedures that employ parameterised SQL statements, effectively segregating the query statements from the accompanying data. Parameterised statements within the stored procedures make use of stored queries with placeholders, allowing for the input data to be securely substituted. Rather than treating the entire query and data as a cohesive string, the database exclusively interprets the stored query as a query language, considering user inputs as a separate collection of parameters that are handled strictly as data. SQL Code 5.4 presents an example of a parameterised statement utilised within the 'Create Schema' script.

```
EXECUTE FORMAT('CREATE SCHEMA IF NOT EXISTS %I AUTHORIZA-
 TION %I;', schemaname, schemaowner);
```

SQL Code 5.4: Parameterised Query

The usage of the '%I' placeholder within the FORMAT() function is employed to represent an identifier, such as a schema or table name (in this context, the former), ensuring that it undergoes appropriate quoting and escaping to effectively prevent SQL injection attacks. By employing the '%I' to replace the schema name and owner in this dynamically constructed SQL query, any potentially harmful input is treated solely

as a literal identifier rather than executable SQL code. This provides protection against SQL injection attacks. One additional technique which assists in the prevention of SQL Injection attacks is the implementation of the least privilege concept. This approach minimises the impact of a successful SQL injection attack by limiting the privileges of database users, thus restricting access to only the required objects and operations. For instance, if a user is granted read-only access, even in the event of a SQL injection vulnerability exploitation, the attacker's capabilities would be limited.

The third countermeasure against SQL injection is the implementation of encryption. Sensitive data like passwords, undergo an encryption process to enhance security. While encryption does not directly prevent SQL injection attacks, it indirectly mitigates their potential damage by limiting the value of extracted data. Lastly, the final technique employed to prevent SQLi attacks is monitoring and auditing. An auditing script is used to track all activities conducted on the database. These measures significantly contribute to the efficient identification of unauthorised SQL statements and potential vulnerabilities. Upon identifying such issues, administrators can promptly take appropriate actions, such as deleting or disabling unnecessary accounts, thus reducing the risk of malicious code being injected and executed.

3. **Trojan Horse**

Trojans can be employed by authorised users to perform a range of malicious actions, including deleting, blocking, altering, or duplicating data, while also disrupting the normal operation of computers. An effective countermeasure which can be implemented to mitigate Trojan horse attacks involves the implementation of audit trails. These logs capture complete details of database activities, encompassing user actions, system events, and data modifications, thereby facilitating the detection of suspicious activities. Subsequent analysis of these records enables the identification of irregular patterns, unauthorised access attempts, or unexpected alterations to database objects. The proposed tool implements another countermeasure by enforcing restrictions on users from creating new objects in the PUBLIC schema of the Postgres instance. This limitation serves to minimise the risks of introducing malicious elements or granting unauthorised access to user-generated objects within this schema.

4. **Loose backups or logs**

One common oversight related to database security is the handling of loose backups or logs. Implementing tight security access control mechanisms becomes futile if the corresponding practices concerning PostgreSQL backups or logs are disregarded. In certain instances, backups or audit logs are co-located within the schema of the primary data, thereby increasing the exposure to a wider range of individuals with access privileges. Consequently, this heightens the vulnerability of sensitive information, including the risk of exposing plain text passwords stored in audit files.

To mitigate this risk, the scripts utilised in the proposed tool ensure a clear separation of database backups and logs from the main data. Specifically, log files are stored within the security schema, while the created backup files are intended to be stored on a separate server by the tenant. Furthermore, to further minimise this potential threat, access to the security schema, where the audit files are situated, is restricted solely to tenant DBAs in the proposed tool.

5. **Disk space attacks**

These types of attacks involve activities conducted with the intention of depleting or exhausting the available disk space on a system or storage device. These attacks pose a significant risk as they can lead to various consequences, such as system slowdown, unavailability of disk space for legitimate operations, the creation of denial of service (DoS) conditions, and potential system crashes. In PostgreSQL, such attacks are easier to conduct particularly because all users with login privileges, including unprivileged ones, can create objects in the default PUBLIC schema by default. This feature's convenience can be exploited, for example by populating a table with a large number of rows (e.g., by using the ‘generate_series’ function). This can exhaust resources like disk space, potentially causing a crash in the PostgreSQL system [166].

To mitigate these types of attacks, it is recommended that application-specific schemas are created so the use of PUBLIC schemas is minimised. The proposed CASE tool follows this recommendation by establishing distinct schemas for tenants, software providers, and security purposes. The number of schemas created is contingent upon the specific type of multi-tenant scenario being used. Another measure employed by the proposed tool to counteract this attack is that of removing the default CREATE rights on both the PUBLIC schema and the PUBLIC role. In fact, the final SQL

Code generated by the tool commences with the revocation of all privileges associated with the PUBLIC schema and role, as demonstrated in SQL Code 5.5.

```
REVOKE ALL ON DATABASE databasename FROM PUBLIC;
REVOKE ALL ON SCHEMA PUBLIC FROM PUBLIC;
```

SQL Code 5.5: Revocation of rights from PUBLIC schema and role

Furthermore, the proposed tool revokes access to the PUBLIC role on all schemas upon their creation, and it also enforces the immediate removal of access to the PUBLIC schema for newly created roles.

5.10 Summary

This chapter focuses on the foundational elements of access control within the system. It begins with an examination of the initial phases of access control, with particular emphasis on role-based access control and the critical importance of establishing a secure environment. Subsequently, the process employed to extract parent-child process associations is outlined, exploring their significance in shaping access permissions. At the core of access control lies the first-level CRUD matrix, introduced to depict the permissions required by processes on tables. This introduction is followed by a comprehensive examination of CRUD matrix generation, along with a detailed discussion of the necessary validation checks implemented post-creation. This matrix is subsequently extended to incorporate an additional dimension—roles, offering the flexibility for generic roles or horizontally and vertically partitioned ones. The integration of these roles into the initial CRUD matrix is explored, along with an explanation of any required validation checks.

Moreover, this chapter specifies software provider information related to access controls, with a specific focus on roles, views, user tables, and functions. It also explores the concept of tenant specifications, enabling customisations to tailor security profiles to individual requirements. Lastly, the chapter addresses the pivotal topic of threat mitigation, highlighting the integrated measures within the proposed tool development to effectively counter potential threats. With these foundational security measures in place, the subsequent chapter outlines the development of SQL scripts, providing a comprehensive guide for their implementation.

6 Implementation and Development of SQL Scripts

6.1 Introduction

This chapter builds upon the foundations laid in the previous two chapters and shifts the focus to the implementation stage. Specifically, it focuses on the development of the deliverable, which consists of SQL-generated code for each tenant, accompanied by the associated security assertions.

It begins with a list of post specifications, ensuring the system's security through the proposed approach. It then delves into the discussion of essential functions necessary for establishing a comprehensive multi-tenant secure database system, providing detailed insights into their implementation. Moreover, the chapter underlines the intricacies associated with the multi-tenant scenario under consideration.

6.2 Post specifications

Prior to the automated generation of the conclusive SQL report and security profile for each tenant, some final checks are conducted to discover potential security design flaws and identify opportunities for design enhancements. These additional assessments go beyond the ongoing checks carried out throughout the process, ensuring a comprehensive evaluation of the generated system's database.

The set of rules specified in this section served as the core foundation around which these checks were centred. Given that the primary aim of the proposed tool in this dissertation, is that of generating secure profiles for each tenant, the identified rules primarily emphasise security-related aspects.

1. **Reachability**

Each object in the database must be reachable, in the sense that all objects in the database must be used or reached by at least one process. This necessity arises from the system's explicit restriction against direct user access to database objects, mandating access exclusively through processes. Ensuring reachability not only prevents space wastage but also mitigates significant security risks by addressing potential misconfigurations and access control oversights. Processes rely on user execution,

so, in addition to ensuring database objects are accessed through processes, it's vital to confirm that all processes are linked to at least one role. A standalone process devoid of the capability for execution by any user or role serves no purpose.

The first step in this verification process entails selecting all database objects, including tables and views, and confirming their association with at least one process. Subsequently, the next phase involves verifying that these processes can be executed by at least one role. This is achieved by selecting all roles and ensuring that at least one of them possesses the capability to execute each respective process.

2. **Accessibility**

Accessibility is a crucial factor that sets itself apart from reachability by addressing how database objects are accessed, particularly in terms of allowable actions. It emphasises the necessity of granting SELECT privileges to every relation or view in the database, while also considering the desirability, though not mandatory requirement, of providing INSERT, UPDATE, and DELETE privileges. A SELECT privilege is indispensable for an object as its absence renders the object inaccessible and useless. Similarly, the absence of an INSERT privilege is typically illogical since it hinders the creation of object instances. Nevertheless, specific scenarios may arise where, for instance, a system administrator intends to maintain a table in a read-only state after being populated with appropriate data. In such cases, the system administrator can safeguard table integrity by removing the INSERT privilege from all roles involved.

UPDATE and DELETE are two other privileges which are commonly desired on tables and views as they allow for data modification and removal of unnecessary and obsolete records. However, their presence is not guaranteed, and the absence of these privileges does not necessarily imply an error or security concern. Certain tables are intentionally non-updateable and non-deleteable, serving purposes like historical data retention, such as audit logs or archived records, while specific data, such as reference data or configuration settings, is deliberately kept immutable to maintain data consistency.

Similar to the reachability check, the accessibility assessment commences by first retrieving all database objects, specifically tables and views. However, unlike the reachability check, which focuses on associated processes, this check centres on confirming the presence of SELECT, INSERT, UPDATE, and DELETE privileges for each

database object. If any of these privileges are missing, a ‘Warning’ is issued, acknowledging that such omissions may not always indicate an error.

3. **Non-Redundant Roles**

The aim of this rule is to identify and eliminate redundant roles found in the database. Redundant roles, having no associated users, serve no functional purpose and can have detrimental effects. Not only do they impose unnecessary performance overhead but also introduce security vulnerabilities. Redundant roles may create loopholes in the access control system, whereby malicious users can associate themselves with such roles and obtain excessive privileges or access to sensitive data. Furthermore, the lack of auditing procedures for these roles, resulting from their lack of association with users, increases the risk of potential misuse. Without proper auditing in place, it becomes challenging to detect and prevent unauthorised usage or manipulation, leaving them vulnerable to exploitation. The implementation of this check involves retrieving all system roles and their associated users. Any role without linked user tuples is marked as redundant and flagged.

4. **Inference channels**

If a role has permission to add a new object to an entity but lacks permission to read tuples of that entity, a user with that role can still deduce primary key values by adding an object with those values. If the system rejects this operation, it indicates that the primary key values already exist in the database. Furthermore, if the role also has permission to delete an instance from the same object, this attack can be executed without leaving the inserted data as evidence. The 'inference_check' function integrated into our design tool can identify instances where SELECT permission is absent while INSERT or DELETE privileges aren’t, potentially indicating the presence of inference channels. It is then the responsibility of the DBA to assess whether this situation is deliberate or otherwise. Pseudocode 5.4 illustrates the implementation of this inference channel check.

Pseudocode 6.1 Algorithm for Inference Check

```
14:  Input: tenant_name text
15:  Output: entities JSON
16:
17:  Algorithm:
18:        >Retrieve all roles
19:    rolArray ← SELECT rolename
20:      FROM spsecurity.roles
21:      WHERE tenantid = sn of tenant_name
22:
23:  For tab in
24:    SELECT table_name
25:    FROM information_schema.tables
26:    WHERE table_schema = tenant_name
27:  Do                                          >Check for 'SELECT' privilege
28:    For ro in array rolArray
29:    Do
30:      selPriv ← SELECT function has_table_privilege(ro, tab, 'SELECT')
31:      If !(selPriv) then                     >Check for 'INSERT' or 'DELETE' privilege
32:      insertPriv ← SELECT function has_table_privilege(ro, tab, 'INSERT')
33:      deletePriv ← SELECT function has_table_privilege(ro, tab, 'DELETE')
34:      If (insertPriv or deletePriv) then
35:      retArray ← ta
29:      End if
30:      End if
36:    End loop
37:  End loop
38:
39:  return retArray
```

6.3 Pre-SQL Script Generation Configuration

After successfully completing the checks outlined in Section 6.2, the CDBMS is configured with the necessary settings, enabling the generation of SQL scripts (explained in more detail in Section 6.4). To uphold consistent performance, availability, and sustained support for the CDBMS, it is imperative to establish a comprehensive Service Level Agreement (SLA) between the CSP and the Software House responsible for presenting the proposed solution. This SLA delineates the mutually agreed-upon terms, encompassing performance expectations, maintenance responsibilities, response times for issue resolution, and other critical factors essential for ensuring the uninterrupted and dependable operation of the system.

At this stage, the cloud Database Administrator (DBA), in close cooperation with the software house developer:

1. Modifies the host file

The host file serves as a mapping of hostnames to IP addresses. To include the IP addresses from which the new tenant operates, an additional line is added to the host file. Figure 6.1 provides a visual representation of the necessary communication between the new tenant and the DBA to enable the operating IP addresses.

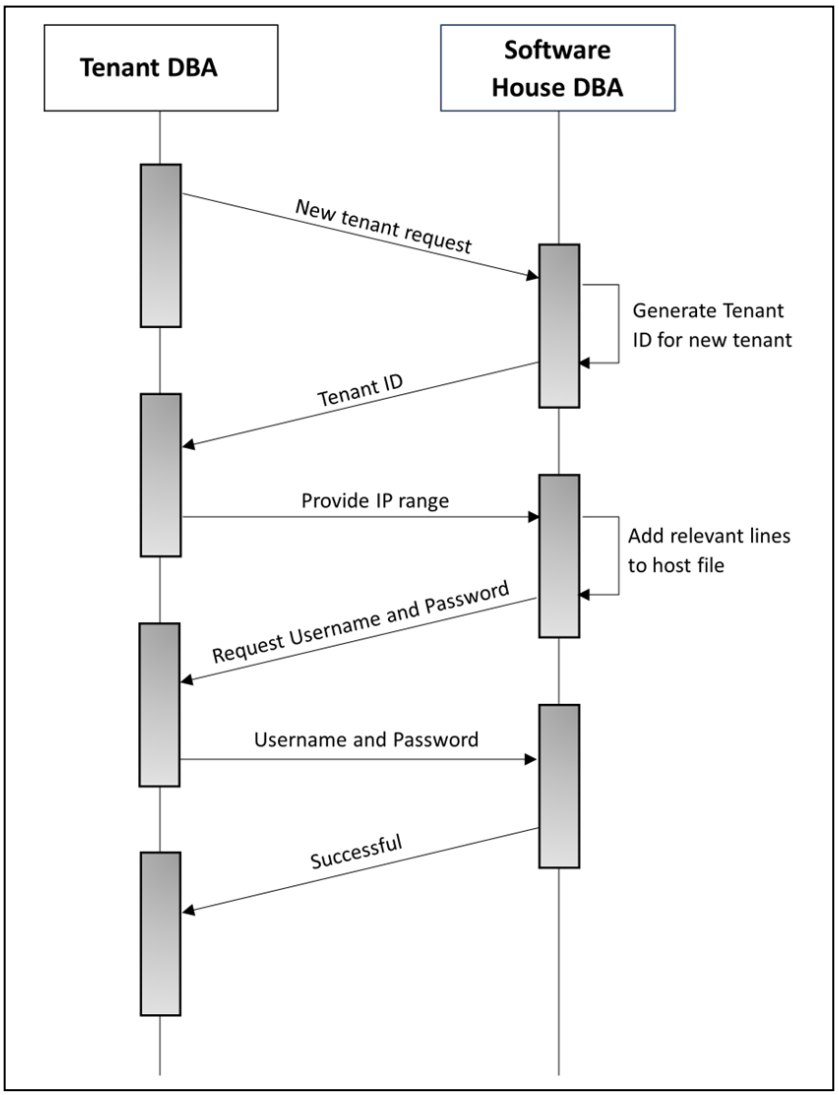


Figure 6.1: Host file sequence diagram

2. **Arranges the configuration file if required**

As new tenants are integrated into the system, the configuration file may necessitate adjustments to accommodate their presence effectively. These adaptations encompass a range of tasks, including expanding the capacity for concurrent connections to

handle the heightened user load, strategically allocating additional resources to ensure efficient load balancing, and determining optimal storage locations for log files. Within the context of this dissertation, which relies on PostgreSQL as its DBMS, these objectives may be achieved through the strategic adjustment of key parameters. These parameters include 'shared_buffers' for optimised RAM allocation, 'max_connections' for the management of concurrent database connections, 'work_mem' for enhancing per-operation memory usage, and 'maintenance_work_mem' for efficient maintenance operations. Such modifications are in direct response to the increasing demands of users, ultimately resulting in seamless resource allocation and load balancing for the system.

Additionally, in scenarios where 'separate schema' or 'same table' multi-tenant approaches are selected, the final phase includes establishing a shared database for all tenants. After this setup is complete, the final SQL Script report can be generated.

6.4 SQL Scripts

The key feature of the CASE tool is its final component, responsible for the automated generation of SQL code depicting a complete system. Each database created is designed to include security considerations specific to each tenant and is based upon the initial system diagrams provided to the tool. The produced code is generated through the invocation of specific SQL scripts, which are carefully selected based on the unique requirements of the tenant and the multi-tenant scenario required. Throughout this section, a comprehensive description of each script is presented. Distinct scripts were developed to cover all the functions outlined in the basic database design architecture (Section 2.5.1), as well as other areas referenced in various literature works on multi-tenant security. Additionally, other scripts were created to cater for the specific individual needs of each tenant (Section 5.8).

Certain SQL scripts, such as those responsible for table creation, may generate different lines of code depending on the multi-tenant scenario being implemented. This is because security aspects required in different scenarios might differ. To ensure a systematic approach, each SQL script was created as a function with a specific set of lines of code encapsulated in it, thus allowing for more efficient management and modification of the codebase.

This section begins with a comprehensive description of each developed script, presented in the expected order of execution, with particular emphasis on the security considerations taken into account during development. Following this, a brief explanation of the different scripts invoked in each of the three multi-tenant scenarios is provided. The full list of scripts, along with an indication of the individual typically responsible for their execution, is presented in a table in Appendix B.3.

6.4.1 Tenant DBA role creation

The initial section of the final SQL report, regardless of the database system in question, comprises a set of lines capable of creating a user associated with the 'tenantDBA' role. This user is mandatory in all systems and holds the highest level of privileges within a specific tenant's database. Examples of these privileges include the capability to create new users and assign them specific roles, among others. Despite having the most elevated level of access in the system, unnecessary privileges such as the ability to create databases are not granted to ensure adherence to the security principle of least privilege.

The code associated with the operation explained in the previous paragraph is produced via the invocation of the script entitled ‘create_user’. This script generates the necessary SQL code to create a user with specific privileges (in this case, tenant DBA) when provided with the relevant parameters such as the username, password, and associated role. It is worth noting that, during the user creation process, the function generates syntax that commences with 'CREATE ROLE,' as depicted in SQL Code 6.1. This arises from PostgreSQL's approach of regarding users as roles with logins.

```
CREATE ROLE "UoM_dba" WITH
INHERIT
NOCREATEDB
CREATEROLE
LOGIN
CONNECTION LIMIT -1
ENCRYPTED PASSWORD '******';
```

SQL Code 6.1: CREATE USER syntax

6.4.2 Database creation

Following the generation of code related to the tenant DBA role, the SQL code responsible for database creation is produced. This is achieved by means of the 'create database' script, which is invoked either once or multiple times. In the case of

‘separate schema’ and ‘same table’ scenarios, the 'create database' script is executed only once by the software provider on initialisation. On the other hand, in the case of the ‘separate database’ scenario, this script is executed every time a new tenant is added to the database system. The type of encoding utilised for the newly created database in a ‘separate database’ scenario is determined by the tenant's input in the 'encoding' section of the 'Requirements' file (as explained in Section 5.8). This encoding is in line with the one employed in both 'same table' and 'separate schema' scenarios.

6.4.3 Schema creation

The code generated incorporates two main schemas: one dedicated to storing information relevant to the software provider and the application program, and another designed to house tenant-specific data. As with the prior script, the frequency of execution of the ‘create_schema’ script is dependent on the specific type of multi-tenant scenario being implemented. In the separate database multi-tenant implementation, the 'create_schema' script is executed twice per tenant, facilitating the establishment of the two previous schemas. In contrast, in the separate schema approach, the software provider schema is created only once during the initial setup, while a new schema is generated for each subsequent tenant. On the other hand, in the 'same table' scenario, the script is employed only twice at the initialisation to create the software provider and tenant schemas, regardless of the number of tenant systems created. This is because all tenant data is stored in the same tables and schemas.

Upon creation of a new PostgreSQL database, a default schema named 'public' is automatically generated. Leaving this schema unattended may lead to potential security vulnerabilities. Thus, all roles that are created are denied access to this 'public' schema as a precautionary measure.

6.4.4 Creation of tables and referential constraints

Following schema creation, the SQL script proceeds to detail the 'CREATE' statements essential for establishing the tables that implement the designed database. In the 'separate database' and 'separate schema' scenarios, the complete set of tables required to implement the ERM diagram is generated for each tenant. In the 'same table' scenario, the table set for a given system is created only once during the initial stage and serves as the storage location for the data of all tenants.

SQL statements for table creation are generated by means of the 'create_tables' function, composed primarily of a FOR-loop and multiple sub-procedures. Each sub-procedure is responsible for generating specific sections of code related to various table components, including attributes and constraints. When these sections are combined, they formulate the complete ‘CREATE TABLE’ statement for each table.

The aforementioned process commences with the invocation of the 'erm_all_entities' procedure, responsible for the retrieval of all ERM entities (strong and weak). Following this, a loop is used to create distinct 'CREATE TABLE' statements for each entity retrieved. Each statement is constructed incrementally via the invocation of a set of subroutines, commencing with the 'create_attributes' function. The responsibility of this function is that of adding the segment of the 'CREATE TABLE' statement pertaining to the attributes of each table.

To achieve this, the function initially employs the 'erm_single_attributes_nderived' sub-procedure to fetch attributes related to the input entity. These attributes are either singular and non-derived or atomic parts of composite attributes. SELECT statements retrieve these attributes from either 'erm_attribute_entity' or 'erm_attribute_weak_entity' tables, based on the nature of the input entity. The WHERE clause ensures only attributes meeting the defined criteria are filtered and returned.

The procedure that manages attribute retrieval when a strong entity is supplied as an input parameter is depicted in Pseudocode 5.5. When dealing with weak entities, an analogous process is performed but a distinct attributes table is used.

Pseudocode 6.2 Attribute retrieval for strong entity

```
1:   Input: sysdiag_name
2:   Output: text[]
3:
4:   Algorithm:
5:     SELECT array_agg(erm_a_name || ' ' || erm_a_type)
6:     INTO attNameAndType
7:     FROM erm_attribute_entity
8:     WHERE erm_a_se_sn = attribute associated with sn of sysdiag_name
9:     AND erm_a_derived = FALSE
10:                          > composite attributes are divided into simple attributes
11:    AND erm_a_simple = TRUE
12:    AND erm_a_valued = TRUE
```

Once the attributes, specified in the prior paragraph are retrieved, the 'create_attributes' process proceeds to store them into an array of 'name-type' pairs, which can then be appended to the 'CREATE TABLE' statement. In multi-tenant scenarios where the same table approach is adopted, an additional column named 'tenant ID' is added to all tables alongside the array of retrieved attributes. This column, of type integer, serves as a means of identification, enabling tenants to access their own data exclusively while preventing unauthorised access to other tenants' data.

Subsequently, the 'create_multivaluedAtts' function is invoked to handle the presence of multi-valued attributes. This function generates a new weak entity composed of a newly generated primary key and the multi-valued attribute itself. Additionally, it establishes a weak one-to-many relationship for each multi-valued attribute, connecting the original entity to the newly created one.

Once all corresponding attributes are appended to the 'CREATE TABLE' statement related to the current entity, its formulation process proceeds by identifying the primary key(s) associated with the entity under consideration. This task is delegated to the 'primarykeys' function which relies on the sub-procedure 'erm_pk_attributes', to execute a SELECT statement that retrieves the attribute(s) designated as the primary key in the ERM specifications. Once acquired, the primary key attribute(s) are returned in textual format, structured as 'CONSTRAINT *pkconstraintname* PRIMARY KEY (*pkname pktype*)', and added to the 'CREATE TABLE' statement of the current entity. This key is then inserted as a tuple, by means of an INSERT statement, into the attribute table of either the 'erm_attribute_entity' or 'erm_entity_weak_entity' relation, depending on which entity is currently being processed. As an example, the CREATE TABLE statement for the 'Employee' entity in the scenario of a separate multi-tenant database is illustrated in SQL Code 6.2.

```
CREATE TABLE IF NOT EXISTS schemaname."Employee"(employeeNum integer,
job varchar, salary double precision, employeeName varchar,
CONSTRAINT pkEmployee PRIMARY KEY (employeeNum));
```

SQL Code 6.2: CREATE TABLE syntax

Subsequently, the next step in the table creation process involves the identification of any attributes that cannot have NULL values. To carry out this task, the

‘notnull’ sub-procedure, responsible for returning textually all such attributes is invoked. This triggers the ‘erm_required_attributes’ function, which retrieves the attributes associated with the entity under consideration whose 'required' column value is set to TRUE. This column is found in either the 'erm_attribute_entity' or 'erm_entity_weak_entity' relations, depending on the type of entity being considered (i.e., strong, or weak). For each required attribute, the function generates an ‘ALTER TABLE *tablename* SET *requiredcolumnname* NOT NULL', appended immediately after the 'CREATE TABLE' statement for the corresponding entity.

Following creation of the required 'CREATE TABLE' statements, relationships are established using the 'create_relationships' function. The function begins by retrieving all relationships specified in the ERM diagram through the 'erm_relationships' function. For each relationship, the participating entities are obtained from either the 'erm_part_ent' table for strong entities or the 'erm_weak_part_ent' table for weak ones. The retrieved information encompasses the entities’ names, participation type, and cardinality.

In binary or self-referencing relationships, with a one-to-one (1:1) relationship involving one ‘total’ and one ‘partial’ participating entity, the primary key of the partial entity is appended into the total entity and set as UNIQUE and NOT NULL. When both entities share the same participation type (either both 'total' or both 'partial'), the primary key can be added to either entity. The difference lies in how it is set: for 'total' participation, the primary key is marked as UNIQUE and NOT NULL, while for 'partial' participation, it is set as UNIQUE without the NOT NULL constraint. Following the completion of this procedure, foreign key constraints are established, and relationship attributes are incorporated into the entity associated with the foreign key. Contrastingly, to establish relationships in one-to-many (1:M) associations, the primary key of the '1' side entity is added to the 'M' side, creating foreign keys. The procedure follows the same pattern as a 1:1 relationship when both sides have total or partial participation. If on the other hand there is a difference in participation types, the foreign key attribute is set to NOT NULL. Any relationship attributes present in the association are added to the ‘M’ side of the relationship.

In many-to-many (M:N) relationships, a relationship table incorporating any relationship attributes and the primary keys of both entities involved, is created. In this

case, UNIQUE and NOT NULL constraints are unnecessary for the primary key columns, as the primary keys of both tables collectively constitute the new composite primary key, which is automatically enforced as NOT NULL through the SQL DDL primary key constraint. Subsequently, connections between the attributes stored in the new table and the attributes of both the '1' and 'M' tables are established via the creation of foreign key constraints. In ternary relationships, the procedure follows a similar approach but with the addition of the primary keys of all three entities in the relationship table. Each entity's primary key attributes are linked as foreign keys to their respective tables. The composition of the primary key set of the new table is then determined according to the cardinality of the three entities in the relationship.

6.4.5 Creation of processes

The generated SQL code for a tenant's database system also encompasses processes closely tied to the system being portrayed within the database. In addition to tables, database systems use functions to encapsulate and modularise processes and application logic, enabling the manipulation of stored data, automation of tasks, and enforcement of business rules. In PostgreSQL functions are available in two flavours – those with a SECURITY INVOKER label and others with a SECURITY DEFINER one. The former ascertains that functions are executed with the permissions of the calling role, whilst the second option allows functions to be executed with the permissions of the function owner [34]. Opting out of the second option is advisable since it may allow a role to execute functions with higher security clearances than its own, making it vulnerable to Trojan horse attacks. To mitigate these risks, the proposed tool sets all generated functions for the application's processes as SECURITY INVOKERs.

The code related to functions is generated via the 'create_processes' script which commences with the invocation of the 'dfd_processes' function, responsible for the retrieval of all processes specified in the DFD. Subsequently, a FOR-loop is then used to iterate through the retrieved processes and classify them into one of five predefined categories. Classification is based on the position of each process within the ELH diagrams as described in Section 5.4.1, as well as other relevant factors mentioned throughout this section. Descriptions of the five categories together with key considerations during code generation, such as input parameters, return values, and process body, are indicated hereunder.

a. Processes associated with **INSERT** operations.

As explained in Section 5.4.1, determining whether a process encapsulates INSERT, UPDATE, or DELETE operations is based on its position in the ELH diagram. If the retrieved process is positioned at the beginning of such diagrams, it is associated with an INSERT operation and the following code generation process ensues: the process's return type is defined as 'void,' and its input parameter is set to JSON. This configuration aligns with the process's purpose, which is to perform insertions without returning values (hence 'void'), while ensuring that insertion values are provided as a list of 'attribute-value' pairs (JSON), signifying the associated column names and values. The input parameter can be modified by tenants, if necessary, to enable insertion exclusively on specific columns. The process body is then created by generating INSERT statements, responsible for inserting the data provided as a parameter into the corresponding table(s). Furthermore, the process body contains an EXCEPTION block to address errors, including those that may occur when attempting to insert a record with an existing primary key and other types of errors. the scenario where multiple tenants are using the 'same table', the INSERT statement is adapted to incorporate an additional column which captures the 'tenant ID' of the present user. SQL Code 6.3 illustrates the code generated for the 'openAccount' process in a 'separate schema' scenario.

```
CREATE FUNCTION aps_ss."openAccount"(attValPair JSON)
 RETURNS void
 AS
 $body$
 DECLARE
 --include required variables
 BEGIN

 --body of function
  BEGIN
   INSERT INTO aps_ss."Clerk_ss_Account"
   SELECT *
   FROM json_populate_record(NULL::aps_ss."Clerk_ss_Account",attValPair);
  EXCEPTION
   WHEN unique_violation THEN
        RAISE EXCEPTION 'Primary key error';
   WHEN others THEN
        RAISE EXCEPTION 'Record cannot be inserted';
  END;
 END
 $body$
 LANGUAGE plpgsql;
```

SQL Code 6.3: Code generation of process carrying out an INSERT operation

b. Processes associated with **UPDATE** operations.

As outlined in Section 5.4.1, processes situated in the middle sections of the ELH diagrams are associated with UPDATE operations. The code generation procedure for these processes closely follows the approach described in Point 1 with some differences in the input parameters and code body. The return type remains unchanged, that is, set to 'void' with no variation. In addition to the JSON list of 'attribute-value' pairs, used also in Point 1, the input parameters also encompass the primary key(s) that identify the table requiring updates. The method's body is comprised of generated UPDATE statements, responsible for the modification of the tuple with a primary key matching that provided by the user. The new information stored in the tuple is derived from the values specified in the first input parameter. SQL Code 6.4 shows the generated code for an UPDATE process in the 'same table' scenario.

```
CREATE FUNCTION "modifyUnitDetails"(attValPair JSON, unitCode varchar)
RETURNS void
AS
$body$
DECLARE
--include required variables
BEGIN

--body of function
 BEGIN
  UPDATE "Unit"
  SET "departmentNumRef"=json_record."departmentNumRef",
  credit=json_record.credit, "unitName"=json_record."unitName",
  "unitCode"=json_record."unitCode"
  FROM json_populate_record(NULL::uom_sd."Unit", attValPair)AS json_re
  WHERE "Unit"."unitCode" = unitCode
  AND "tenantid" = current_tenantid();
 EXCEPTION
  WHEN unique_violation THEN
       RAISE EXCEPTION 'Primary key error';
  WHEN others THEN
       RAISE EXCEPTION 'Record cannot be updated;
 END;

END
$body$
LANGUAGE plpgsql;
```

SQL Code 6.4: Code generation of process carrying out an UPDATE operation

c. Processes associated with **DELETE** operations.

In line with the discussion provided in Section 5.4.1, processes located at the end of ELH diagrams are associated with DELETE operations. Consequently, if the retrieved DFD process is positioned at that juncture, one or more DELETE statements

are generated as part of the process's body. These statement(s) are responsible for deleting the tuple that has a primary key value which corresponds to that inputted by the user.

Thus, it can be inferred that the input parameters necessary for such processes are composed of the primary key(s) of the table containing the tuples to be deleted. Similar to INSERT and UPDATE processes (explained prior), a WHERE condition is used in 'same table' implementations, to ensure that tenants are only allowed to delete tuples associated with their corresponding ‘tenant ID’.

d. Processes associated with **SELECT** operations.

In contrast to the processes considered in the preceding three categories, some processes are present in the DFD but not in the ELH. These processes are typically associated with SELECT operations such as report generation, which do not modify the state of an entity in any manner. When generating code for such processes, the return type is defined as 'setof TABLE', wherein the table type is determined by the source of the selected tuples. Input parameters are not generated for this set of processes as it is presumed that reports or other similar functions require all available information. SELECT statement(s), capable of retrieving tuples from one or more specified tables are generated within the process’ body. If information is to be retrieved from multiple tables, a verification is performed prior to the generation of statements to confirm the feasibility of a join between the tables. This is confirmed by examining the existence of a common column (a foreign key) between them.

e. Parent processes

Conclusively, when the retrieved process is recognised as a parent, the process body is constructed using a series of IF-ELSE statements, with the quantity determined by the number of linked child processes. Each IF-ELSE statement is designed to examine one of the input parameter values (described in the next paragraph) and invoke a specific child process based on that value. To perform this task, the 'crudmatrixpp' table is utilised to determine the number of child processes together with their names and corresponding permissions (SELECT, INSERT, UPDATE, or DELETE). The input parameters for each child process are generated based on the required permis-

sions, according to the steps outlined in the previous four points. Using this information, the method signature is created, enabling it to be included in the IF-ELSE statements for invocation.

Given that parent processes are designed to trigger child processes instead of returning result sets, their return value is specified as 'void'. To ensure successful execution, parent processes encapsulate two types of information in their input parameters: (1) details specifying which child process to execute and (2) input parameter values for the final process in the hierarchy associated with the parent process. Both parameters are formatted as JSON to allow for flexibility in the input. Parameter (1) uses JSON to cater for potential multiple levels of child processes, while parameter (2) allows for a varying number of input values according to the type of the final process in the hierarchy.

SQL Code 6.5 depicts the code generated for the 'createTransaction' parent process in the 'BankingSystem' separate database scenario, consisting of three child processes.

```
CREATE FUNCTION "createTransaction"(childSelection JSON,finalChild-
Params JSON)
 RETURNS void
 AS
 $body$
 DECLARE
 --include required variables
 BEGIN

 --body of function
 IF ((SELECT value
   FROM json_each_text(childSelection->0) LIMIT 1)::INTEGER = 1) THEN
  PERFORM *
  FROM depositPayment"(finalChildParams);
 ELSIF ((SELECT value
  FROM json_each_text(childSelection->0) LIMIT 1)::INTEGER = 2) THEN
  PERFORM *
  FROM "makeDirectPay"(finalChildParams);
 ELSIF ((SELECT value
        FROM json_each_text(childSelection->0) LIMIT 1)::INTEGER = 3) THEN
  PERFORM *
  FROM "withdrawPayment"(finalChildParams);
 ELSE
  RAISE NOTICE 'Selection inputted does not exist';
 END IF;
END
$body$
LANGUAGE plpgsql;
```

SQL Code 6.5: Code generation of parent process 'createTransaction'

6.4.6 Creation of roles

The next set of generated codes is responsible for creating the necessary roles within the system which is facilitated by the 'create_roles' script developed specifically for this purpose. These statements are subsequently incorporated into the generated SQL scripts, along with the other previously mentioned points outlined in this section. As discussed in Section 5.4.5, the initial generic application roles created - not yet considering the partitioned ones - are based on the external entities specified in the DFDs, as they denote all entities interacting with the system. Thus, the initial step of the 'create_roles' function involves the invocation of the 'dfd_external_entities' function, responsible for the retrieval of all external entities. Following this, a FOR-loop is employed to iteratively invoke the 'create_role' function for each external entity in the system, generating the respective 'CREATE ROLE' statement in text format. The latter function takes several parameters into account to generate appropriate code, including the name of each entity which is used to define the corresponding role, as well as parameters related to the privileges assigned to these roles. All parameter values associated with privileges are set to false. This is because generic system roles like ‘Managers’ or ‘Clerks’ should not have elevated privileges like superuser status or the ability to create databases or roles. They should only be granted the minimum privileges needed for their tasks.

The generated generic role statements lack login credentials because only users associated with these roles have logins. This is demonstrated in SQL Code 6.6, showing the 'CREATE ROLE' statement for the 'Clerk' role in the 'UoM' database.

```
CREATE ROLE "ClerkUoM" WITH
NOSUPERUSER
NOCREATEDB
NOCREATEROLE
NOINHERIT
NOREPLICATION
NOBYPASSRLS
NOLOGIN;

REVOKE ALL ON DATABASE uom FROM "ClerkUoM";
GRANT CONNECT ON DATABASE uom TO "ClerkUoM";
REVOKE ALL ON SCHEMA tenant FROM "ClerkUoM";
GRANT usage ON SCHEMA tenant TO "ClerkUoM";
REVOKE ALL ON ALL TABLES IN SCHEMA tenant FROM "ClerkUoM";
REVOKE EXECUTE ON ALL functions IN SCHEMA tenant FROM "ClerkUoM";
```

SQL Code 6.6: CREATE ROLE syntax

Additionally, it can also be noted, that as part of the role creation process, other sub-procedures are called within the FOR-loop. These sub-procedures involve granting connections to the current database for the respective role being created. Additionally, they establish usage permissions on schemas to ensure that each role can access the required resources within the database. As described in Point 3, the number of schemas considered varies depending on the multi-tenant scenario employed. Newly created roles are only granted usage privileges on schemas not on the schema's tables and functions. The latter privileges are then granted based on the CRUD matrix (as explained in the subsequent point) to ensure that the concept of least privilege is implemented.

6.4.7 Granting and Revoking of Privileges on Roles

This step is of great significance when it comes to maintaining a secure environment. Following the establishment of the required tenant roles, specific privileges can be assigned to each role. The assignment and removal of these privileges are facilitated using the GRANT and REVOKE keywords, respectively. A set of functions is responsible for generating the necessary scripts to grant privileges on different database objects, with the function's name indicating the type of object. The parameters within each function depend on the specific privilege associated with each database object. For instance, a database might have CONNECT, CREATE, or TEMPORARY privileges, while a table might have SELECT, INSERT, UPDATE, or DELETE privileges.

Following creation, a newly assigned tenant user receives the CONNECT privilege for their respective database. In addition, the role is granted USAGE or CREATE privileges on specific schemas facilitating access. This is essential since granting privileges like SELECT on specific tables and functions requires the presence of appropriate privileges on the relevant schema.

Subsequently, EXECUTE privileges are granted to roles on functions. In line with best security practices, the principle of least privilege is followed, ensuring that users are granted only the minimal privileges required to perform their tasks. To achieve this, the 3D CRUD matrix, introduced in Section 5.5, is employed. If a function identified within the matrix is classified as a parent, its associated children are also retrieved and EXECUTE privileges are granted to them as well. SQL Code 6.7 illustrates the granting of the EXECUTE privilege for the 'editCustomerDetails' process

to the 'Clerk' role belonging to the 'aps' tenant. Although the procedure for granting access to processes remains consistent across all three types of multi-tenant scenarios (separate database, separate schema, same table), distinctions arise when accessing tables or specific segments of the table. These disparities are dictated by the defined policies, as elucidated in Point 11.

```
GRANT EXECUTE ON FUNCTION aps_sd."editCustomerDetails"(JSON, INTEGER)
TO "Clerkaps_sd";
```

SQL Code 6.7: Granting EXECUTE privilege on function to 'Clerk' role

In addition to that, dedicated scripts have been created to generate GRANT statements for roles on various database elements. Privileges such as SELECT, INSERT, UPDATE and DELETE on the tables and views are granted based on the generated expanded CRUD matrix. This involves the deployment of four scripts, each responsible for granting the relevant privileges. In certain cases, access may be granted on views instead of tables, based on specific requirements.

To counteract for the GRANT privileges defined previously, and to ensure that a holistic, secure database is formulated, there are functions responsible for the creation of scripts required for the revocation of privileges on database objects. Similar to the grant functions, the type of object a function is working on can be noted from the function's name.

6.4.8 Creation of tenant tables and functions

The proposed tool includes a standard set of tenant tables and functions that are automatically generated for all systems it creates, regardless of the system's characteristics or requirements. As outlined in Section 5.7.1, the structure of these objects, is transformed and stored in relational tables from the 'ListOfTenantTables.json' and 'ListOfTenantFunctions.json' files, respectively. Subsequently, appropriate SQL syntax for the creation of the above-mentioned tables and functions is generated and appended to the final generated script.

The latter part is accomplished via the execution of two main functions – 'create_tenant_tables' and 'create_tenant_functions'. The first function is responsible for generating the code required for the creation of tenant tables. This entails retrieving the entity count, iterating through each element using a FOR-loop, and generating a

'CREATE TABLE' statement composed of several attributes and constraints. The attributes encompass those specified in the file, provided that their corresponding data types are valid, along with an additional ‘tenant ID’ column, in separate schemas and same table multi-tenant scenarios. The constraints, determined by multiple conditional statements which identify the presence or absence of primary key, NULL, unique, or foreign key constraints, are then incorporated into the script according to the corresponding syntax, previously specified in Point 4.

The second function, 'create_tenant_functions,' is responsible for the generation of code associated with functions that are present in all systems produced by the proposed tool. As detailed in Section 5.7, these functions are stored in the 'tenant_functions' table of the software provider schema. Code for such functions is generated in the following manner: Each function’s name is set in accordance with the value found in the ‘tf_functionname’ column of this table and the parameters are formulated according to the attributes and data types found in the ‘tf_functionparameters’ array. Successively, the RETURN value of the function (‘tf_returnvalue’) is specified and if required, any variables are declared. Conclusively, the main body of the function, encapsulated by a BEGIN and END statement is articulated according to the specifications listed in the ‘tf_body’ column of the ‘tenant_function’ table.

6.4.9 Creation of partitioned roles

Following the creation of the generic roles (specified in Point 8), the subsequent phase entails setting up partitioned roles. As detailed in Section 5.6, these roles are fine-tuned to cater to specific tenant preferences by incorporating conditions. The role creation process follows the same syntax as specified in the previous point. However, the differentiation arises with the utilisation of policies specifically crafted for these partitions, as elaborated in Point 11.

6.4.10 View Creation

As mentioned in Section 5.6.1, vertical partitions are implemented as views, while horizontal partitions are implemented as RLS policies. The subsequent section of the generated SQL code focuses on the creation of views. To enhance security, all views created by the proposed tool have the SECURITY INVOKER attribute set to

true. This ensures that permissions for accessing the underlying tables are determined based on the privileges of the view's user, rather than the view owner.

Prior to generating the 'CREATE VIEW' statement, each view undergoes property evaluation. As discussed previously, not all views allow insertion, updating, or deletion. Therefore, an assessment is performed against predefined criteria to determine the updateability of the view. Additionally, a check is conducted prior to view creation to ensure that the horizontal partition columns align with those specified in the vertical partition. If this condition is not fulfilled, the software developer is alerted to address any required modifications or adjustments to ensure compliance.

Once all the necessary checks are successfully completed, the views can be created. In addition to generating the corresponding SQL code, such as the one displayed in SQL Code 6.8 below, a new row is inserted in the 'CRUD Matrix' table for each view. This ensures that the matrix provides a comprehensive representation of both tables and views, offering a holistic view of the database structure.

```
CREATE VIEW aps_sd."Clerkaps_sd_Transaction"
WITH (security_invoker = true)
AS SELECT transactionType
FROM aps_sd."Transaction";
```

SQL Code 6.8: SQL Code used for the creation of views

Additionally, following the creation of views based on the specified vertical partitions, the processes described in Point 5 undergo slight modifications. These adjustments are implemented to grant varying levels of access to different roles, in accordance with their respective authorised data objects. Some roles may have direct access to the entire table used by the process, while others are limited to subsets of that table represented by the created views.

6.4.11 RLS Policies

RLS policies play a pivotal role in implementing horizontal partitions on roles within a role-based access control framework. These partitions are instrumental in accommodating specific conditions identified by tenants for individual users, as well as system-imposed conditions to support multiple tenants using the system. The initial

set of RLS policies generated by the proposed tool focuses on establishing RLS policies for essential system tables such as 'roles', 'views', 'user', and 'userconditions' linked to both the tenant's DBA and other roles.

The implementation of these policies ensures that in separate schema and same table approaches, user access to the specified tables is restricted based on their tenant associations, thereby ensuring access solely to authorised data. Policies on these tables are defined such that the DBA can view only the rows corresponding to their 'tenant_id', while other roles are granted access to rows that match both their 'tenant_id' and 'rolename'.

The creation of each policy involves invoking the 'enable_rls' and 'create_policy' functions in succession. The 'enable_rls' function generates the necessary script to enable row-level security on a specific table or view, utilising the 'ALTER TABLE tablename ENABLE ROW LEVEL SECURITY' construct. Additionally, it incorporates the 'FORCE ROW LEVEL SECURITY' construct to enforce row-level security, even for relation owners. The 'create_policy' function, on the other hand, generates the script required to create a policy on a particular table or view using the 'CREATE POLICY' construct.

The second set of policies assumes a significant role in creating tailored rules specifically designed to accommodate horizontally partitioned roles. All entities associated with each partitioned role are retrieved and a corresponding policy is generated. The conditions associated with each policy, are carefully formulated through a series of steps. Initially, each entity is assessed to determine its classification as a strong or weak entity. Next, the data type of each attribute contributing to the partition is retrieved from the corresponding database table. Based on the data type, specific conditional operators such as <, >, =, and != are established.

Subsequently, for each partitioned column and for every allowed conditional operator, the function formulates a SELECT statement responsible for retrieving values from the 'userconditions' table associated with that operator for the current user. In the same table scenario, the 'tenant ID' is also included as part of the conditions. Presented below (SQL Code 6.9) is an example, of an RLS (Row-Level Security) policy established on the 'Branch' table and associated with the 'CFO' role. Only the ‘USING section is displayed i.e., the 'WITH CHECK' part is not included in this representation.

```
CREATE POLICY "policy_Branch_CFOaps_sd"
 ON aps_sd."Branch"
 TO "CFOaps_sd"
 USING ("Branch".location =
     ANY (SELECT CAST(val AS varchar)
   FROM aps_sd.userconditions
   WHERE ucviewname =
     (SELECT DISTINCT views.viewname
      FROM sp.views, sp."viewsColumns"
      WHERE tablename = 'Branch'
      AND columnname = 'location'
      AND horizpart
      AND views.viewname = "viewsColumns".viewname)
   AND op = '='
   AND username = (SELECT current_user))
   AND "Branch".location !=
     ANY (SELECT CAST(val AS varchar)
   FROM aps_sd.userconditions
   WHERE ucviewname =
      (SELECT DISTINCT views.viewname
      FROM sp.views, sp."viewsColumns"
      WHERE tablename = 'Branch'
      AND columnname = 'location'
      AND horizpart
      AND views.viewname = "viewsColumns".viewname)
   AND op = '!='
   AND username = (SELECT current_user)));
```

SQL Code 6.9: RLS Policy established on the 'Branch' table

In the same table multi-tenant scenario, an additional SQL script based on RLS policies is implemented to restrict visibility to rows pertaining only to the current tenant. This script plays a key role in creating policies for every table associated with both the DBA and other tenant roles. When generating policies for the DBA role, a FOR-loop is employed to retrieve all database tables. Each policy is then established with a condition specifying 'tenant_id' as equal to the tenant ID of the currently logged-in tenant. Conversely, when policies are being generated for other roles in the system, only the tables accessible to each role, as specified in the 'rolesprocesses' table, are affected. This ensures that the policies are tailored to each role's permissions, further enhancing data security and access control within the multi-tenant approach.

6.4.12 Backups

The creation of backup files is a crucial component of any robust system, providing a backup strategy in the event of system disruptions. However, implementing an effective backup strategy can be challenging. While creating backups is relatively straightforward in 'separate database' and 'separate schemas' scenarios, the same cannot be said for the 'same table' implementation. In this case, generating

backups on a per-tenant basis becomes almost impossible. This limitation arises due to the risk of overwriting data for other tenants when backing up and restoring tables.

PostgreSQL provides three primary modes of data backup which can be used when backing up database systems or schemas: SQL dump, file system-level backup, and continuous archiving. The choice of backup mode depends on the specific needs and requirements of the user. SQL dump [190], [191] is suitable when a snapshot of the current database table structure and data is sufficient for backup purposes. On the other hand, continuous archiving [191] is typically employed in conjunction with file system-level backup when more frequent backups are required. The procedure involves setting up a dedicated backup table with specific columns designed to store information related to the schema name, table name, and the actual data. When a tenant requests a backup, a process is initiated where the chosen rows from the original table are transferred into this designated backup table. This transfer operation is accomplished using an INSERT statement, and a condition is included in this statement to ensure that only the rows associated with the current tenant's ID are copied over.

On the other hand, when the data restoration process is initiated, the following steps are executed to maintain the integrity of tuples that contain data from other tenants. First, the existing data linked to the current tenant in the table where the restore operation is intended is completely removed. This is done so that the data stored in the backup table can be copied as is and appended seamlessly to the remaining tuples within the original table, which are associated with other tenants. This ensures that the restored data smoothly integrates with the existing dataset while preserving the unique data of each tenant.

6.4.13 Creation of audit trails (logs)

The final SQL script also includes dedicated code for auditing purposes. While logging trails cannot prevent unauthorised access or malicious activities, they play a critical role in detecting and accessing such incidents. The systematic recording of every action that brings about changes in the database system, coupled with the capture of the initiator's identity, is crucial for accurately identifying the individuals responsible for compromising the system’s security. The proposed tool records and logs all data modifications in a dedicated table named 'log_activity'. This table includes columns such as username, activity timestamps, client details, performed queries, and

data snapshots to facilitate comprehensive monitoring and analysis. It incorporates a trigger on each table in the generated system to enable logging of insertion, deletion, or update operations on each relation. The trigger is set to execute the function called 'auditlog_trigger' for each affected row. As can be noted from the semantic representation presented in Pseudocode 6.1, this function is responsible for creating a log tuple containing the necessary information and subsequently inserting it into the 'log_activity' table. The specific information stored within the tuple varies based on the type of operation performed, i.e., either INSERT, UPDATE, or DELETE. In an INSERT operation, the data introduced is recorded by comparing the newly stored data in the table (NEW. *) with the previously stored data. Purged data following a DELETE operation, is logged by retrieving the information prior to deletion (OLD. *), and subsequently comparing it with the current data. Conclusively, when the function is triggered by an UPDATE operation, it captures and compares the data before the update (OLD.) with the current data (NEW.) stored in the table, thus recording the modifications made.

Pseudocode 6.1 Audit trail generator

```
1:   Input: void
2:   Output: trigger
3:
4:   Algorithm:
5:     logrow ← ROW(serial_num, TG_TABLE_SCHEMA, TG_TABLE_NAME,
6:     session_user, current_ timestamp, statement_timestamp(),
7:     inet_client_addr(), inet_client_port(), current_query(), TG_OP);
8:     If TG_OP = 'INSERT' then
9:    currentdata ← diff_json(NEW.*, modifiedColumn)
10:    Elsif TG_OP = 'UPDATE' then
11:   currentdata ← diff_json(OLD.*, modifiedColumn)
12:   altereddata ← diff_json(diff_json(NEW.*, currentdata), modifiedColumns)
13:    Elsif TG_OP = 'DELETE' then
14:    currentdata ← diff_json(OLD.*, modifiedColumn)
15:    Else
16:    EXCEPTION 'Operation not stored in audit file'
17:    End if;
18:
19:   INSERT INTO audit_table
20:   VALUES logrow.*
```

6.4.14 Invocation of functions across diverse multi-tenant scenarios

Each function described in the previous section plays a crucial role in the construction of a comprehensive database system. While all functions are required in some form or another, the frequency and parameters required when invoking each function vary based on the specific multi-tenant scenario being considered.

The preceding section points out the main distinctions associated with each function. Notable variations include the frequency of invoking some of the functions such as 'create_database' and 'create_schema'. These are executed either once or repeatedly depending on the specific multi-tenant scenario utilised. Additionally, significant differences can also be observed in the type of backup and auditing procedures utilised as well as the nature and extent of RLS policies required. Furthermore, the incorporation of the ‘tenant ID’ column is evident across all tables in the same table scenario and in common tables of the separate schema. However, this column does not feature in any relation in the separate database scenario.

To effectively manage these variations, six wrapper functions were developed, two for each multi-tenant scenario. Each function has the capability of invoking the scripts specified in the previous section, with suitable criteria and accommodating multiple invocations as necessary, depending on the specific scenario being addressed by the function. Three functions, labelled as 'create_tenant' with prefixes ‘sd’, ‘ss’, or ‘st’ to indicate the type of multi-tenant scenario, are responsible for handling the preliminary tasks preceding the creation of the tenant environment. These are executed by the software provider DBA and involve creating the tenant DBA role, establishing a database, and granting connect privileges to that database in the case of a separate database scenario. For the separate schema and same table scenarios, the tasks involve creating the tenant DBA role and granting connect privileges to a single database without the need for database creation.

The remaining three functions, entitled ‘create_tenant_environment’ and bearing similar prefixes to the aforementioned ones, are also executed by the software provider DBA and are tasked with creating the tenant’s database. This includes the creation of tables, processes, granting of privileges and RLS policies, among other relevant actions specified already in Section 6.4.

6.5 Summary

This chapter focuses on the critical implementation stage. It commenced with the post specifications checks, which involved conducting comprehensive system checks focused on security. These checks aimed to ensure that the database system aligns with the specified security requirements and standards, reinforcing its overall security posture. It also mentions the pre-script generation configuration phase, providing insights into the necessary configuration settings and host file modifications crucial for the operation of a multi-tenant cloud-based system. Finally, this chapter provided a comprehensive exploration of the development of crucial SQL scripts that serve as the foundation of the entire multi-tenant database system. These scripts encompass a wide range of essential tasks, including database and schema creation, table and function setup, RLS policy implementation, view creation, and backup procedures. Collectively, these scripts assume a crucial role in the overall functionality of the proposed tool, with a particular emphasis on guaranteeing data security and optimising data access management. For a better understanding of the entire process, the next chapter provides a detailed step-by-step description through a CASE study, illustrating the complete lifecycle of the proposed tool.

7 CASE Study

7.1 Introduction

This chapter presents a detailed examination of a CASE study designed to validate the veracity and effectiveness of the proposed tool. Two additional CASE studies were conducted and are available for reference in Appendix C.1 and Appendix C.2 respectively. The methodology highlighted in Figure 4.1 (page 59) is followed sequentially for each study and the complete process of designing and creating a secure multi-tenant database system is analysed at full length. Subsequently, distinct user logins are employed, and a series of queries are executed on the established system to test and verify the attainment of the primary objective of the CASE tool i.e., data security.

To enhance the user interface and presentation of the SQL-based CASE tool, a Java-based GUI program was developed. The utilisation of the latter program, instead of the text-based version, was employed to present the outcomes in this chapter. Consequently, this shift towards a visual representation not only enhances the overall user experience but also significantly improves the comprehensibility of results.

The initial part of the study revolves around the development of the required system design diagrams (ERM, ELH, DFD) following a set of pre-defined EBNF rules (highlighted in Chapter 4). These are then scrutinised via a series of checks and used as the basis for the automatic generation of the first-level CRUD matrix. The next part of the study integrates any non-partitioned roles but also focuses on the inputting of partitioned roles both from a vertical and horizontal viewpoint. Conclusively, following the inputting of diverse tenant-specific requirements, the final SQL script report is produced. Reports are generated for each of the three possible multi-tenant scenarios i.e., separate database, separate schema, and same table to ensure a holistic assessment of the proposed tool.

7.2 Setup and configuration

Following the successful generation of the SQL code, execution of the said code took place in both the local environment and the Google Cloud platform. This dual execution strategy ensures comprehensive testing and validation of the code's functionality

and performance. The local execution allows for immediate verification and debugging, while the execution on the Google Cloud platform provides insights into its behaviour in a cloud-based infrastructure.

To set up the Google Cloud environment, several steps were taken. Initially, a Google Cloud project was created, which served as the foundation for hosting and managing the resources. A virtual machine instance, configured with appropriate specifications (defined in Table 8.1(page 149)), was provisioned to ensure optimal performance during code execution. The PostgreSQL database service offered by Google Cloud, known as Cloud SQL, was utilised to host the database. A secure connection was established between the local development environment and the Cloud SQL instance to facilitate seamless code execution and data synchronisation.

By executing the SQL code both locally and on the Google Cloud platform, the research aims to evaluate and compare the code's behaviour and performance across different deployment scenarios. This approach provides valuable insights into the code's applicability to cloud infrastructure and serves as a foundation for assessing its suitability for real-world cloud-based deployments.

In conclusion, following the successful execution of the SQL code, the database was accessed using various user accounts. Multiple queries were then performed to validate that each user could only view authorised data according to their specific permissions. This comprehensive testing process confirmed the effective enforcement of data access controls and provided assurance that the system operated in accordance with the specified security requirements.

7.3 Case Study: Banking System

The case study presented in this chapter exemplifies the utilisation of the proposed tool for the generation of a secure multi-tenant banking system. Typical banking operations particularly those related to customer transactions are depicted. These are then associated with the customer’s accounts, with an accompanying indication of the specific branch where the account was initially created.

7.3.1 System Design Diagrams

The initial stage of the proposed tool involves accepting three system diagram files written in JSON format and adhering to the EBNF rules defined in Chapter 4. These

files are stored with extensions '.erm', '.elh', and '.dfd', respectively and serve as the primary mechanisms for inferring security assertions. Figure 7.1 indicates the ERM diagrammatic representation of the 'Banking System' database.

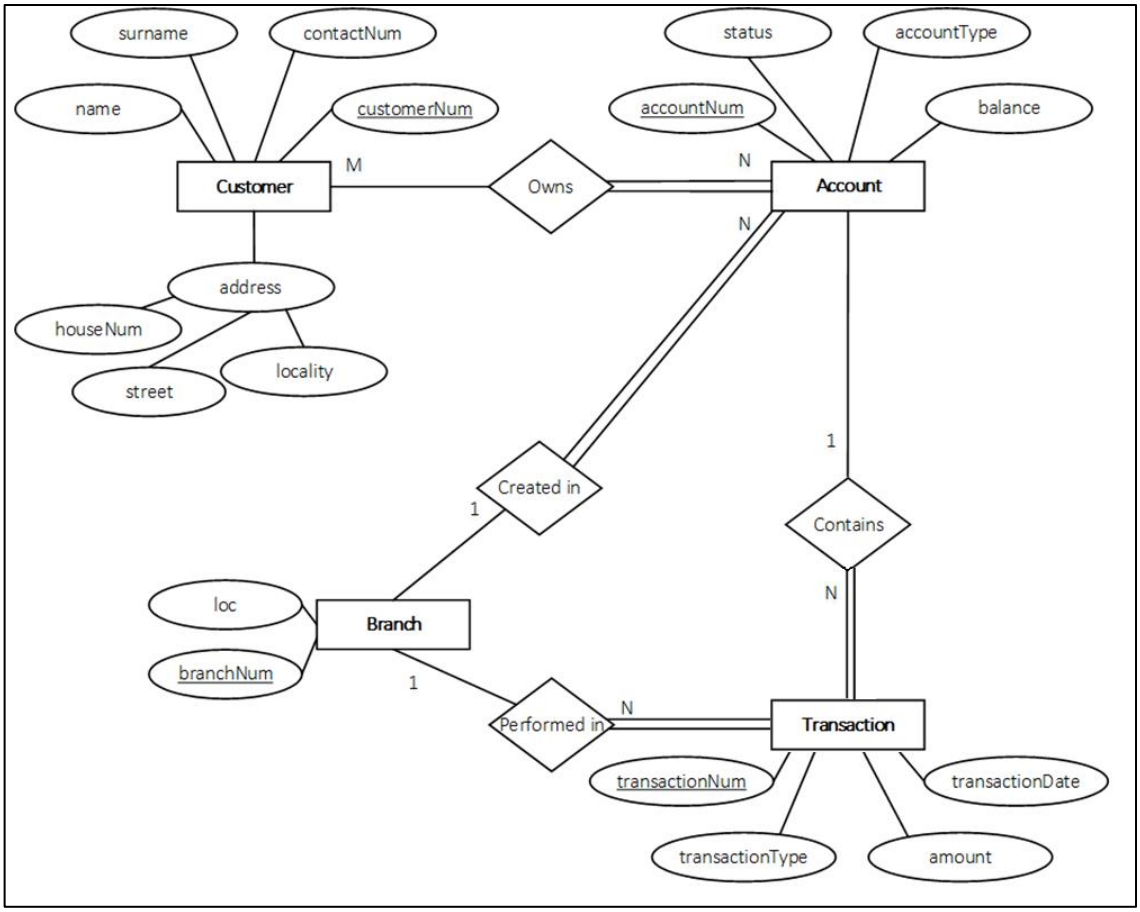


Figure 7.1: Banking System ERM

As evident from the observation of the data structure, this data set consists of four (4) entities, including customers, accounts, branches, and transactions (weak). Additionally, it encompasses a range of relationships with distinct cardinalities, including one-to-one, one-to-many, and many-to-many, comprising both strong and weak associations. Figure 7.2 below denotes part of the JSON presented to the proposed tool. It illustrates the weak entity 'Transaction' and its attributes, 'amount', together with its parent. Additionally, it also portrays one of the relationships in which the 'Transaction' entity participates.

Figure 7.2: JSON-formatted ERM snippet representing a 'Banking' system

One of the attributes present in Figure 7.2 indicates that the ‘amount’ which represents the numerical value associated with a given transaction is of type ‘double precision’ (indicated by the ‘attribute type’ element). This attribute is mandatory (‘required’: ‘yes’), and non-derived i.e., it is not calculated based on other already existent values. Moreover, each field representing this attribute is restricted to a single value and the utilisation of further sub-attributes is not allowed (indicated by the ‘single’ and ‘simple’ attribute mode values). Other attributes can be interpreted in a similar manner. Subsequently, the presented JSON file, also indicates that a ‘SE-WR-WE Relationship’ named ‘Contains’ exists between strong entity ‘Account’ and weak entity

‘Transaction’. The cardinality of the relationship is one-to-many (indicated via the values following the ‘cardinality’ element). The participation of the entities involved in this relationship is contrastive. Whilst, the ‘Account’ entity's participation is ‘partial’, implying that not all its instances participate in the relationship, the participation of the ‘Transaction’ entity is ‘total’.

Subsequently, the processes affecting the states of each entity, were designated, by means of four (4) ELHs, based on the pre-defined EBNF rules (specified in Section 4.5.2). Figure 7.3 and Figure 7.4, illustrate the ELH JSON representation and diagram for the ‘Transaction’ entity. Appendix D.1 contains the diagrammatic representation for the ELHs of entities 'Customer', 'Branch', and 'Account'.

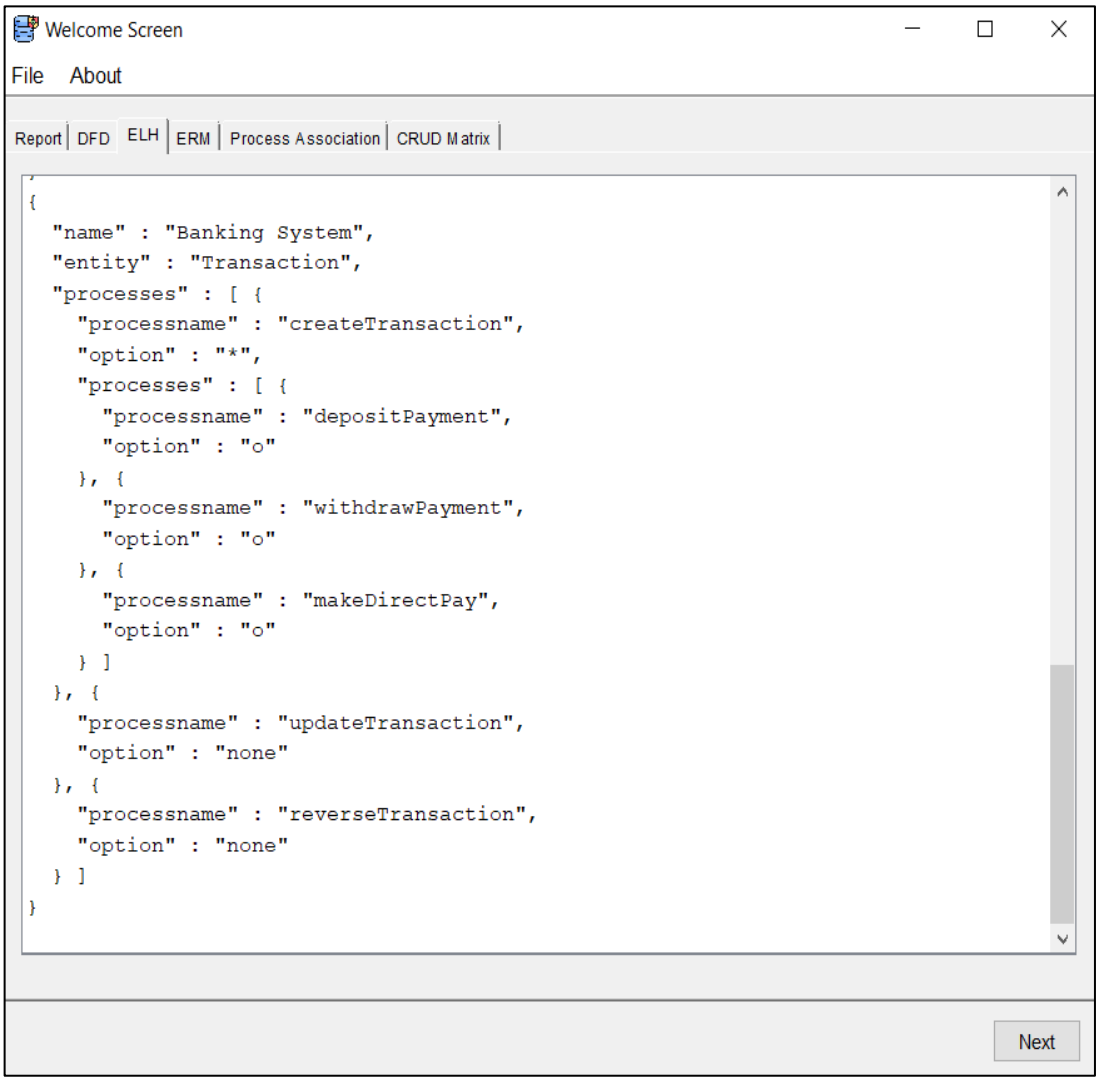


Figure 7.3: Equivalent ELH in JSON

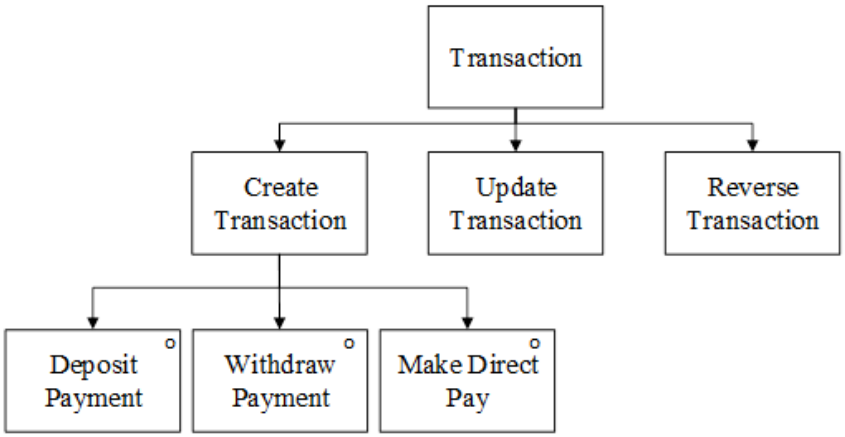


Figure 7.4: 'Transaction' entity ELH in a 'Banking System'

As can be noted from Figure 7.3, the array ‘processes’ is made up of three (3) primary elements. This implies that three main processes affect the state of the ‘Transaction’ entity. Whilst the intermediary and last processes, ‘updateTransaction’ and ‘reverseTransaction’, are responsible for updating and reversing a transaction respectively, the first procedure ‘createTransaction’ handles transaction creation. The latter process consists of a sub-process which further includes three other subprocesses. The values following the ‘option’ element for these sub-processes are set to ‘o’ option, indicating that they are selective, and only one of the three processes can be executed at any given point.

Conclusively, a DFD JSON file, indicating all authorised dataflows in the ‘Banking System’ was presented to the proposed tool. Figure 7.5 displays part of the JSON file, representing three processes and the flow of information between them. The information describing the first dataflow indicates a transfer of data, initiated by the ‘createTransaction’ process (denoted by the ‘outward connected entity’ and ‘outward entity type’ components).

The latter entity passes the required information to the ‘depositPayment’ process (indicated by ‘inward connected entity’ and ‘inward entity type’) so that proper execution of the process can occur. The other two processes can be interpreted in a similar manner.

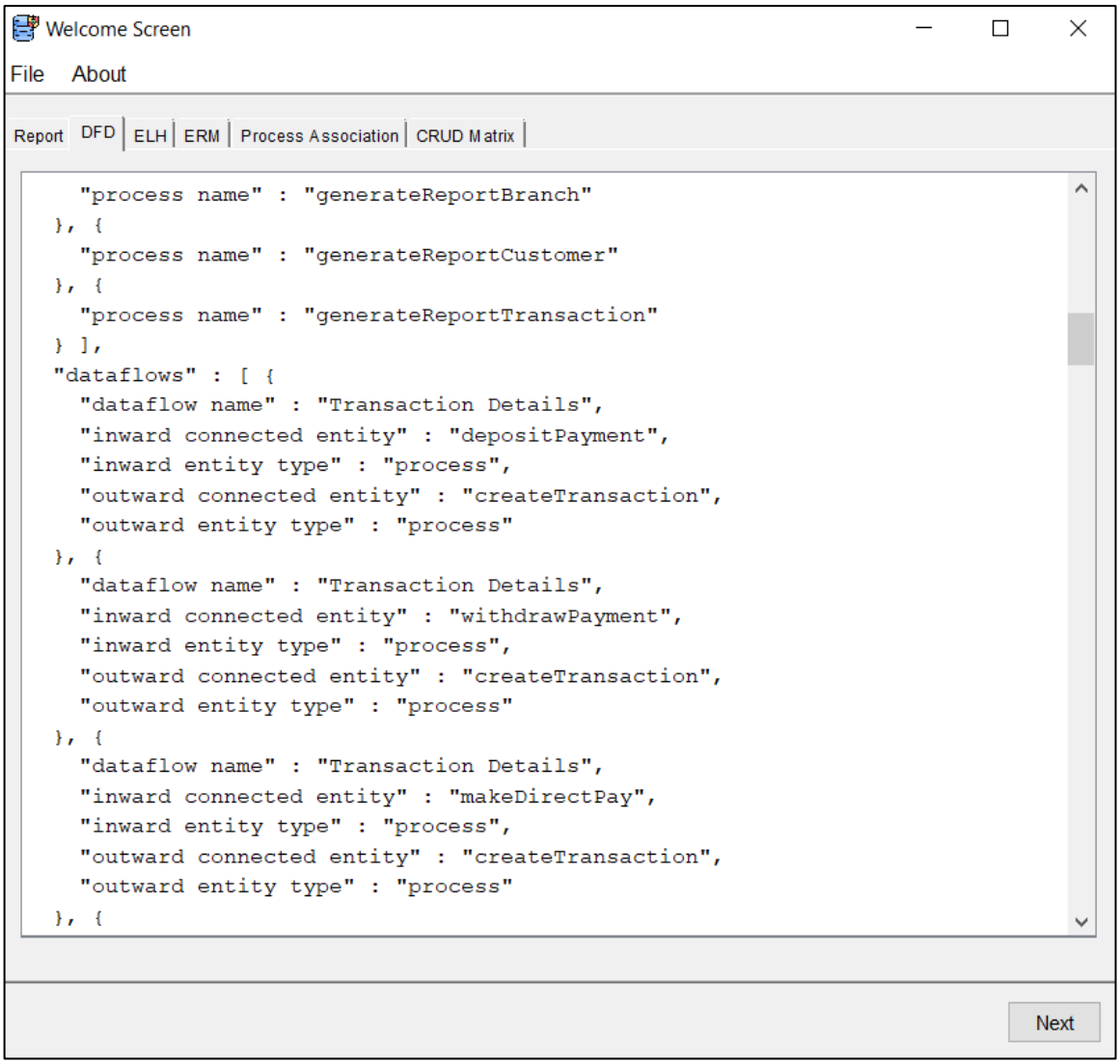


Figure 7.5: JSON-formatted DFD snippet representing a 'Banking' system

In the DFD presented for this case study, alongside traditional external entities, an outside external entity component representing 'customers' is utilised. The storage of this information is segregated because, although customers in this scenario engage with the system, they are not considered part of the system environment. The corresponding diagrammatic representation of the DFD can be observed in Appendix D.2.

7.3.2 Pre and Post checks

Following the inputting of all three system design diagrams, a series of checks, explained in Section 4.8.1, is conducted to establish the initial layer of security. In case of a failed check or the presence of insufficient information, a backtrack is required, and the process should restart once the necessary arrangements are in place. The outcome obtained from each individual diagram check is displayed in Figure 7.6.

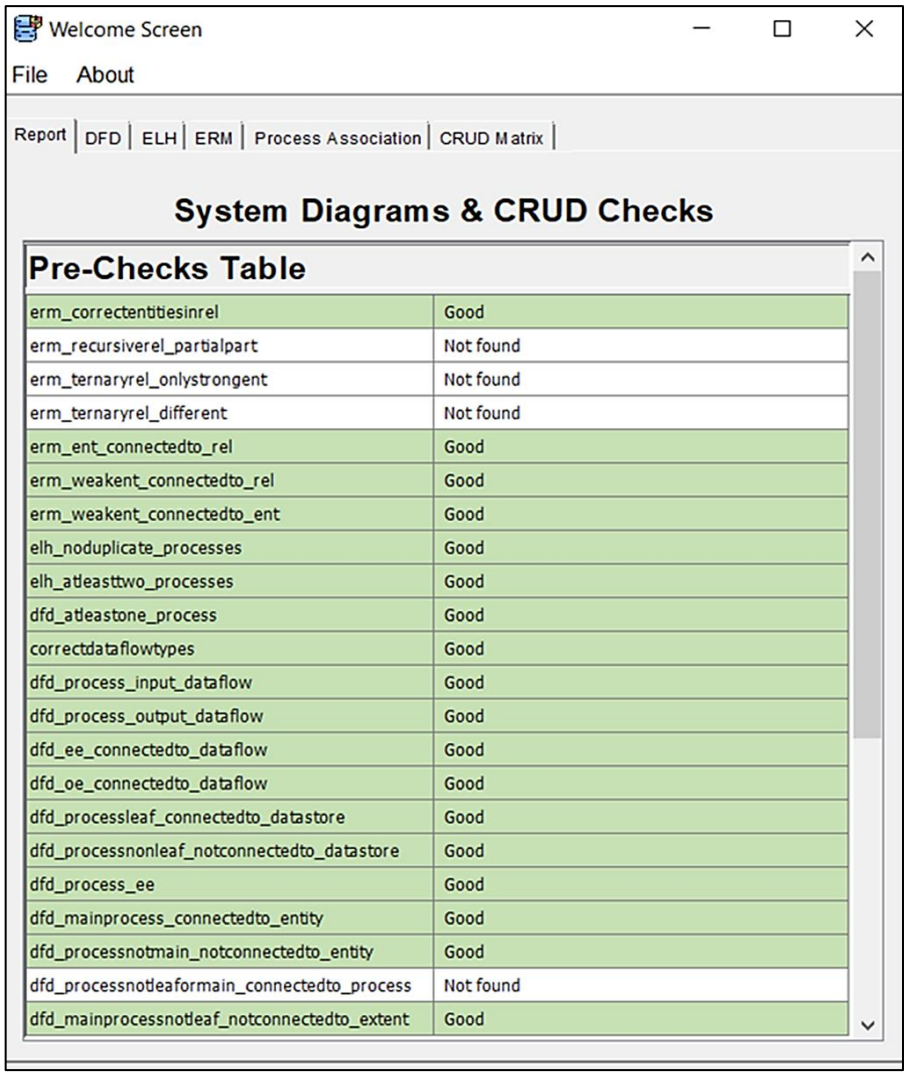


Figure 7.6: Security checks conducted on ‘Banking System’ system diagrams

As can be noted from Figure 7.6, the outcome from each check was either ‘Good’ (green tuples) or ‘Not found’ (white rows). A ‘Good’ indicator implies that the check was executed with no errors. For example, in the ‘Banking System’ being considered, all dataflows in the DFD are connected to at least one process (‘dfd_atleastone_process’ function) and all recursive relationships in the ERM have at least one of the participating entity instances in partial participation (‘erm_recursiverel_partialpart’ process). On the other hand, a ‘Not found’ output result indicates that the aspect verified by the check is not present in the diagrams and hence no inspections are necessary. In this system, no ternary or recursive relationships were detected in the ERM and hence, any checks related to such instances were skipped.

In addition to these two outcomes, two other results could have emerged: 'Warning' (orange tuples) or 'Bad' (red tuples). However, none of the checks yielded

such outcomes in this case study, indicating the successful completion of all operations scrutinised by the checks. Following the positive execution of the pre-checks, four other checks (4) are conducted to assess the cohesiveness of the three inputted system diagrams. Users can only move on to the next phase of the tool, provided that all post checks produce a 'Good' outcome. This is essential both for consistency purposes and to avoid having overlooked security loopholes. Since, as can be observed from Appendix D.3, all four checks yielded a positive outcome ('Good'), generation of the first-level CRUD matrix (described subsequently) can take place.

7.3.3 Generation of first-level CRUD Matrix and related checks

This stage is particularly important due to its close correlation to access controls which have a direct impact on security. Part of the first-level CRUD matrix produced for the 'Banking System', based on the presented system diagrams, indicating the privileges required by each process on the relations present, is shown below in Figure 7.7.

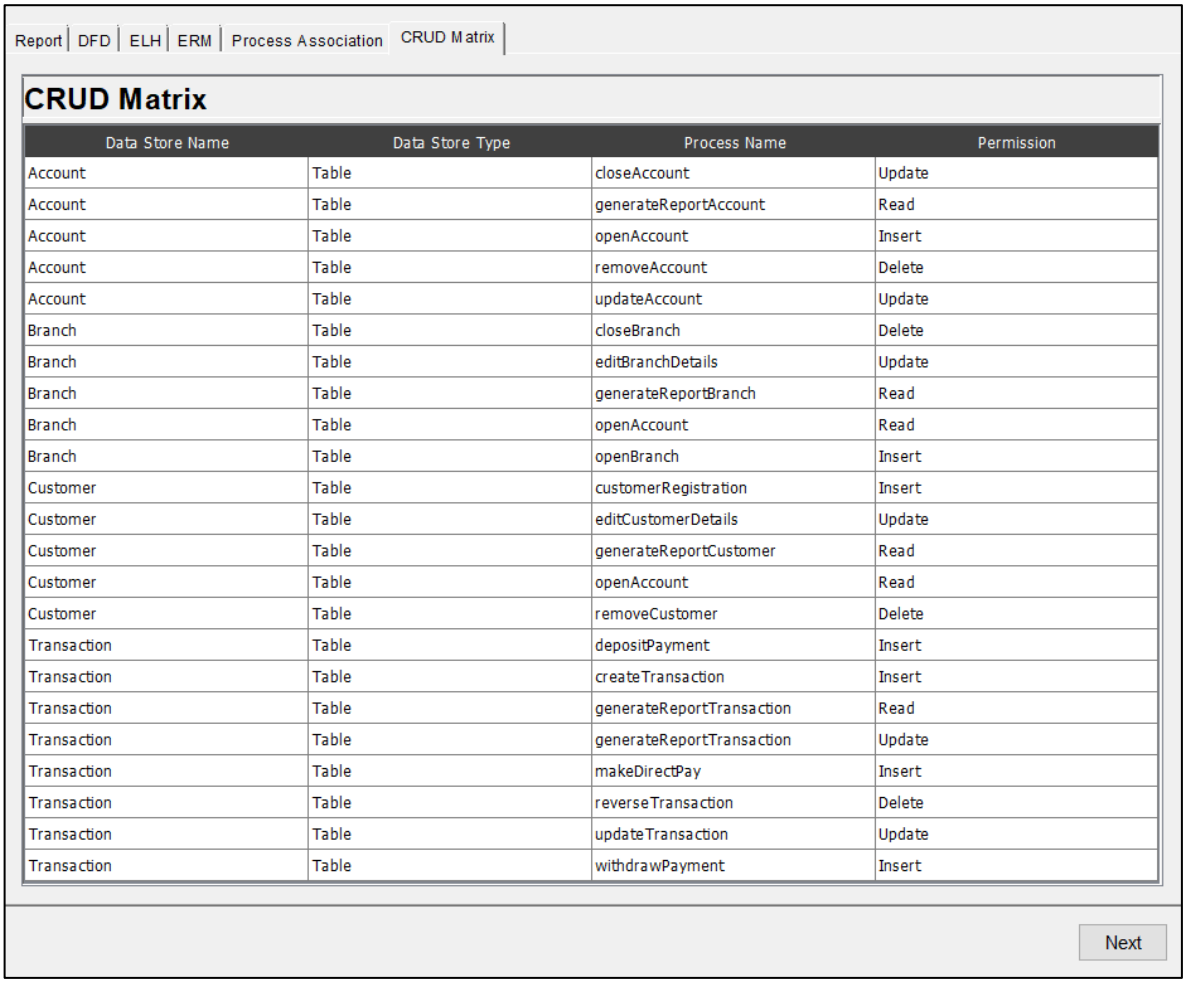


Figure 7.7: First-level CRUD Matrix for 'Banking System'

Since views are not considered in this first-level matrix, all values observed in the ‘crud_ds_type’ column are equal to ‘Table’. The type of permission required on each relation is set to either ‘Read’, ‘Insert’, ‘Update’ or ‘Delete’ and is indicated in the ‘crud_perm’ column. It can be noted from Figure 7.7, that the 'openAccount' process necessitates an INSERT privilege on the 'Account' table, whilst the 'removeAccount' process requires a DELETE privilege on the same table. In contrast, 'editBranchDetails' updates the 'Branch' relation, requiring an UPDATE privilege, while 'generateReportCustomer' retrieves information from the 'Customer' table, needing a SELECT privilege denoted as 'Read' in 'crud_perm'. Other privileges present in the outputted CRUD can be interpreted in a similar manner.

Provided that no exceptions were raised during the generation of the first-level CRUD matrix, a number of checks were conducted on the generated matrix. These inspections serve as a second layer of security, following the checks conducted on the inputted system diagrams. The results obtained following execution of such checks are displayed in tabular format in Appendix D.4.

Positive outcomes were consistently observed in all checks conducted within this section of the case study. Following examination of the first four checks, it becomes evident that all tables possess at least one of the following privileges: SELECT, INSERT, UPDATE, or DELETE. Consequently, it can be inferred that all tables are accessible within the system. Additionally, it can also be deduced that the CRUD matrix inclusively encompasses all the tables and processes specified in the preceding system design diagrams.

7.4 Association of non-partitioned roles and processes

Provided the successful execution of the checks delineated in the prior section, the general roles interacting with the system and the processes necessary for them to effectively perform their tasks are identified. This information is crucial when assigning appropriate process privileges to roles. As discussed in Section 5.4.5, this role-process corelation is derived from the presented DFD. Figure 7.8 illustrates the associations between roles and process for the ‘Banking System’.

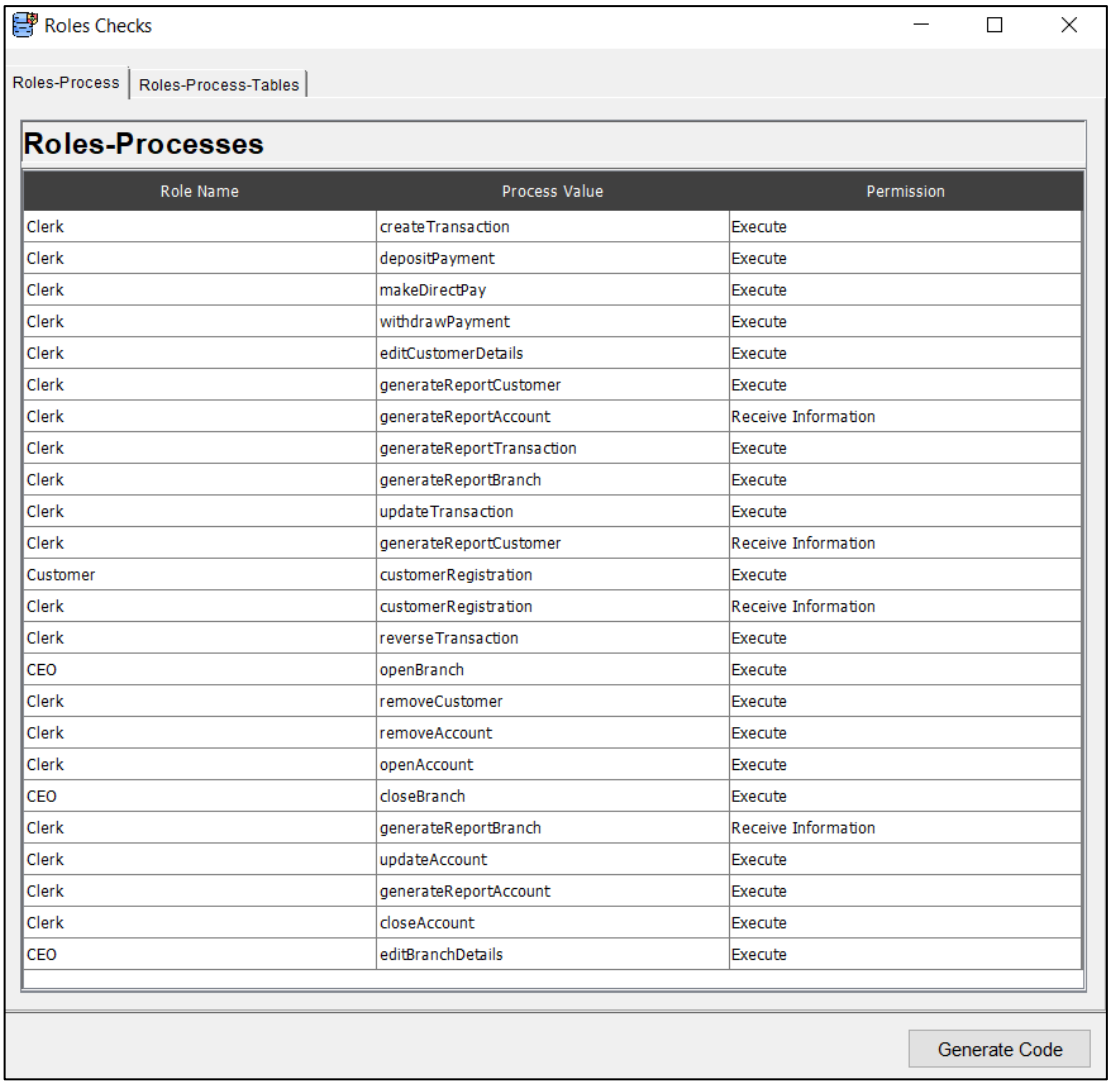


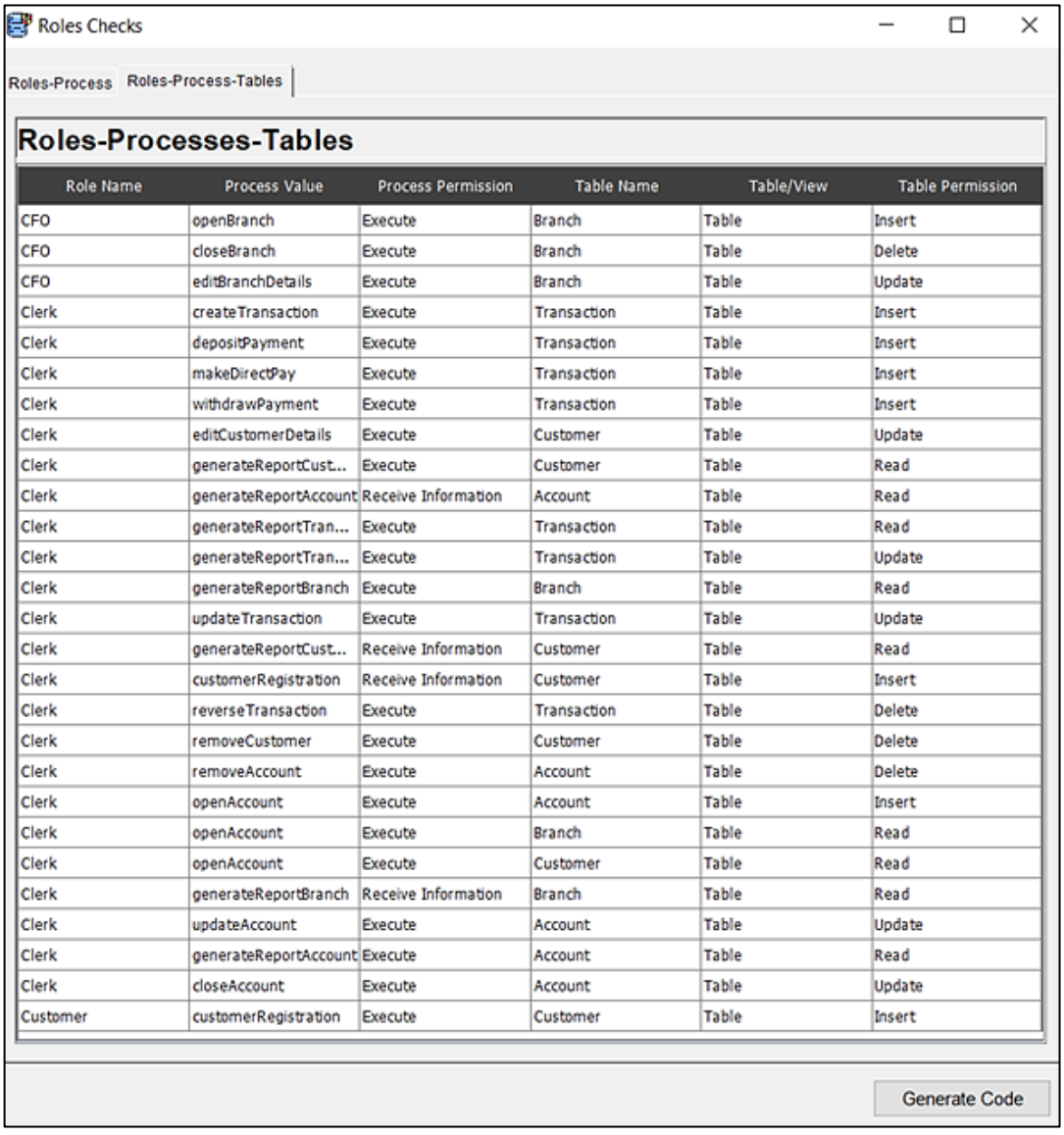
Figure 7.8: Association of roles and processes for ‘Banking System’

The successful generation of the prior matrix, guarantees the linkage of every role to at least one function, ensuring the essential security aspect of role-system interaction. Moreover, it also ensures that the converse is true (i.e., all functions are associated with at least one role), preventing redundancy.

As depicted in Figure 7.8, the 'Clerk' role necessitates access to processes involved in day-to-day bank transactions and operations. These include, for example, the creation of transactions, editing of customer details, and generation of reports. In contrast, the 'Customer' entity, as an outside external entity, should have the capability to register, while the CFO is responsible for executing higher-order functions such as branch opening or closure. Other role-processes associations can be deduced in a similar manner from the same matrix.

7.4.1 Creation of second-level CRUD matrix

The generation of the matrix described in Section 7.3.3 facilitated the construction of a comprehensive second-level CRUD Matrix. While the first-level CRUD Matrix (Figure 7.8) primarily depicts the relationships between processes and tables, the second-level CRUD Matrix introduces an additional dimension by incorporating roles. This inclusion enables the deduction of specific privileges associated with each role for every table. Figure 7.9 illustrates the revised CRUD matrix for the Banking System.



Role Name	Process Value	Process Permission	Table Name	Table/View	Table Permission
CFO	openBranch	Execute	Branch	Table	Insert
CFO	closeBranch	Execute	Branch	Table	Delete
CFO	editBranchDetails	Execute	Branch	Table	Update
Clerk	createTransaction	Execute	Transaction	Table	Insert
Clerk	depositPayment	Execute	Transaction	Table	Insert
Clerk	makeDirectPay	Execute	Transaction	Table	Insert
Clerk	withdrawPayment	Execute	Transaction	Table	Insert
Clerk	editCustomerDetails	Execute	Customer	Table	Update
Clerk	generateReportCust...	Execute	Customer	Table	Read
Clerk	generateReportAccount	Receive Information	Account	Table	Read
Clerk	generateReportTran...	Execute	Transaction	Table	Read
Clerk	generateReportTran...	Execute	Transaction	Table	Update
Clerk	generateReportBranch	Execute	Branch	Table	Read
Clerk	updateTransaction	Execute	Transaction	Table	Update
Clerk	generateReportCust...	Receive Information	Customer	Table	Read
Clerk	customerRegistration	Receive Information	Customer	Table	Insert
Clerk	reverseTransaction	Execute	Transaction	Table	Delete
Clerk	removeCustomer	Execute	Customer	Table	Delete
Clerk	removeAccount	Execute	Account	Table	Delete
Clerk	openAccount	Execute	Account	Table	Insert
Clerk	openAccount	Execute	Branch	Table	Read
Clerk	openAccount	Execute	Customer	Table	Read
Clerk	generateReportBranch	Receive Information	Branch	Table	Read
Clerk	updateAccount	Execute	Account	Table	Update
Clerk	generateReportAccount	Execute	Account	Table	Read
Clerk	closeAccount	Execute	Account	Table	Update
Customer	customerRegistration	Execute	Customer	Table	Insert

Figure 7.9: Second-level CRUD Matrix for ‘Banking System’

Thus far, all values in the ‘rpc_tableorview’ column are set to 'Table' as non-partitioned roles have been the primary focus. The exploration of partitioned roles will

be addressed in the upcoming section. Taking the eighth tuple from Figure 7.9 as an example, it is evident that role 'Clerk' is granted authorisation to execute process 'editCustomerDetails'. However, this permission is conditional upon possessing UPDATE privileges for Table 'Customer', as the process necessitates that privilege for its proper execution. Similarly, upon inspecting other remaining tuples, it is evident that the same role also possesses the capability to execute the 'openAccount' process. However, to successfully perform this task, the role necessitates SELECT privileges on both the 'Branch' and 'Customer' tables.

7.4.2 Matrix extension to include horizontal and vertical partitioning

As discussed in Section 5.6, different tenants may necessitate users with identical roles to access distinct sets of data or tuples, even from the same table and following the execution of the same process. Until now, the focus has been on non-partitioned roles, as this information was directly extracted from the presented system diagrams. However, the proposed tool offers an enhanced capability by allowing tenants to provide relevant details about horizontal or vertical partitions associated with specific roles. The partitioned JSON file utilised for this case study is illustrated in Figure 7.10.

```
{
"name": "Banking System",
"roles": [{
      "rolename": "Clerk",
      "verticalPartition":[{
         "tablename": "Transaction",
         "columnname": "transactionDate"
      },
      {  "tablename": "Transaction",
         "columnname": "transactionType"
      }],
      "partitionBy":[{
         "tablename": "Transaction",
         "columnname": "transactionType"
      }]},
      {
      "rolename": "CFO",
      "partitionBy":[{
         "tablename": "Branch",
         "columnname": "location"
      }]
      }]
}
```

Figure 7.10: Partitioned roles for 'Banking System'

This file indicates the existence of a horizontal partition on the 'location' column of the 'Branch' table for the 'CFO' role, as well as another partition on the 'transactionType' column of the 'Transaction' table roles of type 'Clerk'. Additionally, there is a vertical partition on the 'Transaction' table specific to the 'Clerk' role, restricting access solely to the 'transactionDate', and 'transactionType' columns. Given the absence of any additional information, it can be deduced that the other role presented in the DFD, within the 'Banking System' i.e. Customer, is not subjected to partitioning.

Following the insertion of these partitions, the 'rolesprocesses' and 'rolesprocessestables' tables mentioned in Section 7.3.3 and Section 7.4.1 respectively were updated to incorporate these new components. Consequently, the matrix not only includes 'Tables' data structures but also broadens its scope to incorporate 'views'. Subsequently, verification checks were carried out to ensure the accuracy and integrity of the inserted partitions and once these were passed generation of the final SQL script could take place.

7.4.3 Tenant-specific requirements and generation of SQL Code

Prior to the generation of the final SQL Code representing a secure database system, tenants are given the opportunity to outline their specific requirements. These include preferences related to the authentication mode, as well as their desired approach to multi-tenancy. To ensure the proper functionality of the proposed tool across different scenarios, code was generated for all three possible multi-tenant approaches, taking into account the specifications stated in the corresponding ‘Requirements’ file.

7.4.3.1 Separate Database multi-tenancy approach

To validate and assess the functionality of the generated code for this multi-tenant approach, the tenant’s name was set to ‘aps_sd’ and the requirements were set such that a separate database multi-tenant approach was used, with ‘UTF8’ database encoding, ‘md5’ authentication, and a password expiry set for December 2023. Following inputting and validation of these requirements, the ‘generate code’ button is activated, and the SQL code in Appendix D.5 is displayed.

The code snippet displayed in Figure 7.10 displays part of the generated code responsible for the creation of the 'Account' table. The absence of a 'tenant ID' field or RLS policies can be observed, which can be attributed to the dedicated database

approach where each tenant operates independently with their own set of schemas or tables. This stands in contrast to the other two multi-tenant scenarios, where sharing of resources among tenants is involved.

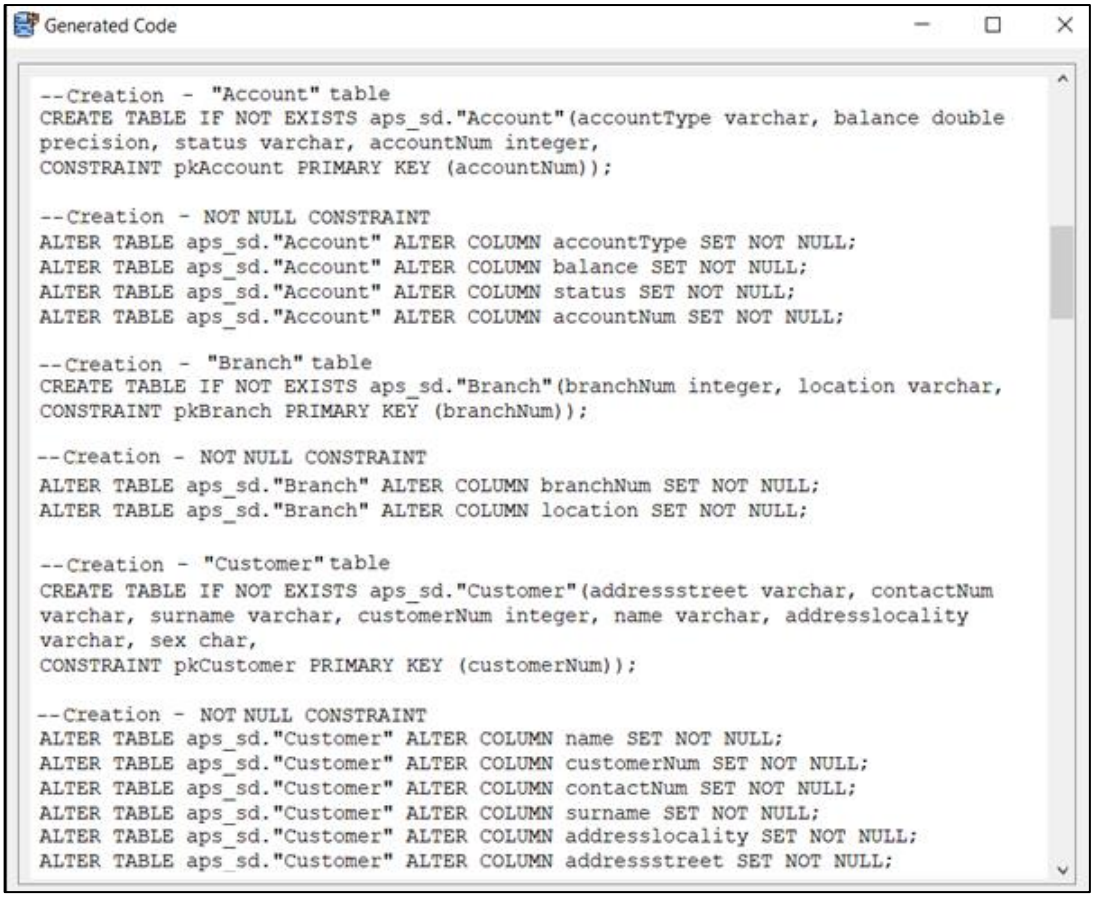


Figure 7.11: Part of the SQL Code generated for separate database scenario

7.4.3.2 Separate Schema multi-tenancy approach

For consistency purposes, the same 'Requirements' file used in Section 7.4.3.1 was utilised to generate the SQL code corresponding to the 'separate schema' approach. The only modifications made to the configuration were setting the multi-tenant approach parameter to 'separate schema' and the database encoding to NULL. The latter change was implemented to align with the specific requirements of the 'separate schema' approach, where the tenant does not have the option to choose the database encoding as the database is only created once at the beginning.

The main variations include the omission of database creation since the same database is used for all tenants and the utilisation of a ‘tenant ID’ in tables present in

common schemas. The code below (Figure 7.12) illustrates a snippet of the code generated for this approach, related to the assignment of SELECT privileges to roles. Some of these privileges were granted on tables, while others were assigned to views, depending on whether the affected role is partitioned or otherwise.

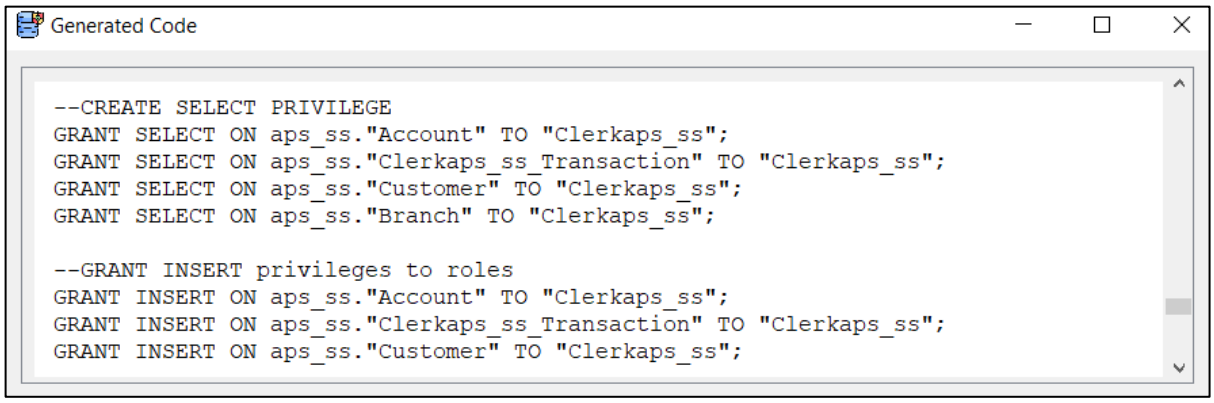


Figure 7.12: Part of the SQL Code generated for separate schema scenario

7.4.3.3 Same Table multi-tenancy approach

In conclusion, SQL code was generated for the 'same table' multi-tenant approach within the context of the same case study. The requirements provided for this approach remained consistent with those of the previous approaches, except for the adjustment made to the first element, specifying the multi-tenant approach as 'same table'.

The code excerpt in Figure 7.13 demonstrates part of the generated code related to the creation of an RLS policy on the 'Branch' relation. These policies are implemented to effectively differentiate and control access for different tenants interacting with the table. In the 'same table' approach, all tables are equipped with a minimum of one Row-Level Security (RLS) policy, specifically designed to achieve this objective. Additional RLS policies can be implemented based on specific tenant-based partitions. For instance, as discussed in Section 7.4.2, the 'CFO' role in this case study is horizontally partitioned based on the 'location' column of the 'Branch' table. This is reflected in the generated code, where an RLS policy is specifically designed to cater to this partition, allowing for precise and fine-grained access control.

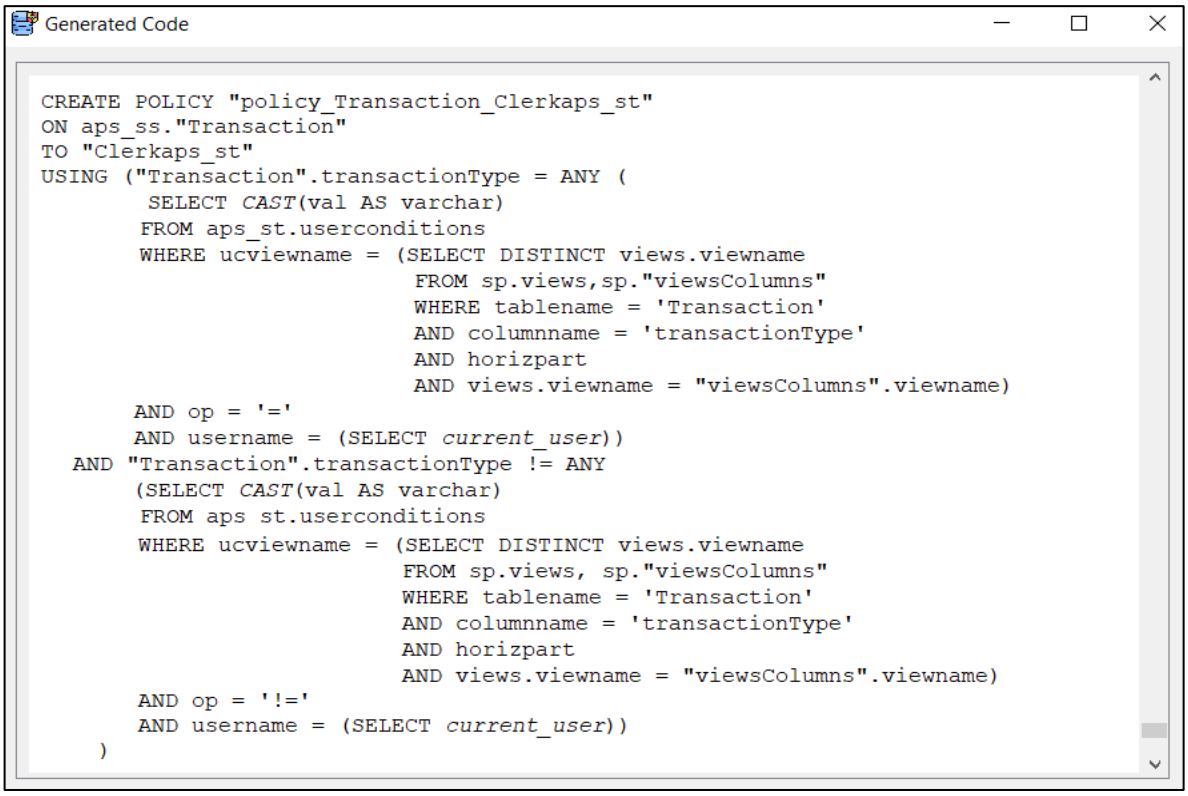


Figure 7.13: Part of the SQL Code generated for same table scenario

7.4.4 User-role associations and possible partition conditions

After generating the SQL code based on the chosen multi-tenant approach, it is executed on both local and cloud-based platforms (Google Cloud in this case) and the tenant environment is set up. Provided that the process is successfully completed, the tenant DBA can proceed to create user accounts for the employees and assign them appropriate roles. In this analysis, three user accounts ('Peter', 'Jane', and 'Tom') were created and assigned to the roles of 'Clerk', 'Customer', and 'CFO' respectively.

During the process of assigning users to roles, any conditions related to horizontal partitions must be specified. Two horizontal partitions are associated with this case study (Section 7.4.2). One is related to the 'CFO' role and is based on the 'location' column of the 'Branch' table, and the other pertains to the 'Clerk' role and is centred upon the 'Transaction type' column of the 'Transaction' table. As can be noted from Figure 7.14 and Figure 7.15 below, in the context of this study, it was specified that ‘Peter’, designated as a ‘Clerk’, is authorised to view tuples where the Transaction Type is 'Cash Withdrawals', while ‘Tom’, appointed as a ‘CFO’, is permitted to access information exclusively related to the branch located in 'Valletta'.

Figure 7.14: User conditions for user ‘Peter’ designated as a Clerk

Figure 7.15: User conditions for user ‘Tom’ designated as a CFO

Post-installation, additional users can be created and linked to specific roles as needed. Moreover, these users can undergo updates to refine their specifications or be removed as required, granting a high degree of flexibility in user management. The same level of adaptability extends to user conditions, provided they remain aligned with the predefined partitions associated with their respective roles. This implies that, apart from inserting user conditions, one can also update or delete them as necessary.

Inclusion and changes to users and user conditions represent the sole post-installation modifications permissible in this proposed tool. If any alterations are made to processes, tables, or roles, the entire process must be restarted from the beginning due to its impact on previously completed security checks and matrix generations.

7.4.5 User Login and Function Execution

Following completion of the preceding section, tenant users are granted the capability to proceed by logging in and commence active system utilisation. In order to validate the exclusive display of authorised information to users of each tenant, comprehensive testing was conducted by logging in as each of the three user accounts specified in Section 7.4.4 and executing a random series of functions.

The login process involved entering the appropriate username, password, and tenant details. Commencing with the user account assigned to 'Peter' in his capacity

as a 'Clerk', upon entering the relevant credentials, the authentication process was initiated, subsequently revealing the functions corresponding to 'Peter's' assigned access privileges. For this particular case study, the deliberate execution of two specific functions took place: 'generateTransactionReport' related to the 'Transaction' table, and 'close account' linked to the 'Account' table. Following the execution of the first function, which entails accessing tuples within the 'Transactions' relation, the expected behaviour dictates that only tuples containing information related to 'Cash Withdrawals' should be displayed. This expectation stemmed from the horizontal partitioning that was enforced. Additionally, due to the vertical partitioning restrictions imposed on this particular user account, 'Peter' was expected to have access solely to the ‘transactionNum’, ‘transactionDate’, and ‘transactionType’ columns of the table. This expectation was indeed confirmed, as illustrated in Figure 7.16 below.

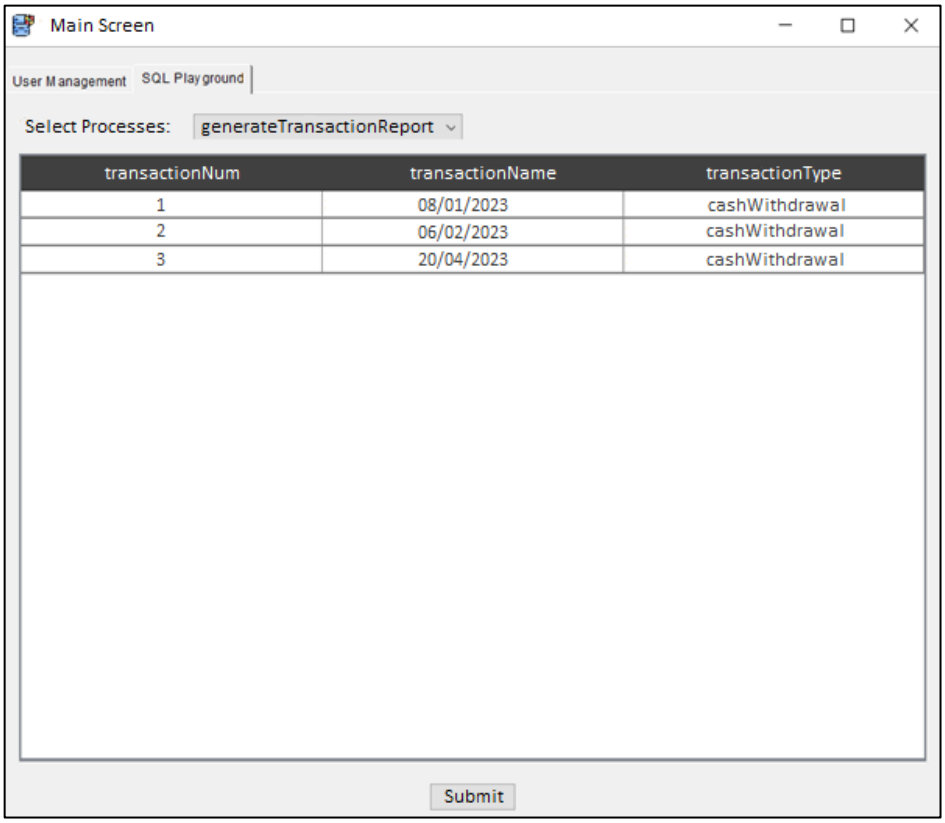


Figure 7.16: Execution of ‘generateTransactionReport’ by a Clerk

In contrast, given the absence of partitioning in relation to the 'Account' table, 'Peter' is expected to encounter no limitations when executing the second function.

As a 'Clerk', 'Peter' has unrestricted access to delete any tuple within the 'Account' table. This expectation was confirmed as 'Peter' was able to delete various accounts with distinct information without encountering any issues.

Next, the account linked to 'Jane', identified as a customer, was subjected to testing. Aligned with the information presented in the DFD, 'Jane' is restricted to accessing only the 'customerRegistration' process. In fact, upon authentication, 'Jane' was presented exclusively with the 'customerRegistration' process. Since there are no partitions associated with the role ‘Customer’, she was granted permission to insert information pertaining to all column values in the 'Customer' relation without encountering any restrictions.

In conclusion, login credentials were successfully provided for 'Tom,' who holds the position of 'CFO.' According to the Data Flow Diagram (DFD), 'Tom,' in his role as 'CFO,' possesses the authority to execute one of three processes associated with the 'Branch' relation. For this specific case study, the 'openbranch' process was chosen, and an attempt was made to insert a tuple with the location 'Mosta.' However, because 'Tom' does not have authorisation to access such a branch, the insertion request was denied, as illustrated in Figure 7.17. Conversely, when attempting to insert another tuple with the location specified as 'Valletta' (in compliance with the imposed restrictions), the insertion was successful, as depicted in Figure 7.18.

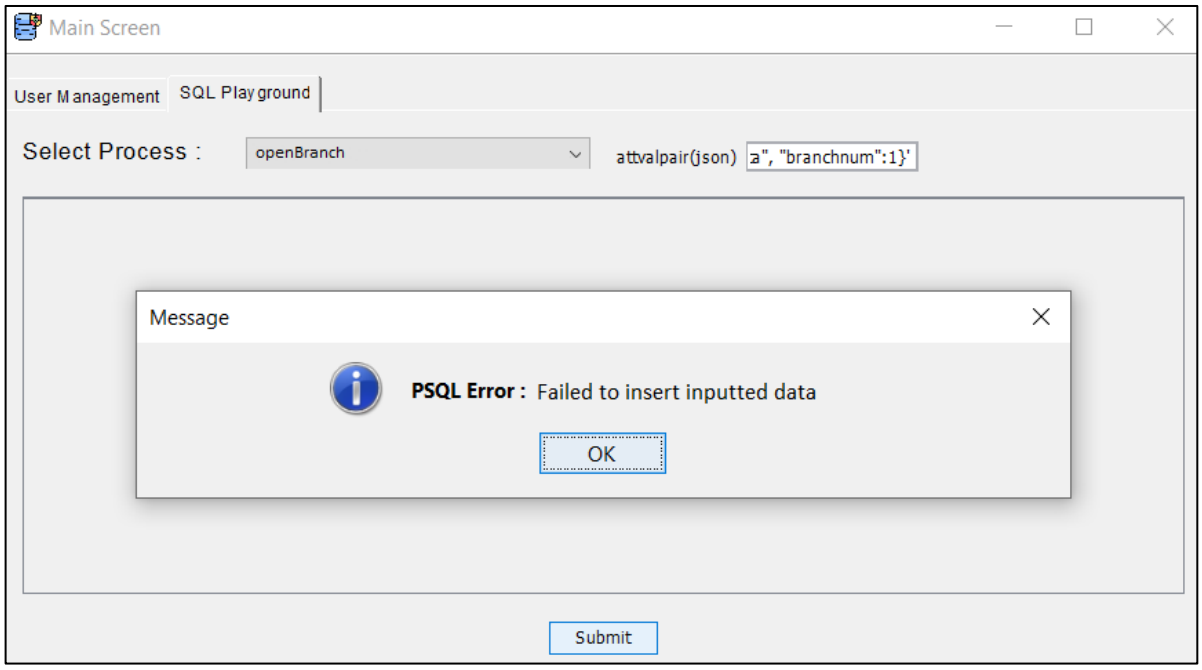


Figure 7.17: Execution of ‘openBranch’ function by a Clerk (unauthorised values)

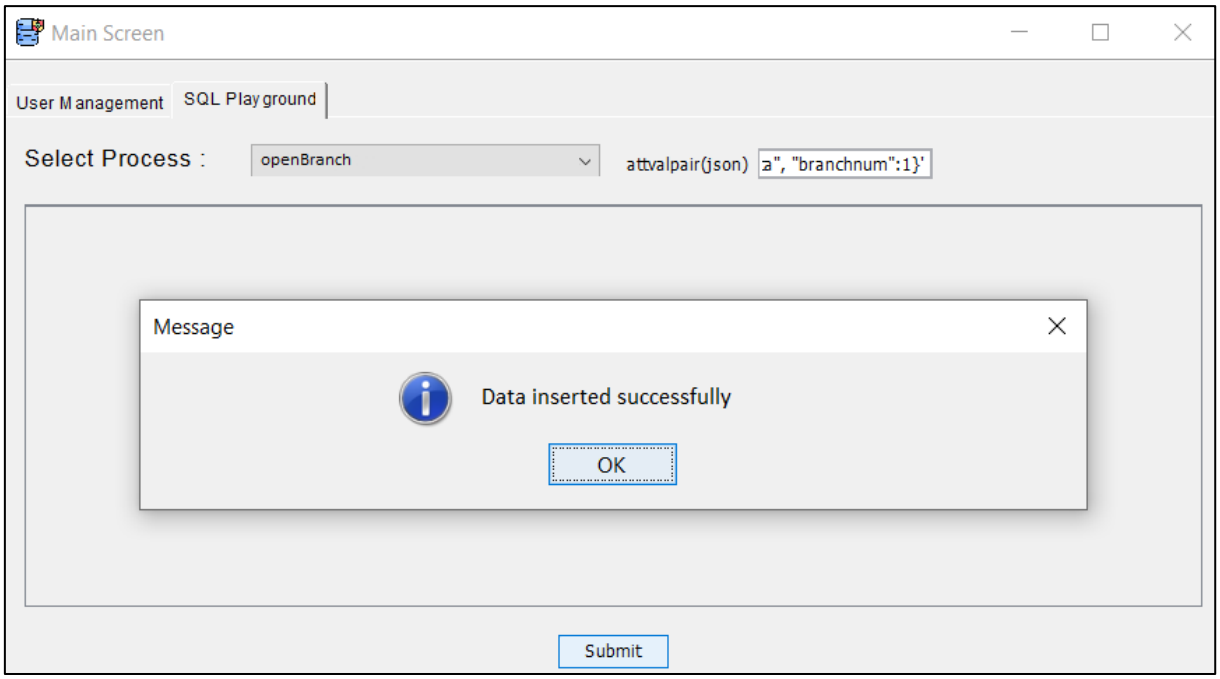


Figure 7.18: Execution of ‘openBranch’ function by a Clerk (authorised values)

7.5 Summary

This chapter provided an extensive analysis of a CASE study conducted on a 'Banking System', aimed at validating the accuracy and efficacy of the proposed tool. The outlined methodology, as depicted in Figure 4.1 (page 59), was meticulously followed in a step-by-step manner, thoroughly describing the entire process of designing and establishing a secure multi-tenant database system. Subsequently, diverse user logins were utilised and a sequence of queries was executed on the implemented system to assess and validate the achievement of the primary objective of the CASE tool, namely data security. The proposed tool demonstrated the anticipated functionality and proved to be successful in its intended purpose.

8 Evaluation

8.1 Introduction

This chapter evaluates the effectiveness and efficacy of the proposed CASE tool vis-a-vis the aim and objectives outlines in Chapter 1. The evaluation was conducted in the context of:

- **System coverage** – are the inputs provided to the CASE tool sufficient in developing a holistic, secure multi-tenant database system?
- **Performance** – How does the tool's performance compare in terms of latency and throughput between secure database systems generated by the tool and corresponding systems devoid of such features?
- **Security** – does the tool really fulfils its aim of providing a security profile for each tenant?

A systematic approach combining both qualitative and quantitative techniques is adopted during the evaluation process to comprehensively assess the tool's above-mentioned criteria. The evaluation results, encompassing data collection, analyses, and conclusions, are presented below. These findings offer insights into the tool's capabilities and limitations, enabling informed decisions regarding its applicability in real-world software development projects.

8.2 System coverage Evaluation

A system coverage evaluation was conducted to assess whether the inputs provided to the CASE tool are sufficiently capable of generating a comprehensive and secure multi-tenant database system and related functionality. Thus, it is imperative to confirm that the data contained in the three input diagrams can effectively represent systems in a holistic manner. According to the widely used Structured Systems Analysis and Design Method (SSADM) methodology and supported by literature works by Sterman [162], and Haimes [163], a holistic system considers multiple viewpoints such as functionality, structure, and transition over time. These viewpoints are vital in designing a system that addresses stakeholder (in this case the software house) needs and can adapt to changing requirements. The evaluation confirmed the importance of

an all-encompassing approach in developing a robust and adaptable system. This section hereunder illustrates how the input to our proposed tool is in-line with the definition of a holistic system as stated in SSADM:

- **Functional -** indicating the main functions and flow of information between them. A DFD is ideal for this representation as it encapsulates system entities, processes, and data stores whilst indicating information flow via dataflows.
- **Structural –** referring to the component organisation and corresponding associations. This viewpoint is addressed by ERM diagrams which delineate the entities contained in the system and their corresponding relationships.
- **Time-based -** the time-based perspective depicts the main sequence of events associated with each system component. The proposed tool encapsulates this perspective by means of the ELH diagram which indicates via processes, the potential lifespan of every entity from creation to deletion.

Hence, it can be inferred that employing ERM, ELH, and DFD diagrams is a suitable approach to depict a comprehensive system encapsulating each of the three elements mentioned previously. A comprehensive analysis hereinafter illustrates the primary strengths and weaknesses of each diagram, while also drawing comparisons with alternative methodologies that may have been utilised.

ERMs offer advantages in representing complex data scenarios with simplicity and can be easily transformed into relational database systems, which is beneficial for our tool. Notwithstanding such advantages, the proposed tool may not be suitable for software developers working with advanced database concepts like aggregation, composition, specialisation, or generalisation since these require a modified version of ERM. ELHs have the strength of selectively reflecting database system modifications without affecting other ELHs. Together with DFDs, they are capable of validating system requirements and retreieving business rules, but do not model time-dependent behaviour or triggers which might affect the system's primary course of action.

An alternative to the mentioned ERM, ELH, and DFD approaches is the utilisation of Unified Modelling Language (UML) diagrams [164]. UML can be adapted for database design, with the Class Diagram serving as a substitute for ERM. However, these are aimed at classes and associations in a program, while ERM diagrams depict entities and connections in a database. Given the objective of this dissertation to build

a CASE tool for database systems, ERM diagrams are more suitable. UML sequence diagrams are a suitable alternative to ELH diagrams, focusing on process execution order whilst use case diagrams in UML are comparable to DFDs but lack a holistic overview and do not include data stores [165]. Therefore, DFDs are considered a more suitable formal model for the system, while Use Case diagrams may be useful as initial drafts during analyst and client discussions.

The following table delineates the primary distinctions between the system diagrams utilised for the proposed tool and their counterparts within the UML family.

Table 8.1: Comparison of system diagrams and UML-based ones

System Diagrams used for proposed tool	**Equivalent UML Diagrams**	**Main Distinctions**
ERM	Class Diagrams	**ERMs** focus on depicting entities and connections within a database system, while **class diagrams** primarily represent classes and their relationships in a program.
ELH	Sequence Diagrams	The **ELH** diagram depicts the processes involved for a database entity's state changes over time, whereas a **sequence diagram** primarily outlines the sequential flow of processes or events within a system.
DFD	Use Case Diagrams	**Use case diagrams** lack the inclusion of data stores, resulting in a less comprehensive overview, making them better suited as initial drafts. Conversely, **DFDs** offer a more holistic representation, making them a better choice for a formal modelling approach.

Following the evaluation outlined in this section, it was concluded that the ERM, ELH, and DFD are the most appropriate diagrams for the proposed CASE tool.

Albeit simple in nature, such diagrams are highly effective due to their ability of representing holistic systems whilst having negligible impact on the additional burden experienced by the software provider.

8.3 Performance Evaluation

Following the qualitative study discussed prior, a quantitative assessment was conducted to evaluate the tool's performance under different workloads and scenarios. The assessment was performed in two distinct environments: a local configuration and a cloud setup. In the local environment, the system consisted of an 8 GB DDR4 RAM, 256 GB SSD storage, and a 2.7 GHz Core i7 processor running Windows 10.

Evaluation on the cloud was conducted on the Google Cloud Platform with two distinct data centres: "europe-north1" and "asia-east2" each housing a single server with hardware specifications listed in Table 8.1. These servers executed PostgreSQL instances and workloads using the 'pg_bench' utility [176]. The consistency in infrastructure across local and cloud environments allowed for isolating the impact of cloud execution on tool performance.

Table 8.2: Google Cloud Evaluation Specifications

Server	OS	CPU	RAM	Disk	Region
Europe	Windows Server 2016	vCPUs: 2@ 2.7 GhZ	8 GB	Zonal SSD	europe-north1
Asia	Windows Server 2018	vCPUs: 2@ 2.7 GhZ	8 GB	Regional SSD	asia-east2

A comparison was conducted between systems with implemented security features and identical ones without security controls to evaluate the proposed system's effectiveness and identify potential overheads. The objective was to demonstrate the benefits of the secure approach while maintaining system performance.

When conducting evaluations of this nature, it's important to acknowledge the potential influence of validity threats on results. One aspect to consider is sampling bias; the selection of servers located in Europe and Asia, selected for this dissertation, provides insights limited to those continents. Consequently, executing the same queries on servers located in other continents may yield slightly different timings.

External factors, such as variations in network conditions and fluctuations in demand, can impact the validity of results by influencing the recorded performance timings. Despite efforts to maintain consistent infrastructure to minimise external influences, factors like network variations and demand fluctuations are beyond control.

Additionally, considering provider-specific factors is crucial when evaluating cloud performance. Each cloud provider may offer unique features, optimisations, or performance characteristics that can influence outcomes. The evaluation of the proposed tool was conducted on Google Cloud, which may exhibit specific attributes distinct from other providers. Understanding these provider-specific factors is essential for accurately interpreting and generalising findings.

8.3.1 Establishing a Benchmark Baseline

A baseline benchmark was established to measure the initial state of the system before any improvements were made. The database systems 'Scott', 'Banking System', and 'School Management System' were used without any security features. These systems were subjected to various workloads using the 'pg_bench' utility, both locally and on the cloud. The experimental setup for cloud service benchmarking proposed by Bermbach et al. [192] (Figure 8.1), was followed, with a minor modification for local execution, where the deployment stage in the ‘Pre-Benchmark Run’ was omitted.

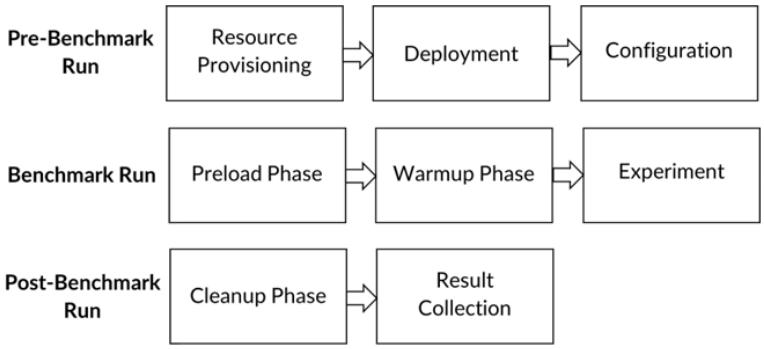


Figure 8.1: Benchmark Experimental Setup and Process [192]

A workload of 100,000 operations was executed on each of the three system baselines, both locally and on the cloud. The workload included customised Create, Select, Insert, Update, and Delete operations tailored to each system's requirements. These operations were selected based on their frequency of use in typical database operations, considering their impact on throughput and latency measurements. Ten

benchmarks were completed, each increasing the 'pg_bench' utility's 'number of clients' parameter in iterations. This parameter affects system accesses and concurrent database connections, allowing the observation of performance under varied load levels. The results of each benchmark, based on three runs, recorded average throughput and latency for all three multi-tenant scenarios in the case studies, presented in Figure 8.2 and Figure 8.3 respectively. In the graph legend, the following acronyms are employed: 'sd' for the separate database scenario, 'ss' for separate schema, and 'st' for separate table.

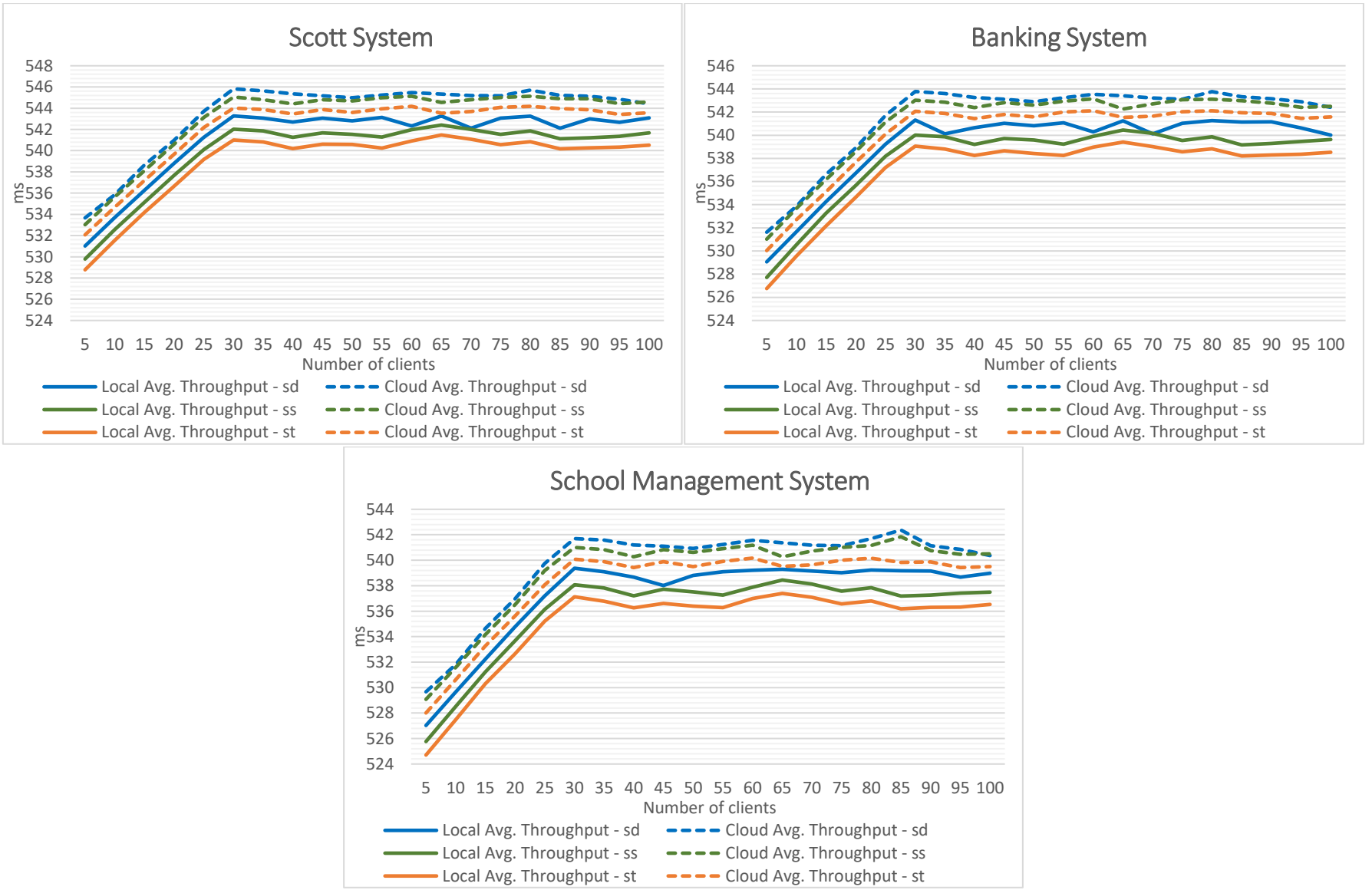


Figure 8.2: Benchmark Baseline – Average Throughput

Figure 8.3: Benchmark Baseline – Average Latency

A noteworthy observation across all three case studies is that the lowest latency and maximum throughput are consistently achieved in the separate database scenario. The reason for this is the presence of a single tenant schema, which eliminates the need for schema identification as in the case of the separate schema scenario. Additionally, the absence of ‘tenant ID’ filtering per query in the separate database scenario yields superior performance compared to the same table scenario.

Three databases (one dedicated to each CASE study) were set-up during the recording of the above-mentioned values. Naturally, an increase in the number of servers hosted on a single server may impact performance, potentially introducing overhead. Consequently, it is anticipated that as more databases are introduced, the 'separate schema' and 'same table' approaches will begin to exhibit superior performance characteristics.

From Figure 8.3, it is evident that the latency increases as the number of clients increases in all three multi-tenant scenarios (separate database, separate schema, and same table) across all three systems. This can be attributed to some bottlenecks which might be created when multiple clients access the same database resources simultaneously, leading to longer wait times. On the contrary, Figure 8.2 clearly shows that as the number of clients accessing the database augments, the throughput also increases. This is because a larger number of requests can be processed simultaneously in parallel, resulting in improved performance.

8.3.2 Performance Benchmark with security

The benchmark harness was then used to establish the throughput and latency of the same workload when executed against the same database systems but incorporating security features i.e., those generated by the proposed tool. Several observations can be elicited, based on the results obtained from this benchmark exercise.

Comparing the performance of systems developed using the proposed tool (Appendix F.1 and F.2) and the baseline systems (Figures 8.2 and 8.3), minimal disruptions were observed. The increase in latency was negligible, with a maximum difference of only 3.066ms in the 'Scott' system with 100 clients. Similarly, the throughput showed only slight variance, with a maximum difference of 2.195 requests per second in the 'School Management' system with 100 clients. These minor differences in performance highlight the efficiency and effectiveness of the proposed tool, even with

the inclusion of security features. These results indicate that the implementation of security did not significantly impact system performance, making the tool a viable option for software houses.

Secondly, the prior figures demonstrate that the system with the greatest number of tables, processes, and roles, namely the School Management System, exhibits the highest latency and lowest throughput when executing the workload across all three multi-tenant scenarios. The increase in the number of database objects, results in the processing of more data, leading to longer operation time for result retrieval. Additionally, this increase may cause contention for system resources, which can lead to delays and reduced throughput as more objects compete for the same resources, such as memory and CPU. Finally, an increase in the number of roles accessing the database can impact performance due to the higher number of security checks, which may cause increased latency and decreased throughput.

Finally, upon comparing the results of workload execution for each of the three multi-tenant approaches (separate database, separate schema, and same table), it is evident that in all three case studies, the separate database approach consistently exhibits lower latency compared to the other two scenarios. This can be attributed to the fact that this scenario necessitates fewer security measures in comparison to the others, as it entails isolating each tenant's data within its own separate database system. In contrast, the separate schema and same table scenario necessitate the implementation of additional security features, such as policies and the inclusion of tenant ID columns, to mitigate the risk of data leakage. Consequently, these scenarios exhibit a slightly longer execution time compared to the separate database approach.

8.4 Security Evaluation

This section evaluates the CASE tool output from a security perspective with the primary objective of providing a secure multi-tenant database code on a per-tenant basis. The results deduced demonstrate whether the proposed tool’s security controls are implemented correctly, operating as intended and providing the desired outcomes with respect to meeting the pre-established security requirements.

The database security evaluation proposed and validated by Vieira et al. [193] was adopted. This framework employs a numerical metric to evaluate various database aspects, ultimately leading to a final score which indicates the overall security level of the database being considered. Subsequently, a series of SQL scripts were devised to thoroughly evaluate and scrutinise the comprehensive security measures implemented in the proposed system. Conclusively 'AppDetectivePRO' [194], an application designed to identify potential security vulnerabilities and misconfigurations in database systems was used. The aforementioned application evaluated the SQL code generated by the proposed tool and ascertained that it meets high security standards.

8.4.1 Security Evaluation Framework

Vieira et al. [193], proposed a security classification system for DBMS, categorising them into classes based on increasing levels of security. Each class has specific requirements and assigns weightings indicating their importance. The system's security score is calculated based on the fulfilment of these requirements, ranging from 0 to 100. Given the tool's focus on automated code generation, elements related to data communication and transfer are considered extraneous. Consequently, the highest attainable score is 44 rather than 100. This can be noted, in Table 8.2 whereby the evaluation criteria, against which the proposed tool was appraised are identified.

Table 8.3: Security evaluation framework (Adapted from [193])

Security mechanism	Requirements	Requirement Weighting (%)
1. User authentication	1.1 Usernames and passwords must be present	10
	1.2 Besides the DBA, no other users have access to the password storage	6
2. User privileges/ Access Rights	2.1 Only DBAs can assign privileges	6
	2.2 At least SELECT, INSERT, UPDATE and DELETE privileges must be present	6
	2.3 Proper consideration is given to both horizontal and vertical partitioning	4
	2.4 No other users besides DBA have admin privileges	4
3. Encryption	3.1 The database must have mechanisms for encrypting data particularly passwords	4
4. Auditing	4.1 The system must provide auditing capabilities to enable DBAs to trace who has accessed each database object	4

Vieira et al. [193], also indicated a series of tests to ascertain the fulfilment or otherwise of each requirement. Provided that the test execution is feasible, then fulfilment of the associated requirement is affirmed. To maintain objectivity, the proposed tool was evaluated against each criterion listed in Table 8.2 and subsequently awarded either a 0 or full marks depending on whether the feature was present or absent. The percentage associated with each requirement was tabulated following a comprehensive evaluation of each feature discussed hereunder. In cases where a requirement is fulfilled in multiple ways, each possible implementation was evaluated, and marks are only awarded if all implementations successfully satisfy the tests.

8.4.1.1 User Authentication

Vieira et al. [193], proposed tests to verify user authentication requirements. The test conducted to address Requirement 1.1 involves considering all users capable of establishing system connections and ensuring that each user has an authentication procedure with distinct usernames and passwords. In the proposed CASE tool, unique credentials are assigned to every user, with passwords programmed to expire after a specified period.

Requirement 1.2 can then be verified by logging in as all potential non-tenant DBA users and ensuring that access to the table where usernames and passwords are stored is strictly prohibited. In the tool being proposed, access to the table containing all usernames and encrypted passwords is granted exclusively to the tenant DBA. This restriction is enforced by revoking access to the schema containing security sensitive information such as passwords, from all users (including PUBLIC) except for the tenant DBA. The test proposed by Vieira et al. [193], was executed, and subsequent login attempts with all non-tenant DBA user accounts revealed consistent denial of access.

8.4.1.2 User Privileges/Access Rights

Chapter 5 introduced the role-based access control mode adopted in the system. Consequently, the requirements related to user privileges and access rights were evaluated in the context of this mode. As delineated in the prior-mentioned chapter, the proposed tool automatically generates a CRUD matrix based on the inputted system diagrams. These diagrams serve as a reference for identifying the necessary privileges on the database objects. Vieira et al. [193], propose a test to validate Requirement 2.1 to ensure that only the DBA has superuser privileges for granting or revoking roles.

Apart from satisfying this condition, the tool further enhances security by storing the matrix and user privileges in the security schema, reducing the risk of unauthorised access and maintaining overall system security.

As outlined in Section 5.4.3, the CRUD matrix and other generated matrices include several checks to ensure that all functions and procedures are reachable and that relational objects can be inserted into, updated, deleted, and selected from. This approach effectively addresses accessibility, which is one of the most important security aspects whilst successfully fulfilling Requirement 2.2. Apart from granting access to entire database objects, DBAs can enforce fine-grained access restrictions for horizontal and vertical partitioning (Requirement 2.3). This is accomplished through row-level and column-level security mechanisms, limiting user access to specific data subsets and ensuring authorised information is accessed. Row-level security is implemented using RLS policies, which apply supplementary filters to tables based on the currently logged-in user. These filters are applied before returning query results, narrowing down data based on the user's role and actions.

The utilisation of RLS policies encapsulates multiple advantages. Apart from augmenting the overall security of the system, RLS policies are directly implemented on the base tables, eliminating the need to construct any additional database objects or structures. Moreover, RLS policies discretely suppress any row which cannot be viewed or modified by the user, thus ensuring that the user is unaware that such a row exists. Furthermore, RLS access coding is centralised in a single location.

Another mode of implementing row-level security in PostgreSQL is by means of views. Although such structures are ideal for implementing column-level security (evaluated subsequently), RLS policies prevail over views in this case. Views are created on the base table with a WHERE clause for filtering and permissions granted to the associated role. However, views may encounter the 'view-update' problem, limiting their ability to perform updates or insert data. The INSTEAD OF trigger can be used to address this problem, but it adds complexity and processing time. For column-level security, the CASE tool uses views or column permissions in PostgreSQL. Views provide logical data independence and conceal sensitive information.

An alternative option to views, is that of granting access to particular columns directly on the base tables by means of column-level permissions. Nonetheless, this

option has some drawbacks when compared to views. Handling rights at a more detailed level can be a challenge and performance might deteriorate due to checks being conducted at both table and column levels. Moreover, it should be ensured that access to the entire table is revoked prior to granting access to specific columns, otherwise the whole exercise would be futile.

The proposed tool introduces a deviation from the traditional approach outlined in Table 8.2 in the ‘separate database’ approach, by granting certain administrative capabilities to data owners for the objects they own, rather than limiting administration rights to DBAs only. This empowerment of data owners allows them to independently perform tasks like adding or modifying columns, enhancing efficiency and reducing dependency on DBAs. It also enhances database management flexibility, empowering data owners to quickly adapt to changing requirements, thereby saving time, reducing administrative burden, and increasing system agility.

8.4.1.3 Encryption

The test proposed by Vieira et al. [193], for validating encryption (Requirement 3.1) involves retrieving stored password values and comparing them to original inputted values, ensuring their differentiation. In the proposed tool, all passwords are encrypted by default, as evidenced by the generated SQL script. When the encryption type is not specified by the tenants, PostgreSQL, ‘MD5’ encryption [195], is adopted by default. To validate this, as indicated by the suggested test, a new tenant user named ‘Testtenant’ was created with password ‘pswd123’. After accessing the users’ table in the data dictionary, particularly the 'pg_authid' table, it was confirmed that the password linked to the 'Testtenant' account was distinct from the original one.

8.4.1.4 Auditing

Conclusively, to assess the fulfilment of the requirement pertaining to the 'auditing' section, Vieira et al. [193], propose the execution of a randomised set of operations and ensuring the comprehensive recording of all operation details. The proposed tool incorporates an auditing process for each developed system (discussed in Chapter 6), designed for traceability and ownership purposes. Following the test suggested in [193], multiple logins and the sequential execution of random functions were per-

formed in each potential multi-tenant scenario. Upon examining the audit table accessed by the tenant DBA, it was observed that all operations, including timestamps, usernames, executed queries, and resulting data changes, were properly recorded.

In Table 8.3, a concise overview of the conducted tests is provided, accompanied by an assessment of their outcomes based on the assigned weights for each requirement. The weightage determines whether a test is recorded as a pass, with full marks, or a failure, denoted by a score of 0.

Table 8.4: Tests conducted and outcome of proposed tool security evaluation.

Security mechanism	Tests to ensure requirements are satisfied	Test outcome	Assigned weighting
1. User authentication	1.1 Considering all users present in the system and verifying that an authentication procedure with distinct credentials exists for each	*Pass*	10
	1.2 Retrieving stored password values and ensuring they are different from the original ones	*Pass*	4
	1.3 Logging in as all potential non-tenant DBA users and confirming that it is not possible to access the table where the usernames and passwords are stored	*Pass*	6
2. User privileges/ Access Rights	2.1 Examining the privileges assigned to users and roles in the database and ensuring that only DBAs are granted privileges related to access rights	*Pass*	6
	2.2 Developing checks to ensure that all relational objects can be inserted to, updated, deleted, or selected from.	*Pass*	6
	2.3 Considering the implementation of horizontal and vertical partitioning	*Pass*	4
	2.4 Examining privileges associated with all users and ensuring that only DBA have administrative rights	*Fail*	0
3. Encryption	3.1 Storing encrypted data and selecting the stored data to ensure that the resulting output is unintelligible	*Pass*	4
4. Auditing	4.1 Performing a random set of operations and ensuring that all required details are recorded	*Pass*	4

It can be noted that the proposed tool obtained a rating of 40 out of 44 which is commendable. The four (4) marks were not awarded for criteria stating that 'no

other users besides DBA have admin privileges'. As explained previously, this approach was adopted not to compromise security but to enhance flexibility and efficiency. In this context, we are specifically referring to the tenant DBA. The cloud DBA does not possess any administrative privileges over the tenant's data but only over the cloud infrastructure and underlying database management system, ensuring the separation of concerns and maintaining data isolation between tenants. As a result, it can be inferred that the suggested tool has effectively outperformed the security evaluation.

8.5 Evaluation Scripts

In addition to the framework discussed in Section 8.4, a set of SQL scripts was developed to thoroughly assess and scrutinise the security measures implemented in the proposed system. Each script was accompanied by a wrapper function that returned boolean values, indicating whether the results of each script matched the expected outcome or otherwise. A description of each script is presented in this section, providing insight into their specific functionalities and objectives.

8.5.1 Edge cases

A dedicated set of scripts was developed to assess edge cases in the system. These incorporate cases which are at the extreme boundaries or limits of what the system is designed to handle. These situations are usually uncommon and unexpected, but still need to be considered and accounted for to ensure the robustness and reliability of the system.

1. **The user has access to the PostgreSQL instance (username and password), but no tenant ID is provided.**

In this scenario, it is crucial to ensure that the user has no visibility or access to any tenant's data and is unable to add any elements within the tenant's scope. To ensure the desired functionality, a dedicated function was developed. Pseudocode 8.1 outlines the implementation of this function specifically designed for separate schema and same table scenarios.

Pseudocode 8.1 Edge Case Evaluation Script

```
1: Input: username text, pass text, tenant_id integer
2: Output: TABLE (obj_name text, obj_type text, priv text, has_access boolean
3:
4: Algorithm:
5: If EXISTS                                                    > authenticate user
6:   SELECT *
7:   FROM pg_auth
8:  WHERE rolname = username && rolpassword = md5(pass)) then
9:                                                  > Set the current 'tenant ID' to NULL
10:  set_config(app.current_tenant_id, tenant_id, true)
11:
12: Create Temp Table (access_results ← Columns (objnm, objtype, priv, hasaccess)
13:
14: For object in
15:   SELECT table_name, table_schema, 'table/view'
16:    FROM information_schema.tables
17:   WHERE rolename = username
18: Do
19:   For each priv in ['SELECT', 'INSERT', 'UPDATE', 'DELETE']:
20:   hasaccess ← has_table_privilege(object.table_schema, object.table_name,
21:     priv, username)
22:   access_results ← Columns (objnm, objtype, priv, hasaccess)
23:   Values (object.table_name, object.object_type, priv, hasaccess)
24:                                                   > Check access to tables
25: End For
26:
27: For object in Columns (p.proname, n.nspname, 'function') Table
28: (pg_catalog.pg_proc, pg_catalog.pg_namespace)
29: Do
30:   hasaccess ← has_function_privilege(object.specific_schema,
31:   object.routine_name, object.argument_types)
32:   access_results ← Columns (objnm, objtype, priv, hasaccess)
33:   Values (object.routine_name, object.object_type, 'EXECUTE', hasaccess)
34:                                                 > Check access to functions
35: End For
36: return access_results
```

The function initiates by accepting two input parameters: username and password, and proceeds with user authentication. Once authentication is successful, the function sets the 'tenant ID' to NULL, indicating that the user does not have access to any specific tenant. Subsequently, the function creates a temporary table to store access permissions of the inputted user on all tables, views, and functions.

The function examines the temporary table to ensure no access permissions have been granted and returns a boolean value through the wrapper function, indicating the presence or absence of unauthorised access. In the case of a separate database scenario, a similar script is implemented, with the first three steps remaining the same. However, in this scenario due to the presence of multiple databases, the script checks the script retrieves all databases and verifies the absence of connect access to them.

2. **The user has access to the PostgreSQL instance (username and password) and provides a valid ‘tenant ID’, but the user is not assigned to that tenant.**

In this scenario, the user is prohibited from accessing or manipulating any elements within the tenant's scope. The functions implemented for the testing of this specific edge case, share similarities with those described in the previous test. The main distinction is that instead of using a NULL value as the ‘tenant ID’, the function accepts a specific ‘tenant ID’ as an input parameter. After verifying the existence of the ‘tenant ID’, a check is performed to ensure that the provided username and password are associated with it. The remaining steps closely resemble those implemented in the functions described in edge case 1, wherein all access permissions associated with the specific user are retrieved and stored in a temporary table. When the user provided is not assigned to the specified tenant, a check is performed on the temporary table to ensure the absence of any access permissions. This verification process was successfully conducted, confirming its functionality in such scenarios.

3. **The user provides a valid username password, and corresponding ‘tenant ID’. However, the user is assigned to the provided tenant and at least one other.**

If this situation arises, it is crucial to ensure that the user has the capability to view and access elements within the current tenant and not those belonging to other tenants. To achieve this objective, a verification procedure is implemented to compare the retrieved objects' 'tenant ID' with the inputted value. By retrieving all privileges associated with the provided user account and examining the 'tenant ID' associated with each object, a comparison is made against the function parameter. If all values align, no errors occur, indicating success. However, if a different 'tenant ID' is found among the retrieved objects, further investigation is needed to address the potential issue or discrepancy.

8.5.2 Authorised Users

1. **The user provides a valid user account (username and password) and tenant ID.**

This check differs from the last one discussed in Section 8.5.1. While the previous one targeted users associated with multiple tenants, ensuring that once they are logged into a tenant, they can only access data within that tenant, this check focuses on ensuring that authorised users logged into a tenant can only access authorised objects. In this case, the user's visibility and access must be strictly limited to the data assigned to them in accordance with the generated CRUD matrix. Distinct functions were implemented to evaluate the user's privileges on tables/views, functions, and data.

a. Tables/views

Three distinct functions were developed when checking privileges associated with tables/views to cater for each of the three possible multi-tenant approaches. The pseudocode for the function utilised in the 'separate database' implementation can be observed in Pseudocode 8.2 hereunder.

Pseudocode 8.2 Privileges associated with Tables/Views Evaluation Script

```
1: Input: systemid integer, tenantid integer, dbarole text
2: Output: boolean
3:
4: Algorithm:
5: Create Temp Table cruddata(rlname text, tblname text, tblVw text, perm text)
6:                                                    > retrieve privileges from CRUD
7: cruddata ← SELECT rolename, tablename, tableorview, perm
8:   FROM crud
8:   WHERE (sysdiag = systemid && tenid = tenantid)
9:  UNION                                          > retrieve RLS policies (partitions)
10:  SELECT rolename, tablename, 'RLS', columnname
11:   FROM policies
12:  WHERE (sysdiag = systemid && tenid = tenantid)
13:                                                  > retrieve actual privileges
14: actualprivileges ← SELECT grantee, table_name, table_type, privilege_type
15:   FROM role_table_grants
16:     WHERE grantee != dbarole
17:     UNION                                        > retrieve actual policies
18:   SELECT roles, tablename, 'RLS', columnname
19:   FROM role_table_grants
20:     WHERE grantee != dbarole
21: If Symmetric difference (cruddata, actualprivileges) IS NOT NULL then
23:  return false
24: Else return true
```

The primary distinction between the function depicted in Pseudocode 8.2 and the counterparts designed for the other two multi-tenant approaches is related to the part dealing with RLS policies. In the other two approaches, RLS policies used for data segregation are also included in the 'cruddata' table prior to the symmetric difference stage. Other steps of the procedure align with the approach employed in Pseudocode 8.2. The execution of these three functions for each case study conducted on systems generated by the proposed tool provided a successful validation of the functionality.

b. Stored procedures/functions

In contrast, when it comes to checking authorised privileges associated with functions, a single procedure is sufficient since the main difference across scenarios lies in the underlying tables. As illustrated in Pseudocode 8.3, the implementation of this check follows a series of steps. Firstly, function privileges for a specific tenant are retrieved from the CRUD matrix and stored in the 'cruddata' temporary table. Subsequently, actual privileges assigned to these stored procedures/functions for all roles are obtained from the tenant's database using the PostgreSQL information schema. A symmetric difference operation reveals no discrepancies in the systems generated by the proposed tool, confirming alignment between the expected and actual privileges.

Pseudocode 8.3 Privileges associated with Stored Procedures Evaluation Script

```
1:  Input: systemid integer, tenantid integer, dbarole text
2:  Output: boolean
3:
4:  Algorithm:
5:  Create Temp Table cruddata(rlname text, functionname text, perm text)
6:
7:  cruddata ← Columns (rolename, functionname, perm) Table (crud)
8:   Condition (sysdiag = systemid && tenid = tenantid)
9:                                                    > retrieve privileges from CRUD
10: actualprivileges ← Columns (grantee, table_name, table_type, privilege_type)
11:      Table (role_table_grants) Condition (grantee != dbarole)
12:                                                    > retrieve actual privileges
13:
14: If Symmetric difference (cruddata, actualprivileges) IS NOT NULL then
15:  return false
16: Else
17:  return true
```

c. Data

Four additional scripts were developed to thoroughly test the proper functionality of SELECT, INSERT, DELETE, and UPDATE operations and ensure the correct retrieval and accessing of data. Regression testing is employed to ensure the proper functionality and integrity of these four operations for authorised users. The procedure for checking the SELECT operation is explained in Pseudocode 8.4.

Pseudocode 8.4 SELECT Operation Evaluation Script

```
1:  Input: schemaname text, tablename text, expectedResult type[]
2:  Output: boolean
3:
4:  Algorithm:
5:  queryText ← SELECT *
6:     FROM schemaname.tablename
7:
8:  If queryText = expectedResult then
9:   return true
10: Else
11:  return false
```

For the INSERT operation testing, a 'true' or 'false' value is passed as an argument to indicate whether an INSERT operation is expected or not. The input parameters for the data to be inserted into a specific table are provided, and a SELECT statement is executed to retrieve the current data from the table. The INSERT operation is performed, followed by another SELECT statement to retrieve the data from the table after the INSERT operation. If the initial expected outcome argument is set to 'true', a difference of one row should be observed between the pre and post-INSERT data. Conversely, if the expected outcome argument is set to 'false', no difference should be observed. Similar procedures as those employed for INSERT are utilised for UPDATE and DELETE operations, with the distinction being the nature of the operations performed (i.e., UPDATE or DELETE) instead.

2. **A valid tenant DBA account (username and password) is supplied alongside a valid tenant ID.**

User accounts authenticated as tenant DBAs should possess more privileges compared to those authenticated as regular users. To ensure this distinction with our pro-

posed tool, a function has been implemented using a similar approach to the one described above. However, the scope of the check expands beyond the retrieval of table and function access. It includes retrieving additional privileges commonly associated with a tenant DBA, which are stored in the PostgreSQL 'information_schema'. These additional privileges may include the ability to create users and assign them to roles.

After retrieving all privileges associated with the provided tenant DBA account, a symmetrical difference check occurs (similar to the one explained in Point 1). However, instead of comparing the actual privileges with those specified in the CRUD matrix they are compared with a custom list of privileges specifically defined for this purpose. This list represents the standard collection of privileges typically associated with tenant DBAs in conventional database systems. The results obtained, following execution of this function with tenant DBAs assigned roles, indicate that systems generated by the proposed tool effectively assign appropriate privileges.

3. **Evaluation of software and cloud providers.**

To ensure the data ownership and security of the tenant database, software and cloud service providers should not possess any capabilities within the database itself. Their role should be limited to creating databases and schemas, if required, and accessing only schemas associated with them. The purpose of this evaluation function is precisely to verify that the aforementioned statements are being adhered to.

Following a similar approach as the preceding function in this section, this function operates by receiving inputs of a username, password, and 'tenant ID'. Upon verifying the provided credentials and associating them with those of a software or cloud server provider, a comparable methodology to the one employed for the testing privileges associated with tenant DBAs is implemented. Consequently, this function retrieves the presented account's privileges from the PostgreSQL 'information_schemas' and compares them to a predefined table that outlines the permitted objects accessible to software and cloud providers. If any additional grants are detected beyond what is specified in the table, a warning is raised to indicate a potential issue. Conversely, if the actual privileges align with the specifications outlined in the table, a successful implementation is evidenced. Execution of this function on systems generated by the proposed tool yielded the anticipated outcome.

8.5.3 PUBLIC Role

Given that PostgreSQL automatically generates a PUBLIC role for each database created an additional verification process was conducted on all databases generated by the proposed tool. This check aimed to verify that the PUBLIC role did not possess any access or privileges on any of the database objects. The pseudocode implementation of this check is presented in Pseudocode 8.5 below.

Pseudocode 8.5 PUBLIC Role Evaluation Script

```
1:  Input: void
2:  Output: TABLE (typ text, objname name, privtype text)
3:
4:  Algorithm:
5:  resultSet ← SELECT 'database', databasename, privilegetype
6:      FROM pg_database
7:    WHERE rolname = PUBLIC
8:                                              >extract database privileges
9:  resultSet ← SELECT 'schema', schemaname, privilegetype
10:     FROM pg_namespace
11:   WHERE rolname = PUBLIC
12:                                              >extract schema privileges
13: resultSet ← SELECT 'function', functionname, privilegetype
14:     FROM role_routine_grants
15:   WHERE grantee = PUBLIC && functionschema != pg_catalog
16:                                              >extract function privileges
17: resultSet ← SELECT 'table', tablename, privilegetype
18:     FROM role_table_grants
19:     WHERE grantee = PUBLIC && tableschema != pg_catalog
20:                                              >extract table privileges
21: return resultSet
```

Due to the absence of a single PostgreSQL table containing all possible privileges, each object in the database had to be individually checked for privileges pertaining to the PUBLIC role. As can be noted from Pseudocode 8.5, the objects considered included databases, schemas, functions, tables, and columns. The output obtained in all case studies developed via the proposed tool consistently showed positive results, confirming that no privileges are associated with the public role.

8.6 AppDetectivePRO

Following the proposed tool's evaluation outlined in the preceding section, a software program capable of scrutinising databases for potential security vulnerabilities was used [194]. The AppDetectivePRO application, developed by AppSec [196], is a cross-platform database security solution which offers various security tests related to access controls, privilege escalation vulnerabilities, and password strength assessments, providing reports on identified security issues. Therefore, this application is valuable for assessing the security of the SQL code generated by the proposed tool.

As already explained, in the 'Performance Evaluation' section, the SQL code generated for each case study was executed both locally and on the cloud. 'AppDetectivePRO' was then used to generate comprehensive security reports, highlighting vulnerabilities and listing user-rights privileges in the database.

8.6.1 Potential Security Vulnerabilities Report

Figure 8.4 displays the security vulnerability report obtained when the local 'Banking System' generated database was presented to 'AppDetectivePRO'. The report is divided into three sections: 'Findings', which presents any security vulnerabilities discovered in the presented database; 'Non-Findings', which identifies security vulnerabilities that were not discovered in the database; and 'Facts' which highlights important information discovered while the database was assessed. Each point is accompanied by an indicator showing the potential damage incurred by the database if such a vulnerability exists, classified as low, medium, or high.

As can be observed from Figure 8.4, the generated database demonstrated a high level of security when tested for vulnerabilities. This delay was a result of the introduction of new PostgreSQL patches during the program's testing phase since the system was developed and tested on PostgreSQL version 15.

No vulnerabilities related to SQL injections or bypass vulnerabilities were observed. This is primarily attributable to the fact that, as explained in Section 5.9, specific measures were deployed to prevent such threats from being present. Additional indicators not visible in the above screenshot suggested that there were no instances of denial of service, password expiration, authentication bypass vulnerabilities, or unauthorised user access.

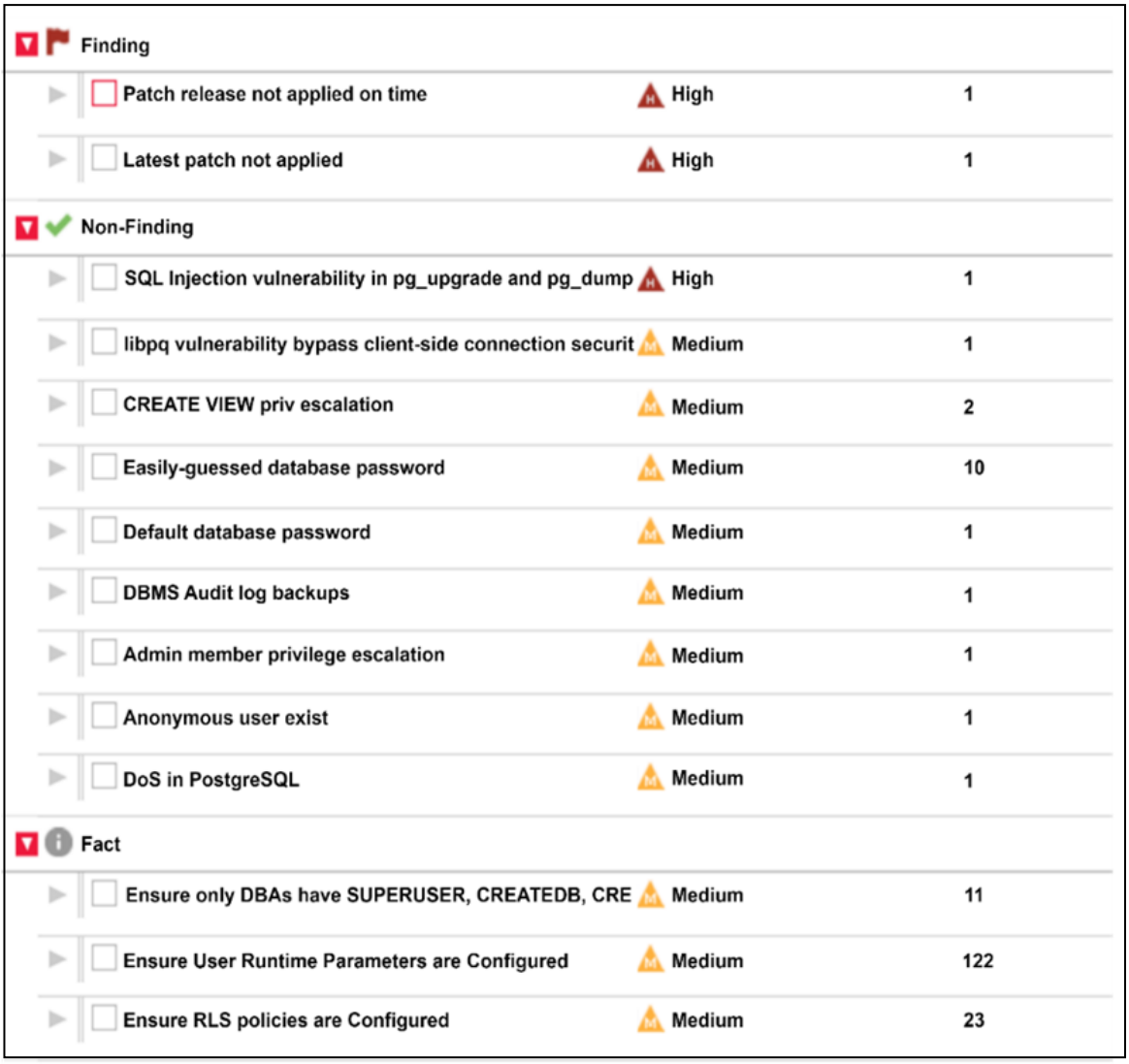


Figure 8.4: Security vulnerability report

8.6.2 User Rights Privilege Report

The 'AppDetectivePRO' tool provides user rights reports listing authorised tasks and privileges for specific database users. These reports aid in assessing security concerns, detecting unauthorised access to sensitive data, preventing data breaches, and identifying users with excessive privileges. This report can be read from either an object or a user perspective. When considered from the user perspective, the report indicates four primary columns that identify the privilege name and type, the grant path, and the grantee. When viewed from an object perspective the report displays all privileges

associated with that object. Consequently, columns for granted to, granted type, state, privilege, and object are included in this report. All entries in the 'granted by' column of this report are equal to 'DBA,' signifying that the Database Administrator holds the authority to grant every privilege. Furthermore, several privileges can be noted ranging from SELECT, INSERT, UPDATE, and DELETE for tables to EXECUTE privileges for functions.

The ‘AppDetectivePRO’ user rights report was visually compared to the proposed tool's CRUD matrix. This comparison ensured that the generated code aligns with the application's security requirements. Through confirmation and verification, it was confirmed that the privileges in the generated report precisely matched the specifications without any additional or excessive privileges. The report also highlighted any instances of unauthorised access to sensitive data, ensuring that the system developed using the proposed tool had only necessary privileges granted.

The user rights privilege report validates data isolation and ensures no data leakage occurs in all three multi-tenant scenarios. In the separate database scenario, each tenant has exclusive access to their own database and objects. In the separate schema scenario, tenants are granted usage privilege on a single schema, with access to other schemas revoked. In the same table scenario, data separation is ensured through RLS policies per user on each relation. Confirmation of this, requires examining the generated policies to ensure appropriateness. The report was reviewed to confirm data isolation in all cases. In the separate database scenario, privileges were checked to ensure no cross-tenant associations. In the separate schema scenario, privileges were validated to ensure access is limited to the assigned schema whilst in the same table scenario, RLS policies were examined to confirm data separation.

While the report generated offers various advantages, it does not provide confirmation for privileges implemented via policies. Nonetheless, such privileges can be cross-checked by referring to the detailed report discussed in the previous section. If any RLS policies are implemented in the database, they are listed under the ‘Findings’ section. Therefore, by looking at the prior report and checking that they exist, it can be ensured that even such policies are implemented as required by the application.

8.7 Summary

This chapter presents an assessment of the proposed CASE tool in alignment with the objectives outlined in Chapter 1. The evaluation covered system coverage, performance, and security aspects. A methodical approach combining qualitative and quantitative techniques was adopted, and the obtained findings were presented. These findings, derived from a series of frameworks, benchmarks, and procedures, offer valuable insights into the capabilities and limitations of the tool.

9 Conclusion

The main aim set out for this dissertation, namely the design and development of an automated CASE tool which incorporates multi-tenant cloud-based security requirements from the initial stages of database design, was accomplished. This chapter sums up the work carried out in previous chapters and highlights the primary contributions that have emerged from this research. Possible future research areas are also presented in this chapter.

9.1 Summary of Chapters

Amid economic uncertainty and growing competition, cost-saving strategies like multi-tenancy have gained increasing prominence within organisations. This formulated the context in Chapter 1 by highlighting the need for various organisations to opt for cost-effective multi-tenant applications. Nonetheless, the introductory chapter also singles out security concerns as the foremost factor hindering the widespread use of multi-tenant applications. According to various leading scholars, such concerns can only be addressed through the holistic consideration of security requirements from the onset of the database design.

Prior to the CASE tool's development, several literature works related to the dissertation's area of interest were reviewed and presented in Chapter 2. This chapter initiated with an overview of cloud computing, focusing particularly on cloud database systems and their architectural aspects. Subsequently, the concept of multi-tenancy was introduced, alongside its implementation, which primarily involves three principal approaches: separate databases, separate schemas, and shared tables. The chapter proceeded to delineate prominent types of access control systems and related security models, while also addressing security considerations inherent to the research domain. It concluded by providing an overview of the significance of CASE tool development and the existent work done by other researchers in the field.

Despite the attractiveness of multi-tenancy in cloud-based database systems, literature indicates minimal adoption due to security concerns. Moreover, a limited number of literature works have examined CASE tools, despite their advantages over prevailing manual approaches that are susceptible to human error. This dissertation

sought to address this gap by investigating and proposing a CASE tool capable of automating security profile development in multi-tenant cloud systems, with an emphasis on security considerations from the outset of database design.

Chapter 2 laid the groundwork for the formulation of a comprehensive set of security requirements for multi-tenant cloud-based systems presented in Chapter 3. These requirements serve as the foundation for the framework developed in this dissertation, considering both literature works and the practices of prominent CDBMS vendors to ensure a comprehensive perspective. Furthermore, upon identifying the requirements, a comprehensive analysis was conducted to determine the key techniques that the proposed framework would employ to fulfil these requirements. This analysis involved evaluating various alternatives and selecting and justifying the most appropriate approach. The primary objective of this analysis was not only to identify suitable solutions for the established requirements but also to provide a comprehensive rationale for the chosen course of action.

Chapter 4 presents the design stages of the proposed tool and outlines the design process employed in its development. It specifically focuses on the significance of system design diagrams — ERM, ELH, and DFD — as foundational elements ensuring security considerations from the initial stages and serving as the basis for constructing unique security assertions for each tenant. The chapter also delves into the design aspects concerning other key users within a multi-tenant scenario, emphasising the holistic consideration necessary for a system's effective operation. Additionally, it addresses the design of the process for converting user-inputted information, in this case, the initial system design diagrams, into relational objects for improved tool utilisation. Furthermore, the chapter introduces and develops checks related to these initial stages (design diagrams) capable of identifying potential security vulnerabilities.

Chapter 5 examines the security access control framework for the proposed tool, building upon insights extracted from previous chapters. It provides a detailed discussion on the implementation of Role-Based Access Control (RBAC), chosen as the access mode for this dissertation. Emphasis is placed on automating the CRUD matrix generation, a pivotal component for defining access permissions, and further explores its enhancement through role integration, enabling a more refined and gran-

ular approach to security management. Additionally, it considers vertical and horizontal partitioning, ensuring the establishment of a robust, fine-grained access control system to effectively safeguard critical resources. Moreover, the chapter discusses tenant specifications and details how the implementation of the proposed tool has been adapted to mitigate threats outlined in Chapter 2, as well as their implications for the access control framework.

Chapter 6 covers the practical implementation concepts of the proposed tool, building upon the design principles established in Chapter 4 and the access control considerations outlined in Chapter 5. The primary focus centres on the SQL scripts developed for the automated generation of secure SQL code tailored to each tenant. These scripts encompass a diverse array of essential functions, including the creation of databases and schemas, configuration of tables and functions, enforcement of RLS policies, generation of views, and the setting up of backup procedures. These scripts collectively play a pivotal role in the overall functionality of the proposed tool, with a specific emphasis on ensuring data security and optimising data access management. Additionally, the chapter explores the post-specification checks conducted to ensure comprehensive security coverage and provides insights into the complexities associated with each of the three multi-tenant approaches.

In Chapter 7, a CASE study was conducted to illustrate the step-by-step operation of the proposed tool. The CASE study centred around a banking system with a subsequent two additional CASE studies, one featuring a simpler scenario (Scott) and the other involving a more intricate system (School Management System). The latter case studies are documented in the Appendix section. The methodology outlined in Figure 4.1 (page 59) was systematically applied to each study, providing a comprehensive examination of the entire process involved in designing and establishing a secure multi-tenant database system.

In Chapter 8, the effectiveness of the proposed tool is demonstrated through rigorous benchmarking and evaluation procedures. The system is evaluated from the perspectives of system coverage, performance and security to ensure the fulfilment of the specified objectives. In the evaluation of system coverage, a qualitative analysis reveals that the selected system diagrams comprehensively represent the system from functional, structural and time-based perspectives. Despite their simplicity, these

diagrams prove highly effective, imposing minimal additional burden on software providers.

A quantitative assessment is conducted, and benchmarks are established to assess the tool's performance efficiency under various workloads and scenarios, both locally and on the cloud. The results indicate that, across all three multi-tenant approaches, the tool's impact on performance is negligible, with minimal disruptions observed. Latency increases only marginally, with a maximum difference of just 3.066ms in the 'Scott' system with 100 clients. Similarly, throughput displays only slight variance, with a maximum difference of 2.195 requests per second in the 'School Management' system with 100 clients. These minor performance differences underscore the efficiency and effectiveness of the proposed tool, even with the integration of security features. These results affirm that the implementation of security has no significant adverse effect on system performance, making the tool a practical choice for software houses.

In terms of security evaluation, a multi-faceted approach was undertaken. Initially, the proposed tool was evaluated against a security evaluation framework and achieved a score of 40 out of 44. Subsequently, several evaluation scripts systematically assessed the security measures integrated into the proposed system. These scripts covered diverse scenarios, including edge cases and actions by both authorised and unauthorised users, all yielding positive results. Finally, 'AppDetectivePro' was utilised to assess database security vulnerabilities. When applied to three distinct case studies developed with our proposed tool, the generated reports demonstrated highly favourable outcomes.

9.2 Objective Alignment and Contributions

Security concerns present a substantial obstacle to the widespread adoption of multi-tenant CDBMSs. Nevertheless, there exists a noticeable research gap in the development of systems that incorporate security from the earliest stages of design. The CASE tool developed in this dissertation presented an approach which addresses this lacuna, in line with the objectives articulated in Chapter 1. This contribution extends

to the technological research community, with specific system design details and empirical test results published in peer-reviewed publications. The key contributions are outlined below.

1. **A cohesive set of security requirements, encompassing base, functional and non-functional aspects, was formulated.**

While literature acknowledges the significance of early requirement establishment in system development, there is a scarcity of works dedicated to this topic. Notably, a gap exists in the literature pertaining to the creation of a comprehensive set of requirements, specifically those designed for secure multi-tenant cloud systems. Consequently, this dissertation compiled a cohesive set of requirements, derived from an analysis of two principal sources: academic literature and the security features prevalent in widely-adopted cloud-based software.

2. **Security considerations were incorporated into the design process through the utilisation of three diagrams: ERM, ELH, and DFD.**

ERMs, ELHs, and DFDs were selected as the core design diagrams for embedding security assertions right from the inception of the design process. This choice stemmed from a thorough evaluation of various alternatives, with these diagrams being preferred for their capacity to provide a comprehensive overview of the application's architecture while maintaining an optimal balance between simplicity and complexity. In contrast to comparable literature that often relies on a single diagram [146] Formatting..., the proposed approach is favoured because it enables a more holistic representation of systems by encompassing functional, structural, and temporal aspects

3. **A comprehensive set of EBNF rules, suitable as the basis for representing ERM, ELH, and DFD, were formulated, and corresponding data dictionaries were created.**

Given EBNF's widely accepted standard for describing language structure syntax concisely and flexibly, these rules serve a valuable role in standardisation efforts. To the best of our knowledge, no collection of such rules in EBNF or any equivalent notation for these diagrams exists in the literature. Furthermore, an automated process was developed to convert ERM, ELH, and DFD diagrams into dedicated relational data dictionaries, accompanied by the establishment of a suite of security checks for

the individual and collective assessment of these three system diagrams. To the best of our knowledge, there is a notable absence in the current literature of a comprehensive checklist that incorporates both security access considerations and mandatory requirements for each diagram, suitable for both individual and collective utilisation of these diagrams.

4. **Devising an automated process for extracting security-related matrices, including CRUD and roles-processes, from the presented system diagrams.**

Throughout this dissertation, security-related matrices were automatically extracted from the presented diagrams. This automated process not only eliminates the need for manual intervention but also proves crucial when implementing role-based access control. Additionally, horizontal and vertical partitioning strategies were employed as a means to fine-tune data granularity and optimise data management within the database system.

5. **Development of a CASE tool capable of generating unique security profiles for each tenant on a multi-tenant cloud database system**

The primary achievement and contribution of this dissertation are encapsulated in the development of the aforementioned automated tool. This tool yields a final output in the form of a comprehensive set of SQL code, encompassing essential database components such as schemas, tables, and functions. Notably, the tool's greatest advantage lies in its capacity to tailor roles, privileges, access permissions, and security measures for each tenant. Moreover, its value is further underscored by its integration of mechanisms designed to mitigate the common threats identified in the literature concerning cloud-centric multi-tenant database systems.

The final proposed tool was validated through the utilisation of three case studies: the 'Scott' system, the 'Banking' system scenario, and the 'School Management' system, serving as test scenarios to confirm the accuracy and functionality of the SQL scripts generated by the proposed tool. A systematic approach, integrating both qualitative and quantitative evaluation techniques, was applied to conduct a comprehensive assessment of the CASE tool's effectiveness and efficiency. This assessment took various factors into account to ensure that the tool provides a holistic understanding of the system, including its security aspects, all while minimising operational disruptions to maintain seamless regular operations.

9.3 Related Work

As highlighted throughout this study, security stands as one of the most critical concerns influencing individuals considering multi-tenant cloud-based systems. While existing research, such as [14] and [145], has made significant strides in addressing various aspects of security across different focuses, it tends to broadly address security in local or cloud databases. This broad focus overlooks the intricacies specific to security within a multi-tenant environment, consequently leaving notable gaps.

Moreover, while security is recognised as an essential element of any information system, few literature works consider security from the early stages of database design. Addressing security as an afterthought rather than a fundamental component during the initial design phase poses significant challenges, as integrating security measures onto an existing database architecture may prove cumbersome. This absence underscores the need for further research into the integration of security measures into CASE tools from the outset of database design.

In addition to the issue of addressing security post-design, current research works, such as [140] and [143], frequently concentrate on tackling isolated security facets like access control mechanisms or encryption, rather than embracing a holistic approach. While these contributions are valuable, they underscore the necessity for a unified framework particularly in the area of multi-tenant cloud-based database systems.

In conclusion, our study's emphasis on integrating security considerations from the outset of multi-tenant database design, while addressing various threats and considerations comprehensively, represents a novel and significant contribution to the field.

9.4 Future Work

Given that multi-tenancy and cloud computing are ever-evolving, characterised by daily emerging solutions and threats that directly affect end-users, it is imperative to consistently review and enhance existing work to maintain its relevance and practicality in the current market.

Furthermore, there are other untapped areas related to the proposed tool, which can be leveraged for further enhancement, as outlined in this section.

- Expanding the scope to encompass data security, not only related to data at rest but also data security during transmission, particularly during transit between systems or over a network.
- Considering the automated generation of security profiles for data stored in non-conventional database systems like object-oriented databases and NoSQL solutions.
- Augmenting customisation capabilities to accommodate a wider range of tenant-specific requirements, including those with a specific focus on addressing threats tailored to the unique characteristics of each scenario.
- Adopting alternative access control frameworks, including MAC or attribute-based models, and potentially investigating hybrid systems.
- Incorporating advanced formalisation and verification processes into the proposed algorithms and implementations, complementing the comprehensive evaluation conducted in Chapter 8.

In conclusion, this dissertation has explored the intricacies of multi-tenant cloud database systems, shedding light on critical issues and proposing innovative solutions. It represents the culmination of years of rigorous research and experimentation. The journey was marked by numerous challenges, from discerning the most fitting security requirements to crafting the CASE tool. Nevertheless, the sense of fulfilment derived from overcoming these obstacles surpasses the difficulties faced. This dissertation serves as a personal milestone, providing the knowledge and insights to envision the future of multi-tenant database systems with confidence and a positive outlook.

References

[1] H. Pallathadka *et al*, "An investigation of various applications and related challenges in cloud computing," *Materials Today: Proceedings*, vol. 51, pp. 2245-2248, 2022.

[2] R. Beri and V. Behal, "Cloud computing: A survey on cloud Computing," *International Journal of Computer Applications*, vol. 111, *(16)*, 2015.

[3] L. Wood, "Global cloud security market trends, opportunities, and forecasts, 2018-2023 & 2024-2028F," Research and Markets, Dublin, May 15. 2023.

[4] Lambert Consulting. *"Statistics and trends in Cloud Computing 2023 - 2024."* https://www.lambertconsulting.ch/en/statistiques-et-tendances-du-cloud-computing-2023-2024/. (accessed 02.08. 2023).

[5] K. Jamsa, *Cloud Computing*. (Second Edition ed.) Jones & Bartlett Learning, 2022.

[6] T. H. Shareef, K. H. Sharif and B. N. Rashid, "A Survey of Comparison Different Cloud Database Performance: SQL and NoSQL," *Passer Journal of Basic and Applied Sciences*, vol. 4, *(1)*, pp. 45-57, 2022.

[7] V. K. Kashinath G, "Cloud services market research, 2031," Allied Market Research, 2023.

[8] F. Chong, G. Carraro and R. Wolter, "Multi-tenant data architecture," *MSDN Library, Microsoft Corporation*, pp. 14-30, 2006.

[9] V. Narasayya and S. Chaudhuri, "Multi-tenant cloud data services: State-of-the-art, challenges and opportunities," in *Proceedings of the 2022 International Conference on Management of Data*.

[10] A. Mallick and R. K. Rathore, "Survey On Database Design For SAAS Cloud Application," *International Journal of Computer Engineering & Technology (IJCET)*, vol. 6, *(6)*, 2015. DOI: JCET_06_06_007.

[11] Check Point Software Technologies Ltd. *"Cloud Security Threats Remain Rampant: Check Point Survey Reveals Heightened Concerns for 76% of Organizations Amid 48% Increase in Cloud-Based Network Attacks."* ttps://www.checkpoint.com/press-releases/cloud-security-threats-remain-rampant-check-point-survey-reveals-heightened-concerns-for-76-of-organizations-amid-48-increase-in-cloud-based-network-attacks/. (accessed 01.08., 2023).

[12] M. Jangjou and M. K. Sohrabi, "A comprehensive survey on security challenges in different network layers in cloud computing," *Archives of Computational Methods in Engineering*, vol. 29, *(6)*, pp. 3587-3608, 2022.

[13] W. Stallings, *Computer Security Principles and Practice*. 2015.

[14] A. Kousalya and N. Baik, "Enhance cloud security and effectiveness using improved RSA-based RBAC with XACML technique," *International Journal of Intelligent Networks*, vol. 4, pp. 62-67, 2023.

[15] A. Rashid and A. Chaturvedi, "Cloud computing characteristics and services: a brief review" *International Journal of Computer Sciences and Engineering*, vol. 7, *(2)*, pp. 421-426, 2019.

[16] K. Jani, B. Kumar and H. Shah, "Degree of multi-tenancy and its database for cloud computing," *International Journal of Engineering Development and Research*, vol. 3, pp. 168-171, 2013.

[17] S. Lee, "Database management system as a cloud service," *International Journal of Future Generation Communication and Networking*, vol. 5, *(2)*, 2012.

[18] T. Garg, R. Kumar and J. Singh, "A way to cloud computing basic to multitenant environment," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 2, *(6)*, pp. 2394-2399, 2013.

[19] P. Mell and T. Grance, "The NIST definition of cloud computing," 2011.

[20] M. Alshinwan *et al*, "Integrated cloud computing and blockchain systems: A review," vol. 7, *(2)*, pp. 941-956, 2023.

[21] Soundarya Jayaraman. *"160+ Fascinating Cloud Computing Statistics for 2023."* https://www.g2.com/articles/cloud-computing-statistics. (accessed 16.09., 2023).

[22] J. Hassan *et al*, "The rise of cloud computing: data protection, privacy, and open research challenges—a systematic literature review (SLR)," vol. 8, 2022.

[23] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," vol. 34, *(1)*, pp. 1-11, 2011.

[24] Q. Zhang, L. Cheng and R. Boutaba,“Cloud Computing: state-of-the-art and research challenges” J Internet Serv Appl," 2010.

[25] A. Bagaeen *et al*, "Storage as a service (staas) security challenges and solutions in cloud computing environment: An evaluation review," in *2019 Sixth HCT Information Technology Trends (ITT).*

[26] S. Roy, P. K. Pattnaik and R. Mall, "A cognitive approach for evaluating the usability of Storage as a Service in Cloud Computing Environment." vol. 6, *(2)*, 2016.

[27] M. Kansal *et al*, "A Systematic Study of Services and Security Model in Cloud Computing: A Brief Overview," pp. 1-16, 2023.

[28] B. S. Rakhimov, F. B. Rakhimova and S. K. Sobirova, "Modeling database management systems in medicine," in *Journal of Physics: Conference Series*, vol. 1889, (2), 2021.[29] B. Rawat and S. Purnama, "MySQL Database Management System (DBMS) On FTP Site LAPAN Bandung," vol. 1, *(2)*, pp. 173-179, 2021.

[30] S. B. Shende and P. P. Chapke, "Cloud database management system (CDBMS)," vol. 4, *(1)*, pp. 1462, 2015.

[31] R. Chopra, *Database Management System (DBMS) A Practical Approach*. 2010.

[32] S. Riggs *et al*, *PostgreSQL 9 Administration Cookbook*. 2015.

[33] PostgreSQL Global Development Group. *"CREATE ROLE - SQL Commands."* https://www.postgresql.org/docs/10/sql-createrole.htm. (accessed 01.08., 2023).

[34] PostgreSQL Global Development Group. *"CREATE FUNCTION - SQL Commands."* https://www.postgresql.org/docs/current/sql-createfunction.html. (accessed 01.08., 2023).

[35] C. Chaubey and M. Nanda, "Cloud Database Management System Architecture," vol. 11, *(10)*, pp. 2020, 2020.

[36] S. Malhotra *et al*, "Cloud Database Management System security challenges and solutions: an analysis," vol. 4, pp. 199-207, 2016.

[37] M. Kadam *et al*, "Cloud Database Management System (CDBMS)," vol. 2, *(3)*, pp. 414-420, 2014.

[38] C. Türker, "Schema evolution in SQL-99 and commercial (object-) relational DBMS," in *FoMLaDO Workshop on Database Schema Evolution and Meta-Modeling*.

[39] PostgreSQL Global Development Group. *"GRANT - SQL Commands."* https://www.postgresql.org/docs/10/sql-grant.html. (accessed 01.08., 2023).

[40] M. Rijah, "Security analysis, threats, & challenges in database," vol. 14, *(4)*, 2021.

[41] Almutairi, A., Sarfraz, M., Basalamah, S., Aref, W. and Ghafoor, A, "A distributed access control architecture for cloud computing. ," vol. 2, *(29)*, pp. 36-44., .

[42] M. R. Anwar, R. Panjaitan and R. Supriati, "Implementation Of Database Auditing By Synchronization DBMS," vol. 1, *(2)*, pp. 197-205, 2021.

[43] B. Rawat and S. Purnama, "MySQL Database Management System (DBMS) On FTP Site LAPAN Bandung," vol. 1, *(2)*, pp. 173-179, 2021.

[44] Z. Deineko *et al*, "Features of Database Types," vol. 5, *(10)*, 2021.

[45] A. Molinaro, *SQL Cookbook: Query Solutions and Techniques for Database Developers*. 2005.

[46] A. R. Thakar *et al*, "The catalog archive server database management system," vol. 10, *(1)*, pp. 30-37, 2008.

[47] S. Manandhar, "Blocking SQL Injection in Database Stored Procedures." , Tribhuvan University, Institute of Science and Technology, 2010.

[48] P. Kraft *et al*, "Apiary: A DBMS-Backed Transactional Function-as-a-Service Framework," 2022.

[49] J. Logeshwaran, G. Ramesh and V. Aravindarajan, "A secured database monitoring method to improve data backup and recovery operations in cloud computing," vol. 2, *(1)*, pp. 37-43, 2023.

[50] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*. 20147.

[51] T. Waizenegger, "Data security in multi-tenant environments in the cloud" 2012.

[52] B. Tang, Q. Li and R. Sandhu, "A multi-tenant RBAC model for collaborative cloud services," in *2013 Eleventh Annual Conference on Privacy, Security and Trust*, .

[53] G. Duraisamy *et al*, "Analysis of access control model for data Security and Privacy on Multi-Tenant SaaS," vol. 24, *(3)*, pp. 1619-1622, 2018.

[54] S. Aldossary and W. Allen, "Data security, privacy, availability and integrity in cloud computing: issues and current solutions," vol. 7, *(4)*, 2016.

[55] B. Li and S. Kumar, "Managing Software-as-a-Service: Pricing and operations," vol. 31, *(6)*, pp. 2588-2608, 2022.

[56] B. Alam, M. N. Doja and S. Mongia, "Analysis of security issues for cloud computing," vol. 11, *(9)*, pp. 117-125, 2013.

[57] G. K. Lydall, "Quality Impact of Configuration and Customisation on Configurable Software." University of the Witwatersrand, 2018.

[58] D. Comer, *The Cloud Computing Book: The Future of Computing Explained*. 2021.

[59] M. Almorsy, J. Grundy and I. Müller, "An analysis of the cloud computing security problem" 2016.

[60] C. Bezemer and A. Zaidman, "Multi-tenant SaaS applications: Maintenance dream or nightmare?" in *Proceedings of the Joint Ercim Workshop on Software Evolution (Evol) and International Workshop on Principles of Software Evolution (Iwpse)*.

[61] Z. Ning *et al*, "Distributed and dynamic service placement in pervasive edge computing networks," *IEEE Trans.Parallel Distrib.Syst.*, vol. 32, *(6)*, pp. 1277-1292, 2020.

[62] F. Amadin and A. C. Obienu, "Multi-tenancy approach: an emerging paradigm for database consolidation," vol. 8, *(1)*, 2017.

[63] G. Karataş *et al*, "Multi-tenant architectures in the cloud: A systematic mapping study," in *2017 International Artificial Intelligence and Data Processing Symposium (IDAP)*, .

[64] A. Simitos, "Multi-tenant databases for SaaS-Security and privacy issues," pp. 4-9, 2016.

[65] D. Jacobs and S. Aulbach, "Ruminations on multi-tenant databases," 2007.

[66] A. O. Abdul *et al*, "A performance evaluation of multi-tenant data tier design patterns in a containerized environment," in *2017 International Conference on Information Society (i-Society)* .

[67] S. S. Gulati and S. Gupta, "A framework for enhancing security and performance in multi-tenant applications," vol. 5, *(2)*, pp. 233-237, 2012.

[68] L. Heng, Y. Dan and Z. Xiaohong, "Survey on multi-tenant data architecture for SaaS," vol. 9, *(6)*, pp. 198, 2012.

[69] Y. Januzaj, J. Ajdari and B. Selimi, "DBMS as a Cloud service: Advantages and Disadvantages," vol. 195, pp. 1851-1859, 2015.

[70] E. B. Fernandez, R. Monge and K. Hashizume, "Building a security reference architecture for cloud systems," vol. 21, pp. 225-249, 2016.

[71] L. M. Kaufman, "Data security in the world of cloud computing," vol. 7, *(4)*, pp. 61-64, 2009.

[72] B. R. Kandukuri and A. Rakshit, "Cloud security issues," in *2009 IEEE International Conference on Services Computing*, .

[73] M. S. Ahmad and G. R. Bamnote, "Data leakage detection and data prevention using algorithm," vol. 6, *(2)*, pp. 394-399, 2013.

[74] S. Kanade and R. Manza, "A Comprehensive Study on Multi Tenancy in SAAS Applications," vol. 181, *(44)*, pp. 25-27, 2019.

[75] W. J. Brown, V. Anderson and Q. Tan, "Multitenancy-security risks and countermeasures," in *2012 15th International Conference on Network-Based Information Systems*, .

[76] M. Ali, S. U. Khan and A. V. Vasilakos, "Security in cloud computing: Opportunities and challenges," *Inf.Sci.*, vol. 305, pp. 357-383, 2015.

[77] C. Rong, S. T. Nguyen and M. G. Jaatun, "Beyond lightning: A survey on security challenges in cloud computing," *Comput.Electr.Eng.*, vol. 39, *(1)*, pp. 47-54, 2013.

[78] K. Hashizume *et al*, "An analysis of security issues for cloud computing," vol. 4, pp. 1-13, 2013.

[79] Z. Xiao and Y. Xiao, "Security and privacy in cloud computing," vol. 15, *(2)*, pp. 843-859, 2012.

[80] B. Alouffi *et al*, "A systematic literature review on cloud computing security: threats and mitigation strategies," vol. 9, pp. 57792-57807, 2021.

[81] D. Hubbard and M. Sutton, "Top threats to cloud computing v1. 0," pp. 1-14, 2010.

[82] T. Islam, D. Manivannan and S. Zeadally, "A classification and characterization of security threats in cloud computing," vol. 7, *(1)*, pp. 268-285, 2016.

[83] M. Kazim and S. Y. Zhu, "A survey on top security threats in cloud computing," vol. 6, *(3)*, pp. 109-113, 2015.

[84] S. V. K. Kumar and S. Padmapriya, "A survey on cloud computing security threats and vulnerabilities," vol. 2, *(1)*, pp. 622-625, 2014.

[85] G. Amalarethinam and S. Rajakumari, "A survey on security challenges in cloud computing," vol. 24, *(1)*, pp. 133-141, 2019.

[86] R. Al Nafea and M. A. Almaiah, "Cyber security threats in cloud: Literature review," in *2021 International Conference on Information Technology (ICIT)*, .

[87] W. Bonasera, M. M. Chowdhury and S. Latif, "Denial of service: A growing underrated threat," in *2021 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*.

[88] C. S. Alliance, "Top threats to cloud computing v1. 0, vol. 23, 2010.

[89] G. Verma and S. Adhikari, "Cloud computing security issues: a stakeholder's perspective," vol. 1, *(6)*, pp. 329, 2020.

[90] S. K. Sood, "A combined approach to ensure data security in cloud computing," vol. 35, *(6)*, pp. 1831-1838, 2012.

[91] Luke Irwin. *"List of data breaches and cyber attacks in May 2019 – 1.39 billion records leaked."* https://www.itgovernance.co.uk/blog/list-of-data-breaches-and-cyber-attacks-in-may-2019-1-39-billion-records-leaked. (accessed 01.08.2023).

[92] S. Kulkarni and S. Urolagin, "Review of attacks on databases and database security techniques," vol. 2, *(11)*, pp. 253-263, 2012.

[93] A. Mousa, M. Karabatak and T. Mustafa, "Database security threats and challenges," in *2020 8th International Symposium on Digital Forensics and Security (ISDFS)*.

[94] G. B. Pallavi and P. Jayarekha, "Secure and efficient multi-tenant database management system for cloud computing environment," vol. 14, *(2)*, pp. 703-711, 2022.

[95] J. Qiu *et al*, "A survey on access control in the age of internet of things," vol. 7, *(6)*, pp. 4682-4696, 2020.

[96] L. Cavique and M. Cavique, "Axiomatic design applied to CRUD matrix in information systems," in *IOP Conference Series: Materials Science and Engineering*.

[97] R. El Sibai *et al*, "A survey on access control mechanisms for cloud computing," vol. 31, *(2)*, pp. e3720, 2020.

[98] A. I. Abdi *et al*, "Blockchain platforms and access control classification for IoT systems," vol. 12, *(10)*, pp. 1663, 2020.

[99] Y. Wang *et al*, "A survey on metaverse: Fundamentals, security, and privacy," 2022.

[100] N. Kashmar, M. Adda and M. Atieh, "From access control models to access control metamodels: A survey," in *Advances in Information and Communication: Proceedings of the 2019 Future of Information and Communication Conference (FICC), Volume 2*.

[101] E. Bertin *et al*, "Access control in the Internet of Things: a survey of existing approaches and open research questions," vol. 74, pp. 375-388, 2019.

[102] R. Aluvalu and L. Muddana, "A survey on access control models in cloud computing," in *Emerging ICT for Bridging the Future-Proceedings of the 49th Annual Convention of the Computer Society of India (CSI) Volume 1*.

[103] K. Soni and S. Kumar, "Comparison of RBAC and ABAC security models for private cloud," in *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*.

[104] N. Meghanathan, "Review of access control models for cloud computing," vol. 3, *(1)*, pp. 77-85, 2013.

[105] F. Cai *et al*, "Survey of access control models and technologies for cloud computing," vol. 22, pp. 6111-6122, 2019.

[106] N. W. Lo, T. C. Yang and M. H. Guo, "An attribute-role based access control mechanism for multi-tenancy cloud environment," vol. 84, pp. 2119-2134, 2015.

[107] I. Indu, P. R. Anand and V. Bhaskar, "Identity and access management in cloud environment: Mechanisms and challenges," vol. 21, *(4)*, pp. 574-588, 2018.

[108] G. Pernul, "Information systems security: Scope, state-of-the-art, and evaluation of techniques," *Int.J.Inf.Manage*, vol. 15, *(3)*, pp. 165-180, 1995.

[109] K. Almarhabi, "Arbiter: A lightweight, secured and enhanced access control mechanism for cloud computing," in *2019 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*.

[110] S. M. Hasani and N. Modiri, "Criteria specifications for the comparison and evaluation of access control models," vol. 5, *(5)*, pp. 19, 2013.

[111] N. Zhang, "Generating verified access control policies through model-checking," 2005.

[112] Y. A. Younis, K. Kifayat and M. Merabti, "An access control model for cloud computing," vol. 19, *(1)*, pp. 45-60, 2014.

[113] M. Cristiá and G. Rossi, "Automated proof of Bell–LaPadula security properties," vol. 65, *(4)*, pp. 463-478, 2021.

[114] G. D. Wurster, "Security mechanisms and policy for mandatory access control in computer systems," 2010.

[115] R. S. Danturthi, *70 Tips and Tricks for Mastering the CISSP Exam.* 2020.

[116] K. J. Biba, "Integrity Consideration for Security Computer System. Technical Report TR-3153," 1977.

[117] M. Toapanta *et al*, "Analysis of the appropriate security models to apply in a distributed architecture," in *IOP Conference Series: Materials Science and Engineering*.

[118] T. Tsegaye and S. Flowerday, "A Clark-Wilson and ANSI role-based access control model," vol. 28, *(3)*, pp. 373-395, 2020.

[119] D. F. Brewer and M. J. Nash, "The chinese wall security policy." in *IEEE Symposium on Security and Privacy*, .

[120] F. K. Parast *et al*, "Cloud computing security: A survey of service-based models," *Comput.Secur.*, vol. 114, pp. 102580, 2022.

[121] M. A. Da Silva and M. Danziger, "The importance of security requirements elicitation and how to do it," 2015.

[122] S. Lim, A. Henriksson and J. Zdravkovic, "Data-driven requirements elicitation: A systematic literature review," vol. 2, pp. 1-35, 2021.

[123] G. R. Tsochev and R. I. Trifonov, "Cloud computing security requirements: A review," in *IOP Conference Series: Materials Science and Engineering*.

[124] R. Kumar and R. Goyal, "On cloud security requirements, threats, vulnerabilities and countermeasures: A survey," vol. 33, pp. 1-48, 2019.

[125] M. Irshad, K. Petersen and S. Poulding, "A systematic literature review of software requirements reuse approaches," vol. 93, pp. 223-245, 2018.

[126] B. Kitchenham, "Procedures for performing systematic reviews," vol. 33, *(2004)*, pp. 1-26, 2004.

[127] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, .

[128] D. Badampudi, C. Wohlin and K. Petersen, "Experiences from using snowballing and database searches in systematic literature studies," in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, .

[129] N. Yousaf *et al*, "Investigation of latest CASE tools for database engineering: A systematic literature review," in *2022 International Conference on Frontiers of Information Technology (FIT)*.

[130] S. S. Patel, "Unified Modelling Language (UML) Tool for Software Architecture," vol. 28, *(11)*, pp. 1618-1627, 2022.

[131] G. Dias, "Evolvement of computer aided software engineering (CASE) Tools: A user experience," vol. *6*, *(3)*, pp. 55, 2017.

[132] D. Dalli, "Automating the Design Process of A Secure Database , University of Malta, 2010.

[133] S. Asghar *et al*, "Analytical framework for studying the effects of user participation in DSS development," in *4th International Conference on New Trends in Information Science and Service Science*.

[134] Meta Case. *"MetaEdit+ Modeler - Supports your modeling language."* https://www.metacase.com/mep/. (accessed 01.08., 2023).

[135] I. Informer Technologies. *"DB-Main 9.2."* https://software.informer.com/about.html. (accessed 02.08., 2023).

[136] H. Mouratidis and P. Giorgini, *Integrating Security and Software Engineering: Advances and Future Visions: Advances and Future Visions*. 2006.

[137] M. Singh and C. Singh, "Multi Tenancy Security in Cloud Computing," vol. 4, *(116)*, pp. 1-7, 2017.

[138] D. Ziani and R. Al-Muwayshir, "Improving privacy and security in Multitenant cloud ERP systems," vol. 8, *(5)*, 2017.

[139] P. Sun, "Security and privacy protection in cloud computing: Discussions and challenges," vol. 160, pp. 102642, 2020.

[140] X. Li *et al*, "Multi-tenancy based access control in cloud," in *2010 International Conference on Computational Intelligence and Software Engineering*, .

[141] M. Anwar and A. Imran, "Access control for multi-tenancy in cloud-based health information systems," in *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*, .

[142] P. R. Kumar, P. H. Raj and P. Jelciana, "Exploring data security issues and solutions in cloud computing," vol. 125, pp. 691-697, 2018.

[143] P. Kumar and A. Kumar Bhatt, "Enhancing multi-tenancy security in the cloud computing using hybrid ECCbased data encryption approach," vol. 14, *(18)*, pp. 3212-3222, 2020.

[144] K. A. Torkura *et al*, "Continuous auditing and threat detection in multi-cloud infrastructure," *Comput.Secur.*, vol. 102, pp. 102124, 2021.

[145] B. Mishachandar, S. Vairamuthu and M. Pavithra, "A data security and integrity framework using third-party cloud auditing," vol. 13, *(5)*, pp. 2081-2089, 2021.

[146] L. Giuri and P. Iglio, "A role-based secure database design tool," in *Proceedings 12th Annual Computer Security Applications Conference*, .

[147] J. Abramov, A. Sturm and P. Shoval, "A pattern based approach for secure database design," in *Advanced Information Systems Engineering Workshops: CAiSE 2011 International Workshops, London, UK, June 20-24, 2011. Proceedings 23* .

[148] M. Wazid *et al*, "Design of secure key management and user authentication scheme for fog computing services," *Future Generation Comput.Syst.*, vol. 91, pp. 475-492, 2019.

[149] R. Zahra, J. G. Vella and E. Cachia, "Specifications of requirements to ensure proper implementation of security policies in cloud-based multi-tenant systems," 2019.

[150] H. AlJahdali *et al*, "Multi-tenancy in cloud computing," in *2014 IEEE 8th International Symposium on Service Oriented System Engineering*.

[151] P. K. Tiwari and B. Mishra, "Cloud computing security issues, challenges and solution," vol. 2, *(8)*, pp. 306-310, 2012.

[152] R. V. Rao and K. Selvamani, "Data security challenges and its solutions in cloud computing," vol. 48, pp. 204-209, 2015.

[153] P. M. Hoener, "Cloud computing security requirements and solutions: a systematic literature review," 2013.

[154] R. Bagate and A. Chaugule, "CLOUD ARCHITECTURE AND SECURITY MEASURES: A," vol. 1, *(2)*, pp. 24-27.

[155] V. S. K. Maddineni and S. Ragi, "Security Techniques for Protecting Data in Cloud Computing., Blekinge Institute of Technology, 2012.

[156] Y. Liu *et al*, "A survey of security and privacy challenges in cloud computing: solutions and future directions," vol. 9, *(3)*, pp. 119-133, 2015.

[157] M. A. Abd Elmonem, E. S. Nasr and M. H. Gheith, "Automating requirements elicitation of cloud-based erps," in *Proceedings of the International Conference on Advanced Intelligent Systems and Informatics 2017*.

[158] H. Tabrizchi and M. Kuchaki Rafsanjani, "A survey on security challenges in cloud computing: issues, threats, and solutions," vol. 76, *(12)*, pp. 9493-9532, 2020.

[159] PostgreSQL Global Development Group. *"PostgreSQL 15 Documentation."* https://www.postgresql.org/docs/15/index.html. (accessed 01.08., 2023).

[160] Amazon. *"AWS Cloud Security."* https://aws.amazon.com/security/. (accessed 01.08., 2023).

[161] Oracle. *"Database Security Guide."* https://docs.oracle.com/cd/B28359_01/network.111/b28531/index.html. (accessed 01.08., 2023).

[162] J. Sterman, "System Dynamics: Systems Thinking and Modeling for a Complex World.", Massachusetts Institute of Technology. Engineering Systems Division, 2002.

[163] Y. Y. Haimes, *Modeling and Managing Interdependent Complex Systems of Systems*. 2018.

[164] A. De Lucia *et al*, "An experimental comparison of ER and UML class diagrams for data modelling," vol. 15, *(5)*, pp. 455-492, 2010.

[165] A. Y. Aleryani, "Comparative study between data flow diagram and use case diagram," vol. 6, *(3)*, pp. 124-126, 2016.

[166] CYBERTEC. *"SECURE POSTGRESQL – A Reminder on Various Attack Surfaces."* https://www.cybertec-postgresql.com/en/secure-postgresql-a-reminder-on-various-attack-surfaces/?fbclid=IwAR24UGTSDjoiml12YgOZGyw0KPrr_YYRDcr32kgnALJaC-bEv1IbaAEmQZLA. (accessed 01.08., 2023).

[167] PostgreSQL Global Development Group. *"21.5. Password Authentication - Chapter 21. Client Authentication."* https://www.postgresql.org/docs/current/auth-password.html. (accessed 02.08., 2023).

[168] PostgreSQL Global Development Group. *"21.3. Authentication Methods - Chapter 21. Client Authentication."* https://www.postgresql.org/docs/15/auth-methods.html. (accessed 02.08., 2023).

[169] Oracle. *"Configuring Privilege and Role Authorization."* https://docs.oracle.com/en/database/oracle/oracle-database/23/dbseg/configuring-privilege-and-role-authorization.html#GUID-89CE989D-C97F-4CFD-941F-18203090A1AC (accessed 16.09., 2023).

[170] PostgreSQL Global Development Group. *"F.28. pgcrypto - Appendix F. Additional Supplied Modules."* https://www.postgresql.org/docs/current/pgcrypto.html. (accessed 02.08., 2023).

[171] Oracle. *"Manually Encrypting Data."* https://docs.oracle.com/en/database/oracle/oracle-database/23/dbseg/manually-encrypting-data.html#GUID-B90F0869-1F69-45D9-A835-4985A19F7AB5 (accessed 16.09., 2023).

[172] PostgreSQL Global Development Group. *"Chapter 26. Backup and Restore - Server Administration."* https://www.postgresql.org/docs/15/backup.html (accessed 17.09., 2023).

[173] AWS. *"Supported AWS resources and third-party applications."* https://docs.aws.amazon.com/aws-backup/latest/devguide/whatisbackup.html (accessed 16.09., 2023).

[174] Oracle. *"Oracle Recovery Manager (RMAN)."* https://www.oracle.com/ro/database/technologies/high-availability/rman.html#:~:text=Oracle%20Recovery%20Manager%20(RMAN)%20provides,detection%20during%20backup%20and%20restore. (accessed 16.09.2023).

[175] AWS. *"AWS Audit Manager."* https://aws.amazon.com/audit-manager/ (accessed 16.09. 2023).

[176] PostgreSQL Global Development Group. *"pgbench- PostgreSQL Client Applications."* https://www.postgresql.org/docs/14/pgbench.html. (accessed 03.08., 2023).

[177] AWS. *"Amazon CloudWatch Concepts."* https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/cloudwatch_concepts.html (accessed 16.09., 2023).

[178] Arup Nanda. *"Oracle Database 11g: The Top Features for DBAs and Developers."* https://www.oracle.com/technical-resources/articles/database/sql-11g-spa.html (accessed 16.09., 2023).

[179] R. Zahra and J. G. Vella, "Incorporating security features in system design documents utilized for cloud-based databases," in *Information Systems and Management Science: Conference Proceedings of 3rd International Conference on Information Systems and Management Science (ISMS) 2020*, .

[180] Oracle, “Historical Test Database provided by Oracle". https://www.oracle.com. (accessed 02.08., 2023).

[181] R. Camps, "Transforming N-ary Relationships to Database Schemas: An Old and Forgotten Problem," 2002.

[182] A. Watt and N. Eng, *Database Design–2nd Edition*. (2nd Edition ed.) Victoria, Australia: BCcampus, 2014.

[183] R. Feynman and C. Objectives, "Ebnf: A notation to describe syntax," pp. 10, 2016.

[184] E. Markoska *et al*, "Cloud portability standardization overview," in *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*.

[185] H. Chen and H. Liao, "A comparative study of view update problem," in *2010 International Conference on Data Storage and Data Engineering*.

[186] R. A. Teimoor, "A review of database security concepts, risks, and problems," vol. 5, *(2)*, pp. 38-46, 2021.

[187] PostgreSQL Global Development Group. *"CREATE DATABASE SQL Commands."* https://www.postgresql.org/docs/current/sql-createdatabase.html. (accessed 02.08., 2023).

[188] M. Yıldırım and I. Mackie, "Encouraging users to improve password security and memorability," vol. 18, pp. 741-759, 2019.

[189] Chad Kime. *"How to Prevent SQL Injection: 5 Key Prevention Methods."* https://www.esecurityplanet.com/threats/how-to-prevent-sql-injection-attacks/#prevention-methods. https://www.esecurityplanet.com/threats/how-to-prevent-sql-injection-attacks/#prevention-methods (accessed 01.08., 2023).

[190] PostgreSQL Global Development Group. *25.1. SQL Dump, Chapter 25. Backup and Restore."* https://www.postgresql.org/docs/10/backup-dump.html. (accessed 01.08., 2023).

[191] PostgreSQL Global Development Group. *25.3. Continuous Archiving and Point-in-Time Recovery (PITR), Chapter 25. Backup and Restore."* https://www.postgresql.org/docs/10/continuous-archiving.html. (accessed 01.08., 2023).

[192] D. Bermbach, E. Wittern and S. Tai, *Cloud Service Benchmarking*. 2017.

[193] M. Vieira and H. Madeira, "Towards a security benchmark for database management systems," in *2005 International Conference on Dependable Systems and Networks (DSN'05)*.

[194] Trustwave Holdings Inc., "AppDetectivePRO 8.6 user guide," Trustwave Holdings, Inc, 24.11. 2015.

[195] PostgreSQL Global Development Group. *"19.8. Encryption Options - Chapter 19. Server Setup and Operation."* https://www.postgresql.org/docs/current/encryption-options.html. (accessed 02.08.2023).

[196] E-SPIN. *"The De-Facto Standard for Corporate Auditors and IT Advisors."* https://www.e-spincorp.com/the-de-facto-standard-for-corporate-auditors-and-it-advisors/ (accessed 01.08.2023).

Appendix

A.1 EBNF Rules for ERM

Listing A.1: EBNF rules for ERM

1: ERM := *{"name":* string*, "strong entities":[*lstSTRONG_ENTITY*] [{, "weak entities":[*lstWEAK_ENTITY*}] [{, "SE-SR-SE relationships":[* lstSE_SR_SE_RELATIONSHIP*}] [{,"SE-WR-WE relationships":[* lstSE_WR_WE_RELATIONSHIP*}] [{,"WE-WR-WE relationships":[* lstWE_WR_WE_RELATIONSHIP*}] [{,"WE-SR-SE relationships":[* lstWE_SR_SE_RELATIONSHIP*}] [{,"Ternary relationships":[* lstTERNARY_RELATIONSHIP*}]}*

2: lstSTRONG_ENTITY := *{*STRONG_ENTITY*}* [{,*{* STRONG_ENTITY*}*}]

3: lstWEAK_ENTITY := *{*WEAK_ENTITY*}* [{,*{* WEAK_ENTITY *}*}]

4: STRONG_ENTITY := *"strong entity name"*: string [{, *"attributes": [*[{lstATT}]*],* *"strong/weak parents":[*lstSTRONG_PARENTS*]*}]

5: WEAK_ENTITY := *"weak entity name"*: string [{, *"attributes": [*[{lstPK_ATT}] [{,}] [{ lstATT }] [{, lstCOMPOSITE_ATT }]*]* }]

6: lstATT := *{*ATTRIBUTE*}* [{, *{*ATTRIBUTE*}*}]

7: lstPK_ATT := *{*PK_ATT*}* [{, *{*PK_ATT*}*}]

8: ATTRIBUTE := *"attribute name"*: string, ATT_TYPE, REQUIRED, ATT_MODE

9: PK_ATTR := *"attribute name"*: string, ATT_TYPE, *"required": "yes", "attribute mode":["non-derived", "single", "primary key simple"]*

10: ATT_TYPE := *"attribute type": "integer"* | *"boolean"* | *"character varying* [(numberOfChars)]*"* | *"date"* | *"double precision"*

11: REQUIRED := *"required": "yes"* | *"no"*

12: lstCOMPOSITE_ATT := *{*COMPOSITE_ATT*}* [{, *{*COMPOSITE_ATT*}*}]

13: COMPOSITE_ATT := *"main attribute name"*: string, *"composition": "composite"* [{, lstCOMPOSITE_ATT }], *"secondary attributes":[*lstATT*]*

14: numberOfChars := integer

15: ATT_MODE := *"attribute mode": [*DERIVE, VALUED, *"simple"]*

16: DERIVE := *"derived"* | *"non-derived"*

17: VALUED := *"single"* | *"multi"*

18: lstSTRONG_PARENT := *{*STRONG_PARENT*}* [{,*{*STRONG_PARENT*}*}]

19: STRONG_PARENT := *"strong parent"*: string , *"direct/indirect": "direct"* | *"indirect"*

20: lstSE_SR_SE_REL := *{*SE_SR_SE_REL*}* [{,*{* SE_SR_SE_REL*}*}]

21: lstSE_WR_WE_REL := *{*SE_WR_WE_REL*}* [{,*{*SE_WR_WE_REL*}*}]

22: lstWE_WR_WE_REL := {WE_WR_WE_REL} [{,{WE_WR_WE_REL}}]

23: lstWE_SR_SE_REL := {WE_SR_SE_REL} [{,{WE_SR_SE_REL}}]

24: lstTERNARY_REL := {TERNARY_REL} [{,{TERNARY_REL}}]

25: TERNARY REL := *"relationship name"*: string , *"strong participating entity 1"*: STRONG_PART_ENTITY, *"strong participating entity 2"*: STRONG_PART_ENTITY, *"strong participating entity 3"*: STRONG_PART_ENTITY [{,*"relationship attribute"*: lstREL_ATTR}]

26: SE_SR_SE_REL := *"relationship name"*: string, *"strong participating entity 1"*: STRONG_PART_ENTITY, *"strong participating entity 2"*: STRONG_PART_ENTITY [{,*"relationship attribute"*: lstREL_ATTR}]

27: SE_WR_WE_REL := *"relationship name"*: string, *"strong participating entity"*: STRONG_PART_ENTITY_1, *"weak participating entity"*: WEAK_PART_ENTITY_TOT [{,*"relationship attribute"*: lstREL_ATTR}]

28: WE_WR_WE_REL := *"relationship name"*: string, *"weak participating entity parent"*: WEAK_PART_ENTITY_1, *"weak participating entity child"*: WEAK_PART_ENTITY_TOT [{,*"relationship attribute"*: lstREL_ATTR}]

29: WE_SR_SE_REL := *"relationship name"*: string, *"weak participating entity"*: WEAK_PART_ENTITY, *"strong participating entity"*: WEAK_PART_ENTITY [{, *"relationship attribute"*: lstREL_ATTR}]

30: STRONG_PART_ENTITY := *{"strong participating entity name"*: string, REL_PART, REL_CARD*}*

31: STRONG_PART_ENTITY_1 := *{"strong participating entity name"*: string, REL_PART, *"cardinality": 1}*

32: WEAK_PART_ENTITY := *{"weak participating entity name"*: string, REL_PART, REL_CARD*}*

33: WEAK_PART_ENTITY_1 := *{"weak participating entity name"*: string, REL_PART, *"cardinality": 1}*

34: WEAK_PART_ENTITY_TOT := *{"weak participating entity name"*: string, *"participation in relationship": "total"*, REL_CARD*}*

35: REL_PART := *"participation in relationship"*: *"total"* | *"partial"*

36: REL_CARD := *"cardinality"*: *"1"* | *"M"* | *"N"* | *"P"*

37: lstREL_ATTR := {REL_ATTR*}* [{,{REL_ATTR}}]

38: REL_ATTR := *"attribute name"*: string, ATT_TYPE

A.2 Conversion of DFD to relational data structures

Pseudocode A.2 Conversion of DFD to relational data structures

```
1:  Input: sysdiag_name text
2:  Output: void
3:
4:  Algorithm:
5:    sysDiag ← sysdiag_sn of sysdiag_name
6:    dfd ← SELECT id_json
7:            FROM import_dfd
8:            WHERE id_sysdiag_serno = sysDiag
9:
10:   INSERT INTO dfd
11:   VALUES (SELECT id_json -» 'name'::text, id_sysdiag_serno
12:             FROM import_dfd
13:             WHERE id_sysdiag_serno = sysDiag
14:   RETURNING dfd_h_sn INTO current_d_sn
15:                                          >extract the external entity fragment
16:  eec ← json_array_length(dfd → external entities)
17:  if eec > 0 then
18:    dfd_extent_ee  ← function json_extract(dfd, external entities)
19:    for rec_curs in
20:      SELECT t.rel -» 'external entity name'
21:      FROM dfd_extent_ee → 'external entities' AS t(rel)
22:    do
23:      INSERT INTO dfd_external_entity
24:      VALUES (current_d_sn, rec_curs.een)
25:      RETURNING dfd_ee_sn INTO current_ee_sn
26:    end for
27:   end if
28:                                  >extract the outside external entity fragment
29:  oeec ← json_array_length(dfd → outside external entities)
30:  if oeec > 0 then
31:    dfd_extent_oee  ← function json_extract(dfd, outside external entities)
32:    for rec_curs in
33:      SELECT t.rel -» 'out ext ent name'
34:      FROM dfd_extent_oee → 'outside external entities' AS t(rel)
35:    do
36:      INSERT INTO dfd_outside_external_entity
37:      VALUES (current_d_sn, rec_curs.oeen)
38:      RETURNING dfd_oe_sn INTO current_oee_sn
39:    end for
40:   end if
41:                                          >extract the datastores fragment
42:  dsc ← json_array_length(dfd -> data stores)
43:  if dsc > 0 then
44:    dfd_extent_ds  ← function json_extract(dfd, data stores)
```

```
45:     for rec_curs in
46:       SELECT t.rel -» 'data store name'
47:       FROM dfd_extent_ds → 'data stores' AS t(rel)
48:     do
49:       INSERT INTO dfd_datastore
50:       VALUES (current_d_sn, rec_curs.dsn)
51:       RETURNING dfd_d_sn INTO current_ds_sn
52:     end for
53:  end if
54:                                                  >extract the external entities fragment
55:  pc ← json_array_length(dfd → processes)
56:  if pc > 0 then
57:    dfd_extent_p  ← function json_extract(dfd, processes)
58:    for rec_curs in
59:      SELECT t.rel -» 'process name'
60:      FROM dfd_extent_p → 'processes' AS t(rel)
61:    do
62:      INSERT INTO dfd_process
63:      VALUES (current_d_sn, rec_curs.pn)
64:      RETURNING dfd_p_sn INTO current_p_sn
65:    end for
66:  end if
67:
68: dfcounter ← json_array_length(dfd →  dataflows)
69:    if dfcounter == 0 then
70:      dfd_extent_df ← function json_extract(dfd, dataflows)
71:     for rec_curs in
72:       t.rel -» dataflow_name, t.rel -» inward_entity_type,
73:        CASE t.rel -» inward_entity_type
74:         external_entity then
75:           SELECT dfd_ee_sn
76:           FROM external_entity
77:           WHERE dfd_ee_name = t.rel-»'inward connected entity'
78:               AND dfd_ee_h_sn = current_d_sn)
79:         datastore then
80:        SELECT dfd_d_sn
81:            FROM data_store
82:            WHERE dfd_ee_name = t.rel -» 'inward connected entity'
83:                AND dfd_d_h_sn = current_d_sn
84:     process then
85:         SELECT dfd_p_sn
86:            FROM process
87:            WHERE dfd_p_name = t.rel -» 'inward connected entity'
88:                AND dfd_p_h_sn = current_d_sn)
89:    END CASE,
90:    t.rel  -» 'outward entity type' ,
91:    CASE t.rel -» outward entity type …
92:    END CASE,
```

```
93:   FROM dfd_extent_df→'dataflows' AS t(rel)
94:
95:   do                                          >insert each dataflow as a tuple
96:       INSERT INTO dfd_dataflow
97:       VALUES (current _d_sn, rec_curs.dfn, rec_curs.iet, rec_curs.iesn,
98:                   rec_curs.oet, rec_curs.oesn)
99:    RETURNING dfd_df_sn INTO current_df_sn
100:  end for
101:  end if
```

A.3 Conversion of ERM relationships to relational structures

Pseudocode A.3 Conversion of ERM relationships to relational data structures

```
1:   Input: erm_sn integer, json_fragment json
2:   Output: void
3:
4:   Algorithm:
5:   rc ← json_array_length(json_fragment → SE-SR-SE rel)
6:   If rc > 0 then
7:      erm_extent_sss ← function json_extract(json_fragment, SE-SR-SE rel)
8:    For rec_curs_sss in
9:      (erm_extent_sss → SE-SR-SE rel)
10:  Do
11:    INSERT INTO erm_relationship
12:    VALUES (erm_sn, rec_curs_sss.rn, 'strong')
13:    RETURNING erm_r_sn INTO current_r_sn
14:
15:    INSERT INTO erm_part_entity
16:    VALUES (current_r_sn, rec_curs_sss.particEnt1, rec_curs_sss.cardEnt1,
17:                  rec_curs_sss.partEnt1)
18:    RETURNING erm_pe_sn  INTO current_pe_sn
19:
20:    INSERT INTO erm_part_entity
21:    VALUES (current_r_sn, rec_curs_sss.particEnt2, rec_curs_sss.cardEnt2,
22:                  rec_curs_sss.partEnt2)
23:    RETURNING erm_pe_sn  INTO current_pe_sn
24:
25:   For rec_curs_rel_att in
26:     SELECT rel→'relationship attribute' -» 'attribute name', rel →
27:                  'relationship attribute' -» 'attribute type'
28:     FROM erm_extent_sss → SE-SR-SE rel
29:     WHERE rel → 'relationship attribute' -» 'attribute name' IS NOT NULL
30:     AND rel -» 'relationship name' = current_r_sn.name
31:   Do
32:     INSERT INTO erm_attribute_relationship
```

```
      VALUES (current_r_sn, rec_curs_rel_att.attName, rec_curs_rel_att.attType)
      RETURNING erm_ra_sn INTO current_ra_sn
    End for
  End for
End if

rc ← json_array_length(json_fragment → WE-SR-SE rel)
If rc > 0 then
    erm_extent_wss ← function json_extract(json_fragment, WE-SR-SE rel)
 For rec_curs_wss in
    (erm_extent_wss → WE-SR-SE rel)
 Do
    INSERT INTO erm_relationship
    VALUES (erm_sn, rec_curs_wss.rn, 'strong')
    RETURNING erm_r_sn INTO current_r_sn

    INSERT INTO erm_weak_part_entity
    VALUES (current_r_sn, rec_curs_wss.particEnt1, rec_curs_wss.cardEnt1,
                rec_curs_wss.partEnt1)
    RETURNING erm_wpe_sn  INTO current_wpe_sn

    INSERT INTO erm_part_entity
    VALUES (current_r_sn, rec_curs_wss.particEnt2, rec_curs_wss.cardEnt2,
                rec_curs_wss.partEnt2)
    RETURNING erm_pe_sn  INTO current_pe_sn

   For rec_curs_rel_att in
     SELECT rel→'relationship attribute' -» 'attribute name', rel →
                'relationship attribute' -» 'attribute type'
     FROM erm_extent_wss → WE-SR-SE rel
     WHERE rel → 'relationship attribute' -» 'attribute name' IS NOT NULL
     AND rel -» 'relationship name' = current_r_sn.name
   Do
     INSERT INTO erm_attribute_relationship
     VALUES (current_r_sn, rec_curs_rel_att.attName, rec_curs_rel_att.attType)
     RETURNING erm_ra_sn INTO current_ra_sn
    End for
  End for
End if

rc ← json_array_length(json_fragment → SE-WR-WE rel)
If rc > 0 then
    erm_extent_sww ← function json_extract(json_fragment, SE-WR-WE rel)
 For rec_curs_sww in
    (erm_extent_sww → SE-WR-WE rel)
 Do
    INSERT INTO erm_relationship
    VALUES (erm_sn, rec_curs_sww.rn, 'weak')
```

```
81:     RETURNING erm_r_sn INTO current_r_sn
82:
83:     INSERT INTO erm_part_entity
84:     VALUES (current_r_sn, rec_curs_sww.particEnt1, rec_curs_sww.cardEnt1,
85:                     rec_curs_sww.partEnt1)
86:     RETURNING erm_pe_sn  INTO current_pe_sn
87:
88:     INSERT INTO erm_weak_part_entity
89:     VALUES (current_r_sn, rec_curs_sww.particEnt2, rec_curs_sww.cardEnt2,
90:                     rec_curs_sww.partEnt2)
91:     RETURNING erm_wpe_sn  INTO current_wpe_sn
92:
93:   For rec_curs_rel_att in
94:       SELECT rel→'relationship attribute' -» 'attribute name', rel →
95:                     'relationship attribute' -» 'attribute type'
96:       FROM erm_extent_sww → SE-WR-WE rel
97:       WHERE rel → 'relationship attribute' -» 'attribute name' IS NOT NULL
98:       AND rel -» 'relationship name' = current_r_sn.name
99:   Do
100:      INSERT INTO erm_attribute_relationship
101:      VALUES (current_r_sn, rec_curs_rel_att.attName, rec_curs_rel_att.attType)
102:      RETURNING erm_ra_sn INTO current_ra_sn
103:  End for
104:  End for
105:  End if
106:
107:  rc ← json_array_length(json_fragment → WE-WR-WE rel)
108:  If rc > 0 then
109:       erm_extent_www ← function json_extract(json_fragment, WE-WR-WE rel)
110:   For rec_curs_www in
111:        (erm_extent_www → WE-WR-WE rel)
112:   Do
113:     INSERT INTO erm_relationship
114:     VALUES (erm_sn, rec_curs_www.rn, 'weak')
115:     RETURNING erm_r_sn INTO current_r_sn
116:
117:     INSERT INTO erm_weak_part_entity
118:     VALUES (current_r_sn, rec_curs_www.particEnt1, rec_curs_www.cardEnt1,
119:                     rec_curs_www.partEnt1)
120:     RETURNING erm_wpe_sn  INTO current_wpe_sn
121:
122:     INSERT INTO erm_weak_part_entity
123:     VALUES (current_r_sn, rec_curs_www.particEnt2, rec_curs_www.cardEnt2,
124:                     rec_curs_www.partEnt2)
125:     RETURNING erm_wpe_sn  INTO current_wpe_sn
126:
127:      For rec_curs_rel_att in
128:         SELECT rel→'relationship attribute' -» 'attribute name', rel →
```

```
                        'relationship attribute' -» 'attribute type'
        FROM erm_extent_www → WE-WR-WE rel
        WHERE rel → 'relationship attribute' -» 'attribute name' IS NOT NULL
        AND rel -» 'relationship name' = current_r_sn.name
    Do
        INSERT INTO erm_attribute_relationship
        VALUES (current_r_sn, rec_curs_rel_att.attName, rec_curs_rel_att.attType)
        RETURNING erm_ra_sn INTO current_ra_sn
    End for
  End for
  End if

rc ← json_array_length(json_fragment → Ternary relationships)
If rc > 0 then
   erm_extent_tern ← function json_extract(json_fragment, Ternary relationships)
For rec_curs_tern in
     (erm_extent_tern → Ternary relationships)
Do
    INSERT INTO erm_relationship
    VALUES (erm_sn, rec_curs_tern.rn, 'strong')
    RETURNING erm_r_sn INTO current_r_sn

    For i from   to 3
    Do
       INSERT INTO erm_part_entity
       VALUES (current_r_sn, rec_curs_tern.particEnt[i], rec_curs_tern.cardEnt[i],
                  rec_curs_tern.partEnt[i])
       RETURNING erm_pe_sn INTO current_pe_sn
    End for

     For rec_curs_rel_att in
       SELECT rel→'relationship attribute' -» 'attribute name', rel →
                   'relationship attribute' -» 'attribute type'
       FROM erm_extent_tern → Ternary relationship
       WHERE rel → 'relationship attribute' -» 'attribute name' IS NOT NULL
       AND rel -» 'relationship name' = current_r_sn.name
     Do
       INSERT INTO erm_attribute_relationship
       VALUES (current_r_sn, rec_curs_rel_att.attName, rec_curs_rel_att.attType)
       RETURNING erm_ra_sn INTO current_ra_sn
      End for
End for
End if
```

A.4 Other key user settings

A.4.1 Cloud Service Provider Settings

```
--Create CSP Template/Role
CREATE ROLE csp WITH
NOSUPERUSER
CREATEDB
CREATEROLE
NOINHERIT
REPLICATION
NOBYPASSRLS
NOLOGIN;

--Create CSP User
CREATE ROLE cspUser WITH
INHERIT
CREATEDB
CREATEROLE
LOGIN
CONNECTION LIMIT 1
ENCRYPTED PASSWORD 'cspUserPass'
IN ROLE csp;

--Create database for CSP User
CREATE DATABASE cspDb
WITH OWNER cspUser
ENCODING = 'UTF8'
LC_COLLATE = 'English_Malta.1252'
LC_CTYPE = 'English_Malta.1252'
CONNECTION LIMIT = -1;

REVOKE ALL ON DATABASE cspDb FROM public;

--Create software provider DBA role
CREATE ROLE spdba WITH
NOSUPERUSER
CREATEDB
CREATEROLE
NOINHERIT
NOREPLICATION
NOBYPASSRLS
NOLOGIN;

GRANT pg_read_server_files TO softwareproviderdba;

--Create software provider developer role
CREATE ROLE spdeveloper WITH
NOSUPERUSER
NOCREATEDB
NOCREATEROLE
NOINHERIT
NOREPLICATION
NOBYPASSRLS
NOLOGIN;

--Create software provider DBA user
CREATE ROLE spdbaUser WITH
INHERIT
CREATEDB
CREATEROLE
LOGIN
CONNECTION LIMIT -1
ENCRYPTED PASSWORD 'spdbaUserPass'
IN ROLE spdba;

GRANT pg_read_server_files TO gigabytedba;
```

```
64.
65. --Create software provider developer user
66. CREATE ROLE spdeveloperUser WITH
67. INHERIT
68. CREATEDB
69. CREATEROLE
70. LOGIN
71. CONNECTION LIMIT -1
72. ENCRYPTED PASSWORD 'spdeveloperUserPass'
73. IN ROLE spdeveloper;
```

A.4.2 Software Provider Settings

```
 1. --Create database for software provider
 2. CREATE DATABASE spdb
 3. WITH OWNER softwareproviderdbaUser
 4. ENCODING = 'UTF8'
 5. LC_COLLATE = 'English_Malta.1252'
 6. LC_CTYPE = 'English_Malta.1252'
 7. CONNECTION LIMIT = -1;
 8.
 9. REVOKE ALL ON DATABASE spdb FROM public;
10. REVOKE ALL ON DATABASE spdb FROM spdeveloperUser;
11. GRANT CONNECT ON DATABASE spdb TO spdeveloperUser;
12. CREATE SCHEMA "spdba" AUTHORIZATION spdbaUser;
13. CREATE SCHEMA "spdeveloper" AUTHORIZATION spdbaUser;
14. GRANT USAGE ON SCHEMA "spdeveloper" TO spdeveloperUser;
15. CREATE SCHEMA "spsecurity" AUTHORIZATION spdbaUser;
16. GRANT USAGE ON SCHEMA "spsecurity" TO spdeveloperUser;
17.
18. --Create tenant DBA role
19. CREATE ROLE tenantdba WITH
20. NOSUPERUSER
21. NOCREATEDB
22. CREATEROLE
23. NOINHERIT
24. NOREPLICATION
25. NOBYPASSRLS
26. NOLOGIN;
```

Appendix B

B.1 Common Tables used in all systems

Table name	Schema
roles	softwareprovider
views	softwareprovider
viewsConditions	softwareprovider
user	tenant
userconditions	tenant

B.2 Common Functions used in all systems

Function name	Schema
insert_user	tenant
user_creation	tenant
update_user	tenant
delete_user	tenant
user_deletion	tenant
login	tenant
insert_usercondition	tenant
update_usercondition	tenant
delete_usercondition	tenant
getroles	tenant
getpartitions	tenant
insert_csvdata	tenant
enable_tenantusers	tenant
disable_tenantusers	tenant
getProcesses	tenant
getprocessparams	tenant

B.3 List of SQL Scripts and storage locations

B.3.1 Software Provider Tables

Table name	Schema
tenant	dba

import_tenanttables	dba
import_tenantfunctions	dba
tenant_tables	dba
tenant_functions	dba
systemDiagrams	dba
import_dfd	dba
dfd	dba
dfd_external_entity	dba
dfd_outsoide_external_entity	dba
dfd_datastore	dba
dfd_process	dba
dfd_dataflow	dba
import_elh	dba
elh	dba
elh_process	dba
import_erm	dba
erm	dba
erm_entity	dba
erm_weak_entity	dba
erm_attribute_entity	dba
erm_attribute_weak_entity	dba
erm_relationship	dba
erm_part_ent	dba
erm_weak_part_ent	dba
erm_attribute_relationship	dba
erm_relationship_entity	dba
erm_attribute_relationship_entity	dba
pre_checks	security
post_checks	security
crudMatrixPP	security
crudMatrix	security
crud_checks	security
import_roles	dba
rolesProcesses	dba
rolesprocessestables	dba
import_requirements	dba

B.3.2 Functions for data import

Function name	Schema
is_json	developer
import_data	dba
json_extract	developer

B.3.3 Functions for common tenant tables and functions

Function name	Schema
tenant_import	developer
tenanttables_import2predicates	dba
tenantfunctions_import2predicates	dba

B.3.4 Functions for ERM, ELH, DFD

Function name	Schema
sysDiag_import	developer
erm_updateSysDiag	dba
erm_import2predicates	developer
erm_relationships_import2predicates	developer
elh_updateSysDiag	dba
elh_import2predicates	developer
elh_import2predicates_processes	developer
elh_any	security
dfd_updateSysDiag	dba
dfd_import2predicates	developer
delete_elh_predicates	developer
delete_elh_predicates	developer
delete_elh_predicates	developer
delete_erm_predicates	developer

B.3.5 Functions for Pre and Post checks

Function name	Schema
erm_correctentitiesinrel	security
erm_recursiverel_partialpart	security
erm_ternaryrel_onlystrongent	security
erm_ternaryrel_different	security

erm_ent_connectedto_rel	security
erm_weakent_connectedto_rel	security
erm_weakent_connectedto_ent	security
recursivefunctionrel	security
checkifstrongfound	security
elh_noduplicate_processes	security
elh_atleasttwo_processes	security
dfd_atleastoneprocess	security
dfd_correctdataflowtypes	security
dfd_process_input_datafow	security
dfd_process_output_datafow	security
dfd_ee_connectedto_dataflow	security
dfd_oe_connectedto_dataflow	security
dfd_processleaf_connectedto_datastore	security
dfd_processnonleaf_notconnectedto_datastore	security
dfd_process_ee	security
dfd_mainprocess_connectedto_entity	security
dfd_processnotmain_notconnectedto_entity	security
dfd_processnotleaformain_connectedto_process	security
dfd_mainprocessnotleaf_notconnectedto_enxtent	security
insert_precheck	security
loopthroughprechecks	security
elh_dfd	security
erm_dfd	security
dfd_erm	security
erm_elh	security
insert_postcheck	security
loopThroughPostChecks	security

B.3.6 Functions for CRUD creation and checks

Function name	**Schema**
crud_matrixPP	security
crud_matrix	security
insertintocrud	security
IUD	security
crud_checkSelect	security
crud_checkInsert	security
crud_checkUpdate	security

crud_checkDelete	security
crud_checkTables	security
crud_checkProcesses	security
insert_crudchecks	security
loopThroughCRUDChecks	security

B.3.7 Functions for CRUD creation and checks

Function name	Schema
roles_updateTenantId	dba
roles_checkColumnExist	security
roles_checkroleexist	security
roles_atleastonefunction	security
role_process	dba
insert_roleprocess	dba
roles_process_table	dba
insert_rolesprocessestables	dba
roles_roleentity	security
roles_rolefunction	security

B.3.8 Functions for Post-implementation checks

Function name	Schema
reachability	security
accessibility	security
nonredundancy	security
inference	security

B.3.9 Functions for Tenant-specific specifications

Function name	Schema
requirements_updateTenantId	dba
requirements_checks	security
req_checkcolumnexist	security

B.3.10 Database architecture functions used in SQL Script generation

Function name	Schema
codegeneration	dba

sd_create_tenant	dba
sd_create_tenant_environment	dba
ss_create_tenant	dba
ss_create_tenant_environment	dba
st_create_tenant	dba
st_create_tenant_environment	dba
create_database	dba
create_schema	dba
create_extensions	dba
create_tables	dba
create_processes	dba
create_roles	dba
create_user	dba
create_trigger_tt	dba
grant_create_privilege	dba
grant_connect_privilege	dba
grant_execute_privileges	dba
grant_usage_privilege	dba
grant_select_privileges	dba
grant_insert_privileges	dba
grant_update_privileges	dba
grant_delete_privileges	dba
grant_schematables_privilege	dba
grant_schemafunctions_privilege	dba
grant_domain_privilege	dba
grant_foreigndatawrapper_privilege	dba
grant_foreignserver_privilege	dba
grant_language_privilege	dba
grant_type_privilege	dba
grant_tablespace_privilege	dba
grant_sequence_privilege	dba
grant_largeobject_privilege	dba
revoke_schema_privilege	dba
revoke_schematables_privilege	dba
revoke_schemafunctions_privilege	dba
revoke_domain_privilege	dba
revoke_foreigndatawrapper_privilege	dba
revoke_foreignserver_privilege	dba
revoke_language_privilege	dba

revoke_type_privilege	dba
revoke_tablespace_privilege	dba
revoke_sequence_privilege	dba
revoke_largeobject_privilege	dba
create_tenant_tables	dba
create_tenant_functions	dba
enable_rlsAndPolicies_systemtables	dba
enable_rlsAndPolicies_conditiontables	dba
enable_rlsAndPolicies_tenanttables	dba
backup_function	developer
roles_import2predicates	developer
tenant_import2predicates	developer

B.3.11 Invoked functions during SQL Script generation

Function name	**Schema**
erm_all_entities	dba
erm_strong_entities	dba
erm_weak_entities	dba
erm_single_attributes_nderived	dba
erm_multivalued_attributes	dba
erm_pk_attributes	dba
erm_required_attributes	dba
erm_relationships	dba
erm_participatingEntities	dba
erm_strongPartEntities	dba
erm_weakPartEntities	dba
erm_pk_attributesandtypes	dba
erm_relationship_attributes	dba
tenant_entities	dba
tenant_functions	dba
tenant_entities_sp	dba
tenant_entities_tenant	dba
tenant_functions_sp	dba
tenant_functions_tenant	dba
dfd_external_entities	dba
dfd_outside_external_entities	dba
get_pk	dba
dfd_processes	dba

create_attributes	dba
create_multivaluedAtts	dba
primarykeys	dba
notnull	dba
create_relationships	dba
create_role	dba
revoke_database_privilege	dba
select_privilege	dba
execute_privilege	dba
insert_privilege	dba
update_privilege	dba
delete_privilege	dba
enable_rls	dba
create_policy	dba
create_view	dba

Appendix C

C.1 Scott Case Study

The first case study presented in this section is centred around the 'Scott' system, which portrays a simple company scenario whereby each employee is associated with a particular department. In addition to keeping track of all wages and salaries, the system indicates the manager of each employee by means of a self-referencing relationship [47]. This system is straightforward in nature but contains sufficient information to assess the main features of the proposed tool. More complex features are analysed in the subsequent study.

C.1.1 System Design Diagrams

This data set is made up solely of three (3) tables, storing information related to employees, departments, and salaries. Figure hereunder, indicates the ERM diagrammatic representation of the 'Scott' database. The relationships between each table are illustrated via connecting lines and primary key fields are denoted by an underline.

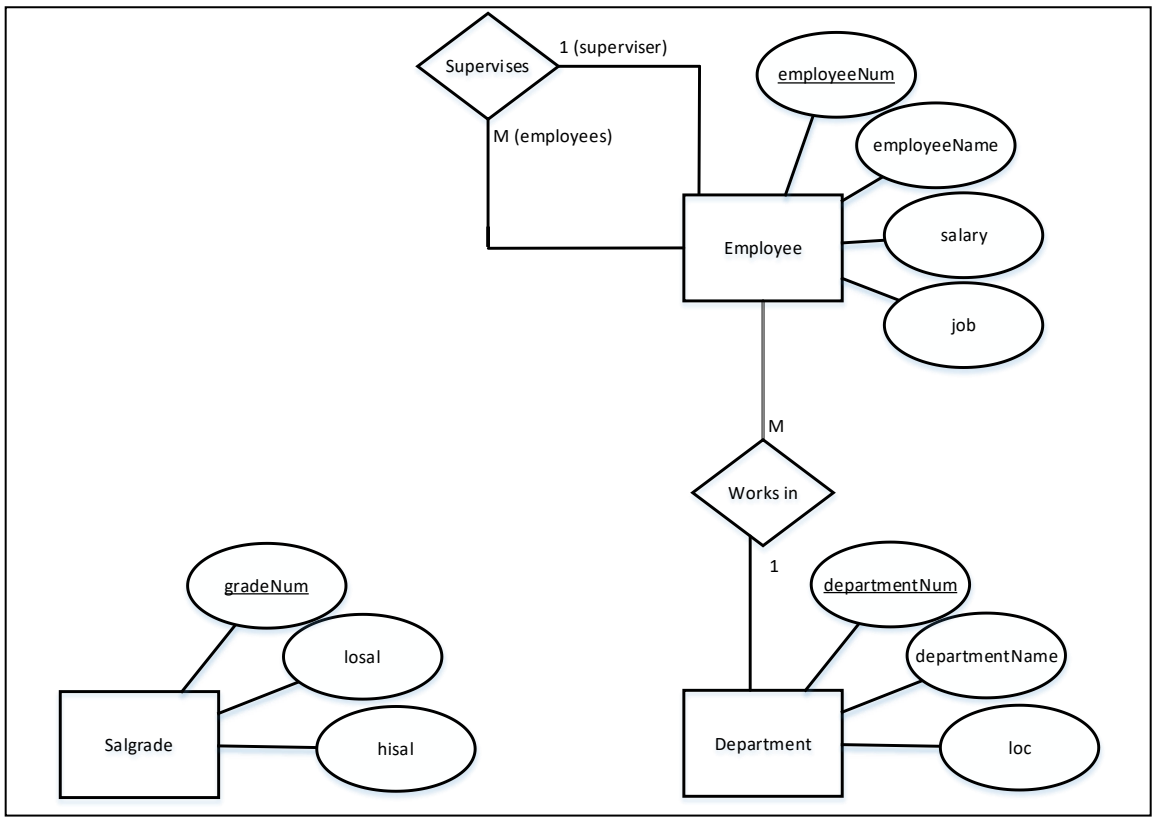


Figure C.1.1: ERM for 'Scott' system

Figure C.1.2 below, denotes the equivalent JSON representation of the ‘Employee’ entity, and its attributes.

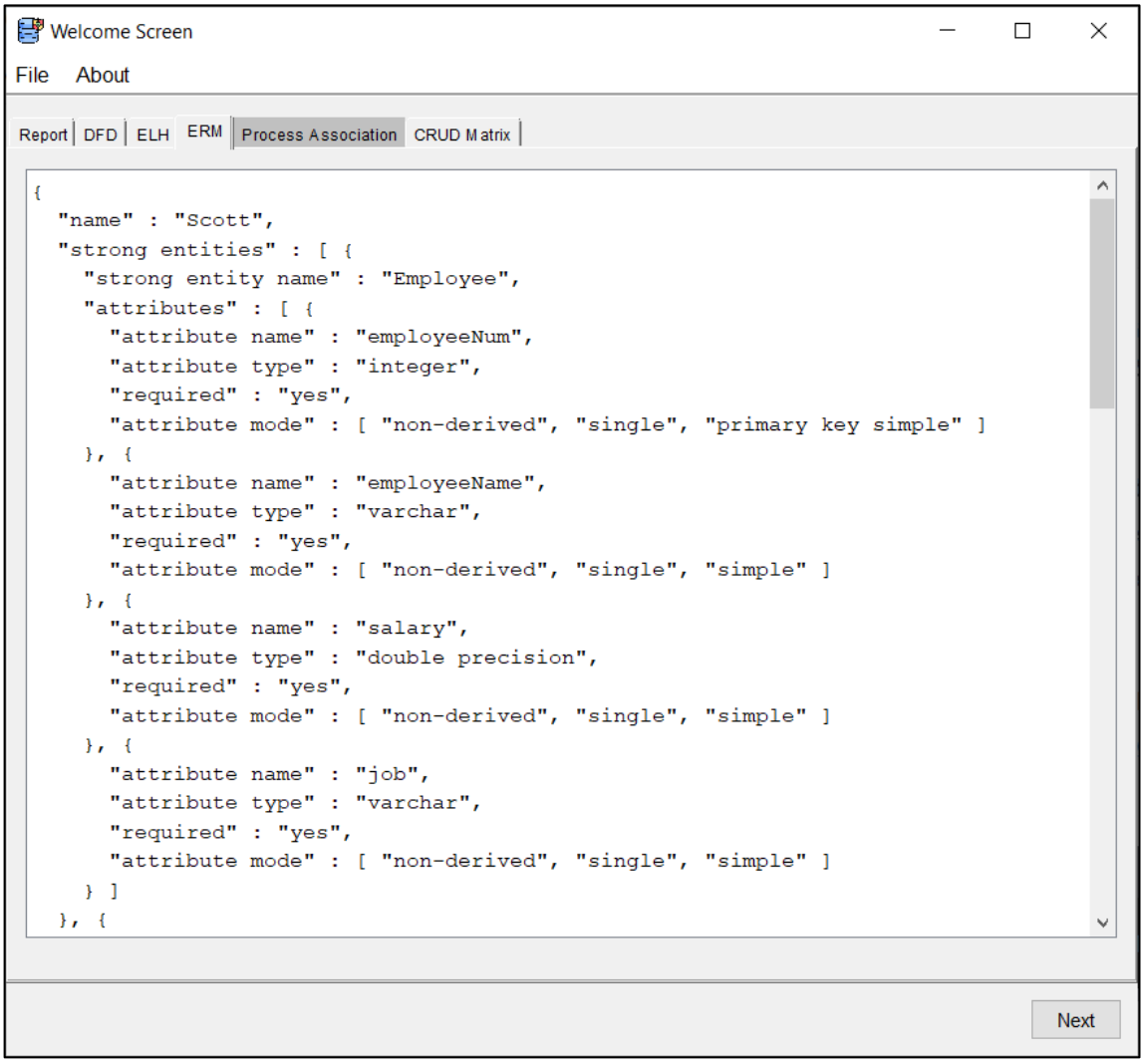


Figure C.1.2: JSON representation of ‘Employee’ entity

The JSON file commences with the 'name' component, a string indicating the system’s name. It is then followed by the ‘strong entities’ array, denoting all entities comprising the prior system (‘Scott’ in this case). Only one entity, 'Employee,' is illustrated for the purpose of this example. Each entity is described via the ‘attributes’ array which contains information related to the entity's characteristics and their accompanying descriptions. In this example, the first attribute displayed, ‘employeeNum’ is integer-based and mandatory, thus null values cannot be associated with this attribute. Moreover, the nested array ‘attribute mode’, indicates that the ‘employeeNum’ attribute is the primary key and must not be inferred or comprised of multiple values.

Subsequently, the different states of each entity were denoted via the construction of ELH diagrams. Figures C.1.3 and C.1.4 below illustrate the developed ELH and its equivalent JSON representation for the 'Department' entity. It can be observed that the execution of the three functions has an effect on the entity's state. The 'newDepartment' process inserts new tuples into the 'Department' table, whilst the 'modifyDepartmentInformation' function is updates or changes the attributes of already existent department tuples. Finally, the 'closeDepartment' function is capable of removing tuples of departments which are no longer in operation.

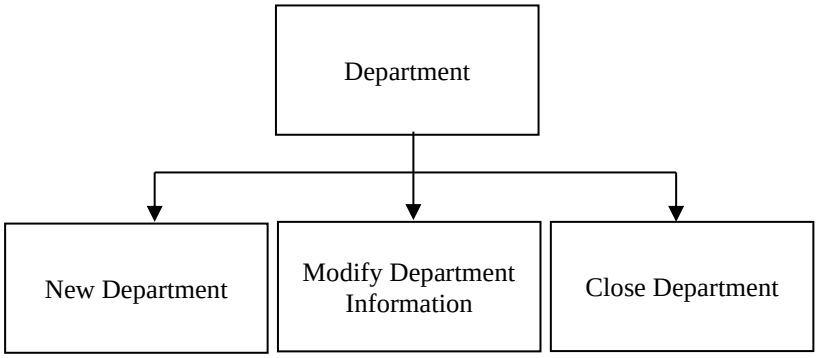


Figure C.1.3: Equivalent ELH in JSON

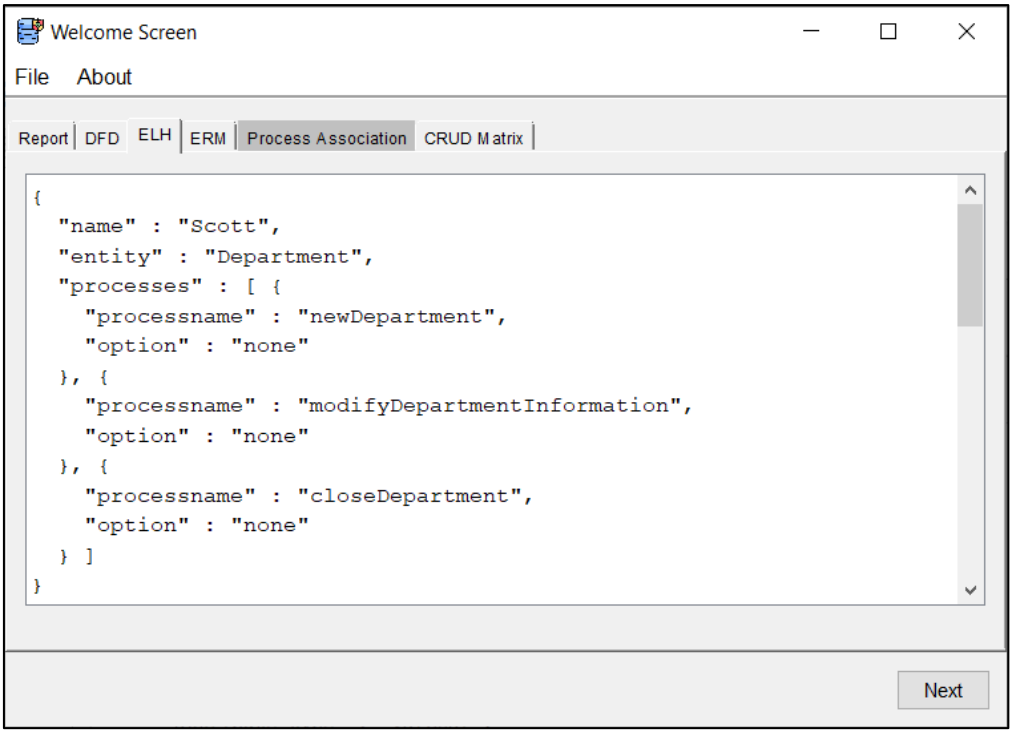


Figure C.1.4: ELH Diagram for 'Department' table

Finally, all permissible dataflows in the system were highlighted by means of a DFD (Figure C.1.5).

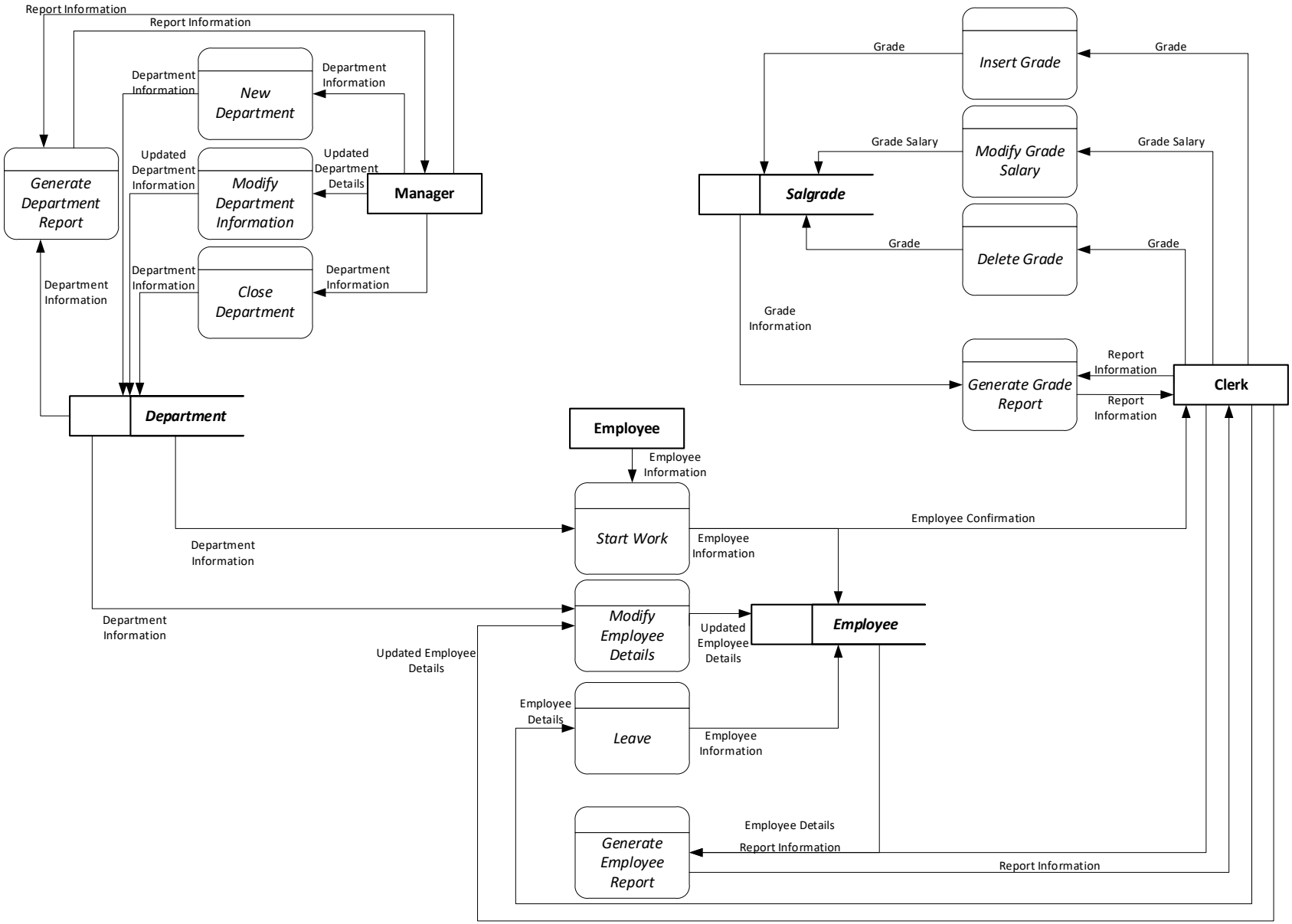


Figure C.1.5: DFD for 'Scott' system

Figure C.1.6 displays the JSON used to represent the external entities, data stores and processesfor this system. The dataflow array (not illustrated in the Figure) can be found in the last part of the JSON file and indicates the name of the dataflows and their corresponding connecting entities' names and type. For instance, examining one of the dataflows reveals a data transfer triggered by the external entity 'Clerk,' as signified by the 'outward connected entity' and 'outward entity type' components. The latter entity passes the required information to the 'modifyEmployeeDetails' process (indicated by 'outward connected entity' and 'outward entity type') so that proper execution of the process can occur.

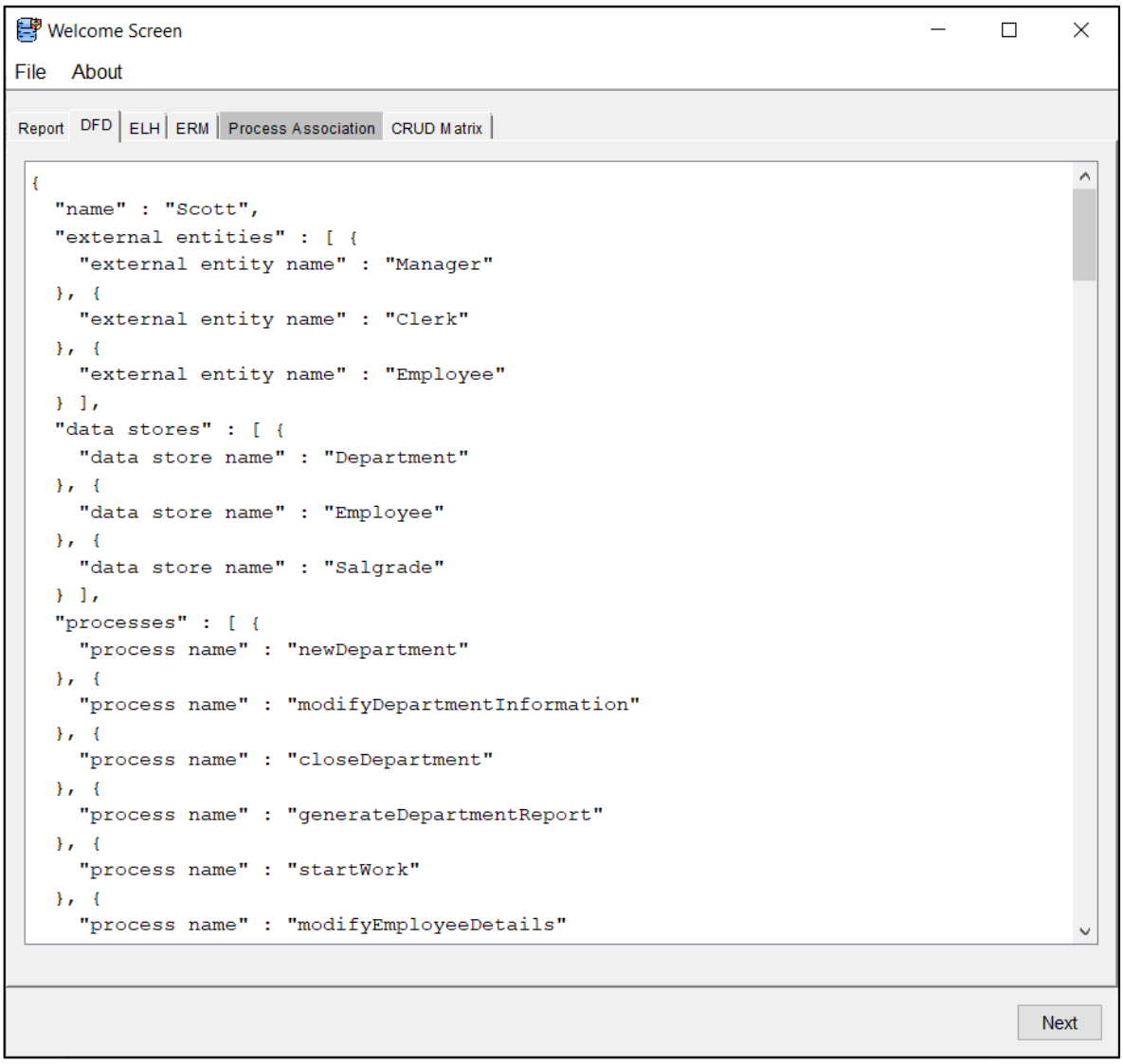


Figure C.1.6: Part of JSON-formatted DFD for 'Scott' system

C.1.2 Pre and Post checks

Analogous to what happened when the 'Banking System' was presented, the proposed tool then conducts a series of security checks based on the three diagrams presented. The outcome of each individual diagram checks, following insertion of each JSON file elucidated in Section C.1. is as follows.

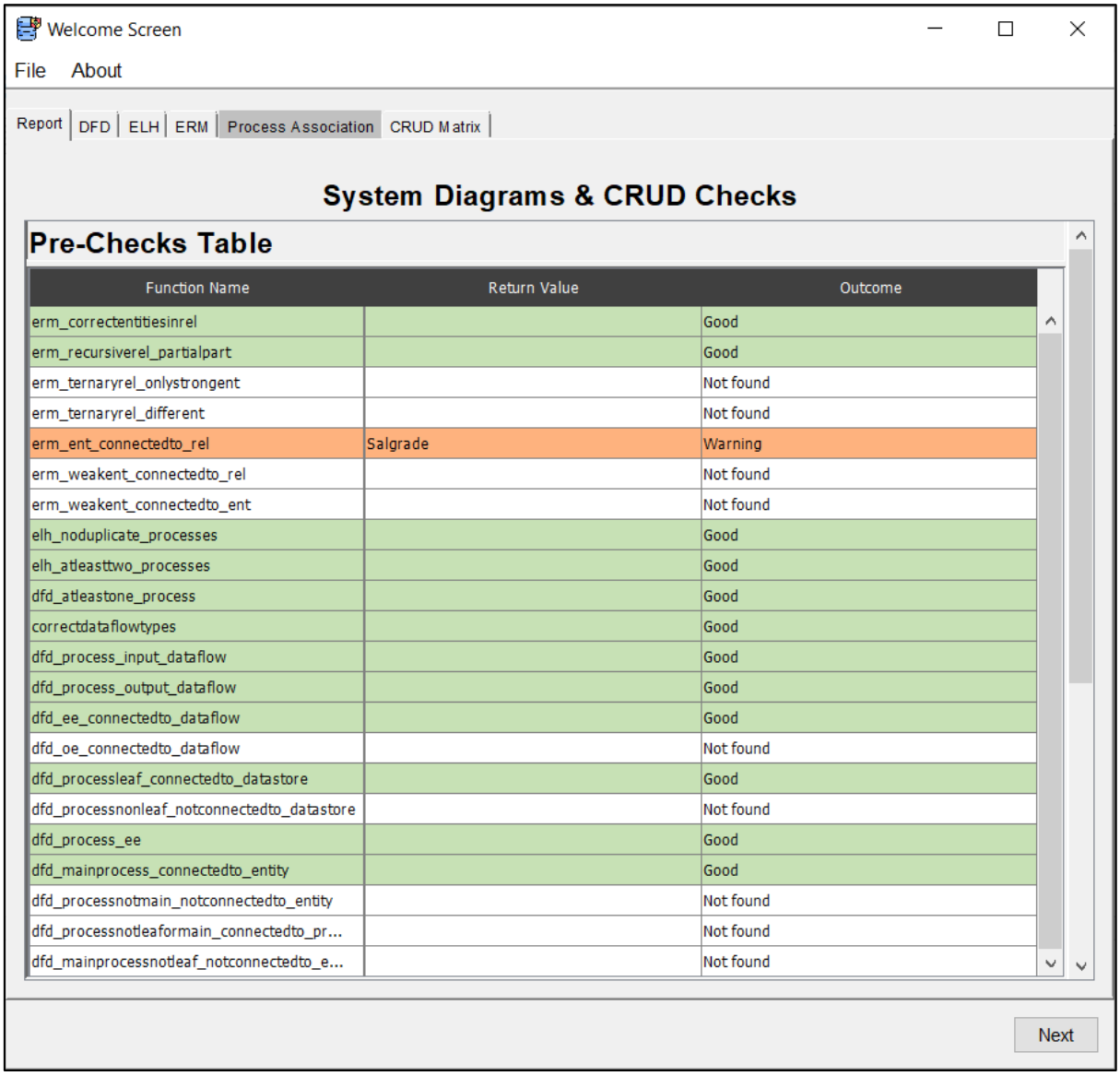


Figure C.1.7: Pre-checks for 'Scott' system

In the 'Scott' database, no weak entities or ternary relationships were detected and hence, any checks related to such instances were skipped. On the other hand, all dataflows in the DFD are connected to at least one process ('dfd_atleastone_process'

function) and all recursive relationships in the ERM have at least one of the participating entity instances in partial participation ('erm_recursiverel_partialpart' process). Conclusively, the 'Warning' output, indicates that following check execution, some non-conforming instances were identified. However, such instances do not always imply a security vulnerability. In some cases, the utilisation of such instances might be done deliberately for the appropriate functioning of the system. Hence, the user is notified about such instances but can still continue program execution if deemed fitting. In the 'Scott' case study one warning can be noted, indicating that 'Salgrade' is a stand-alone entity. After considering its functionality, one can denote that this is not a mistake and thus, since no 'Bad' outcomes are observed, the next stage of the CASE tool can be executed.

As already discussed in the case study presented in Chapter 7, this involves performing a few checks targeted towards assessing the three inputted system diagrams collectively. Since all checks conducted on the 'Scott' system diagrams succeeded (Figure C.1.8), generation of the first-level CRUD matrix (described subsequently) can take place.

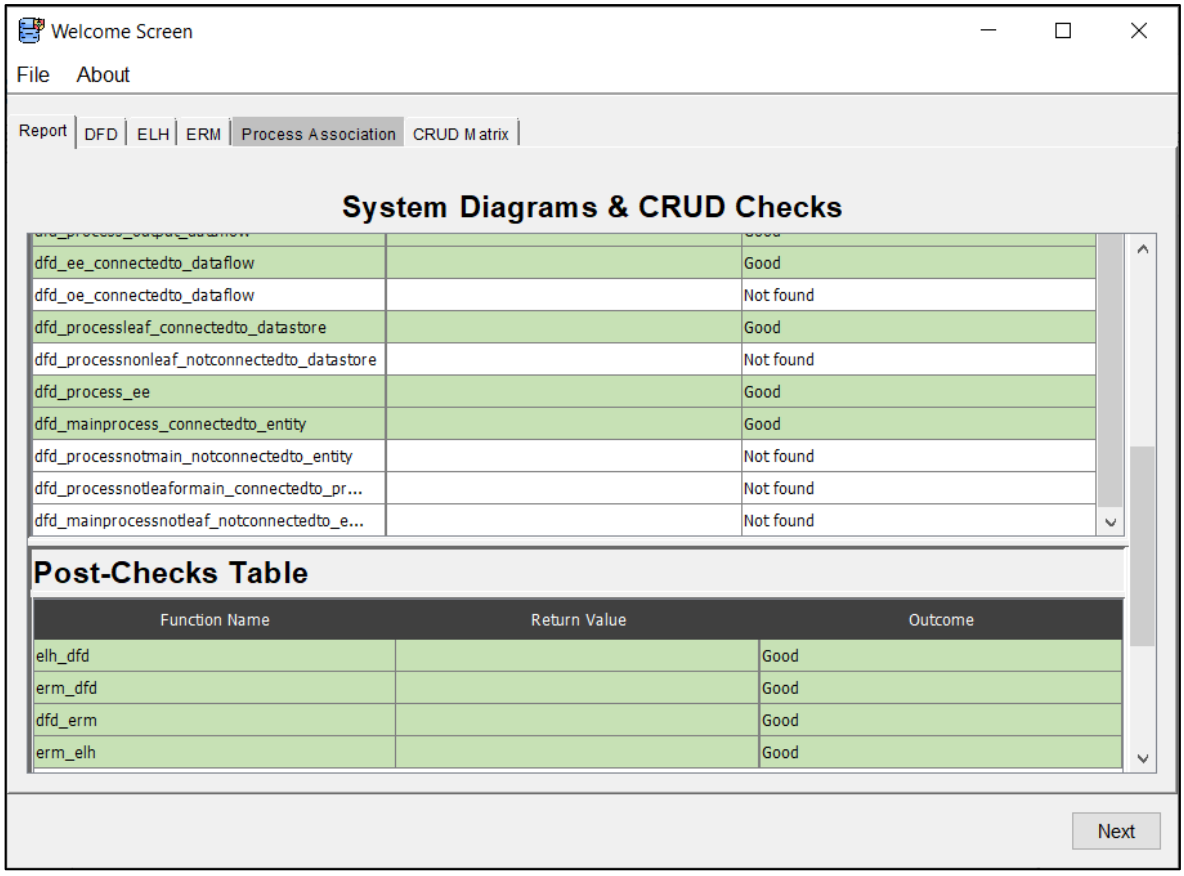
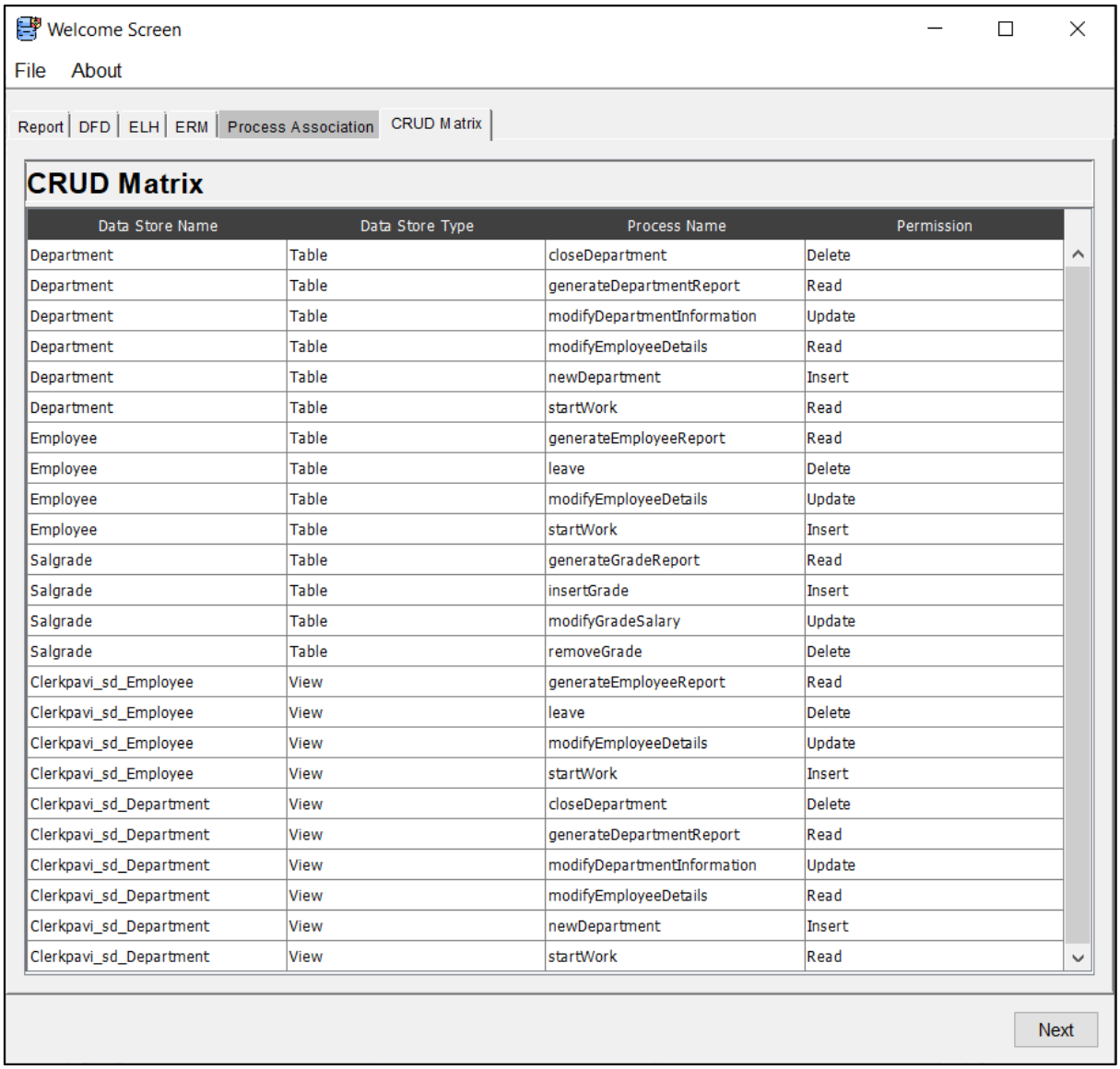


Figure C.1.8: 'Scott' system diagram checks

C.1.3 Generation of first-level CRUD Matrix

The first level CRUD matrix is generated for the ‘Scott’ system, based on the pre-sented system diagrams is depicted hereunder.



Data Store Name	Data Store Type	Process Name	Permission
Department	Table	closeDepartment	Delete
Department	Table	generateDepartmentReport	Read
Department	Table	modifyDepartmentInformation	Update
Department	Table	modifyEmployeeDetails	Read
Department	Table	newDepartment	Insert
Department	Table	startWork	Read
Employee	Table	generateEmployeeReport	Read
Employee	Table	leave	Delete
Employee	Table	modifyEmployeeDetails	Update
Employee	Table	startWork	Insert
Salgrade	Table	generateGradeReport	Read
Salgrade	Table	insertGrade	Insert
Salgrade	Table	modifyGradeSalary	Update
Salgrade	Table	removeGrade	Delete
Clerkpavi_sd_Employee	View	generateEmployeeReport	Read
Clerkpavi_sd_Employee	View	leave	Delete
Clerkpavi_sd_Employee	View	modifyEmployeeDetails	Update
Clerkpavi_sd_Employee	View	startWork	Insert
Clerkpavi_sd_Department	View	closeDepartment	Delete
Clerkpavi_sd_Department	View	generateDepartmentReport	Read
Clerkpavi_sd_Department	View	modifyDepartmentInformation	Update
Clerkpavi_sd_Department	View	modifyEmployeeDetails	Read
Clerkpavi_sd_Department	View	newDepartment	Insert
Clerkpavi_sd_Department	View	startWork	Read

Figure C.1.9: First- level CRUD matrix for 'Scott'

By examining Figure C.1.9, it becomes apparent that the ‘generateEmploy-eeReport’ process requires a SELECT (Read) privilege on the ‘Employee’ table, whilst the ‘closeDepartment’ function necessitates a DELETE privilege on the ‘Department’ table. On the other hand, an INSERT privilege on table ‘Salgrade’ is required for the

proper execution of the ‘insertGrade’ function, whilst the ‘Employee’ table is UPDATEd during execution of the ‘modifyEmployeeDetails’ process. Similar interpretations can be applied to the remaining tuples of this matrix.

The proposed tool proceeds by conducting a thorough evaluation through a set of checks on the developed CRUD matrix. The results obtained following execution of such checks are displayed in tabular format in Figure C.1.10.

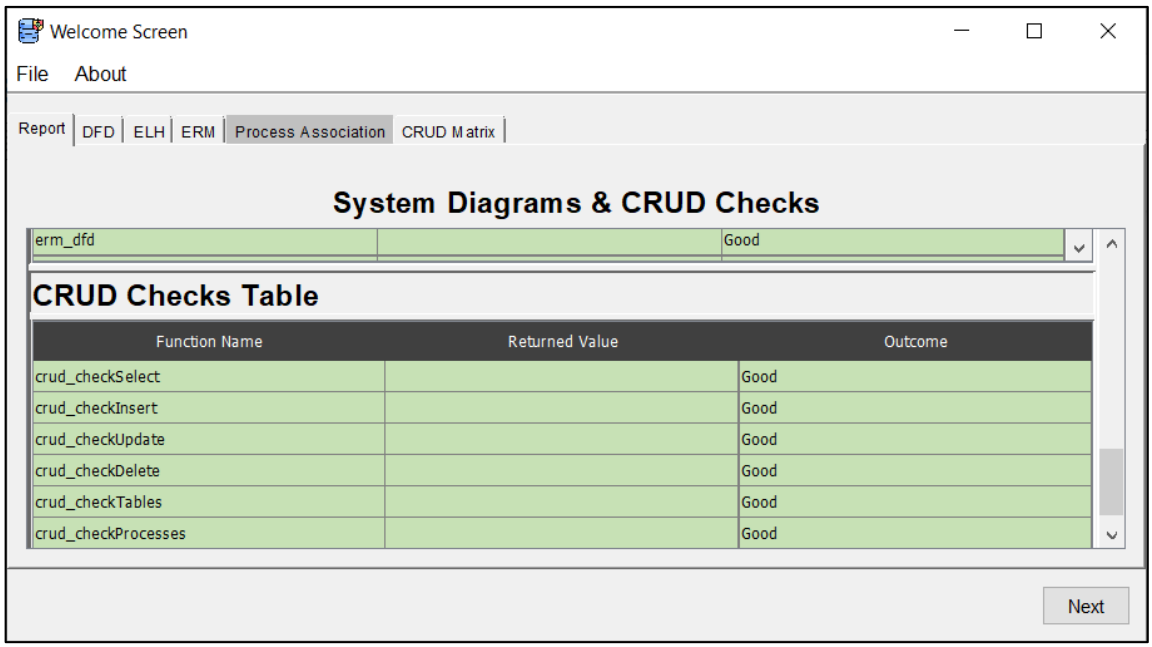


Figure C.1.10: CRUD Checks for ‘Scott’ system

It can be noted that every check yielded a positive outcome (‘Good’), signifying that all elements scrutinised during each assessment were not only present but also adhered to the stipulations outlined by the inspection. Hence, it can be inferred that all three (3) system entities are associated with at least one SELECT, INSERT, UPDATE, and DELETE privilege. Additionally, it can also be deduced that the CRUD matrix comprehensively incorporates all tables and processes specified in the preceding system design diagrams (confirmed via the ‘crud_checkTables’ and ‘crud_checkProcesses’ checks).

C.1.4 Association of roles and processes

Upon successful completion of the checks outlined in the previous section, the system identifies the key roles and their associated processes required to effectively

carry out their respective tasks. The roles-processes matrix produced for the ‘Scott’ system can be observed in Figure C.1.11.

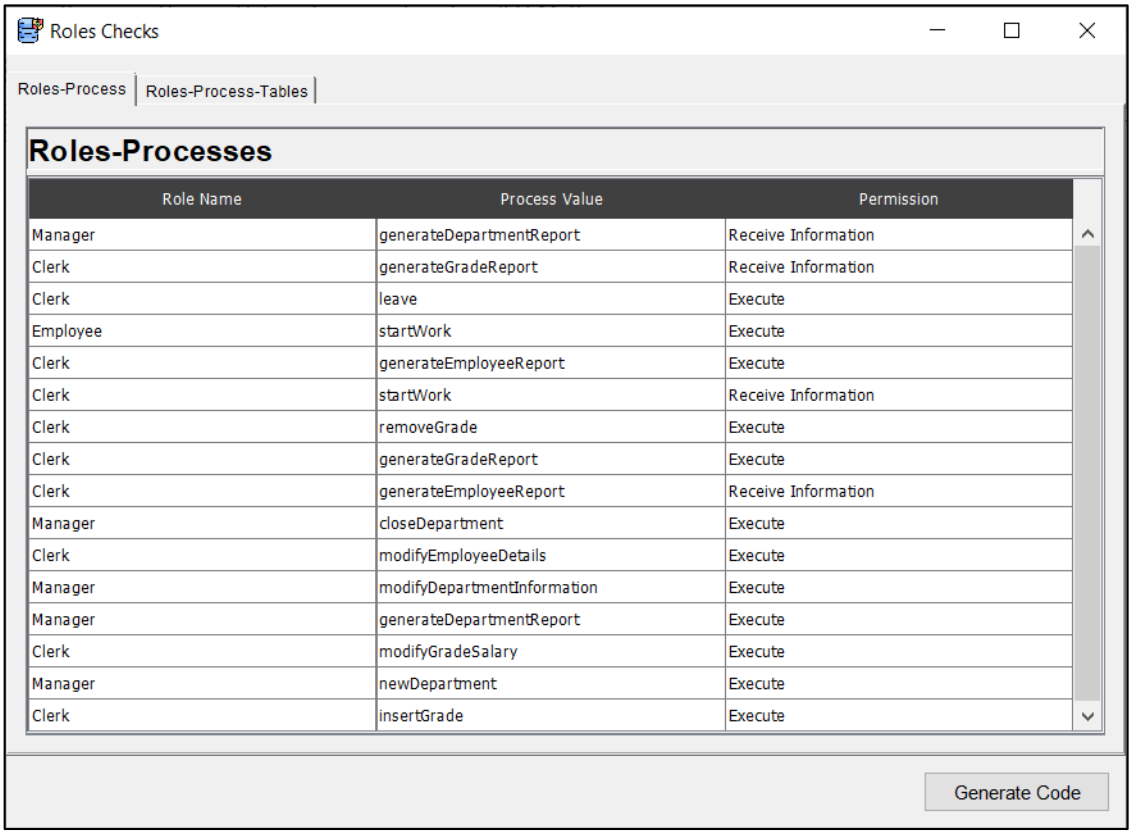


Figure C.1.11: Association of roles and processes for ‘Scott’ system

Figure C.1.11 illustrates that the role ‘Clerk’ is allowed access to a specific set of processes to within his responsibility such as ‘insertGrade’, ‘modifyEmployeeDetails’ and ‘leave’. The role ‘Manager’, on the other hand, is authorised to perform higher order functions such as creating or closing a department. In addition to execute privileges, the presence of the text 'Receive Information' in the 'perm' column can be observed. This signifies that certain roles may require information from specific processes. For instance, a clerk might necessitate receiving some output or reply when generating a report.

C.1.5 Creation of extended CRUD matrix

The subsequent operation conducted by the proposed tool involves revisiting the initial-level CRUD matrix to generate an extended 3D matrix that incorporates an additional dimension for roles. The resulting 3D CRUD matrix for the 'Scott' system is presented in Figure C.1.12 below.

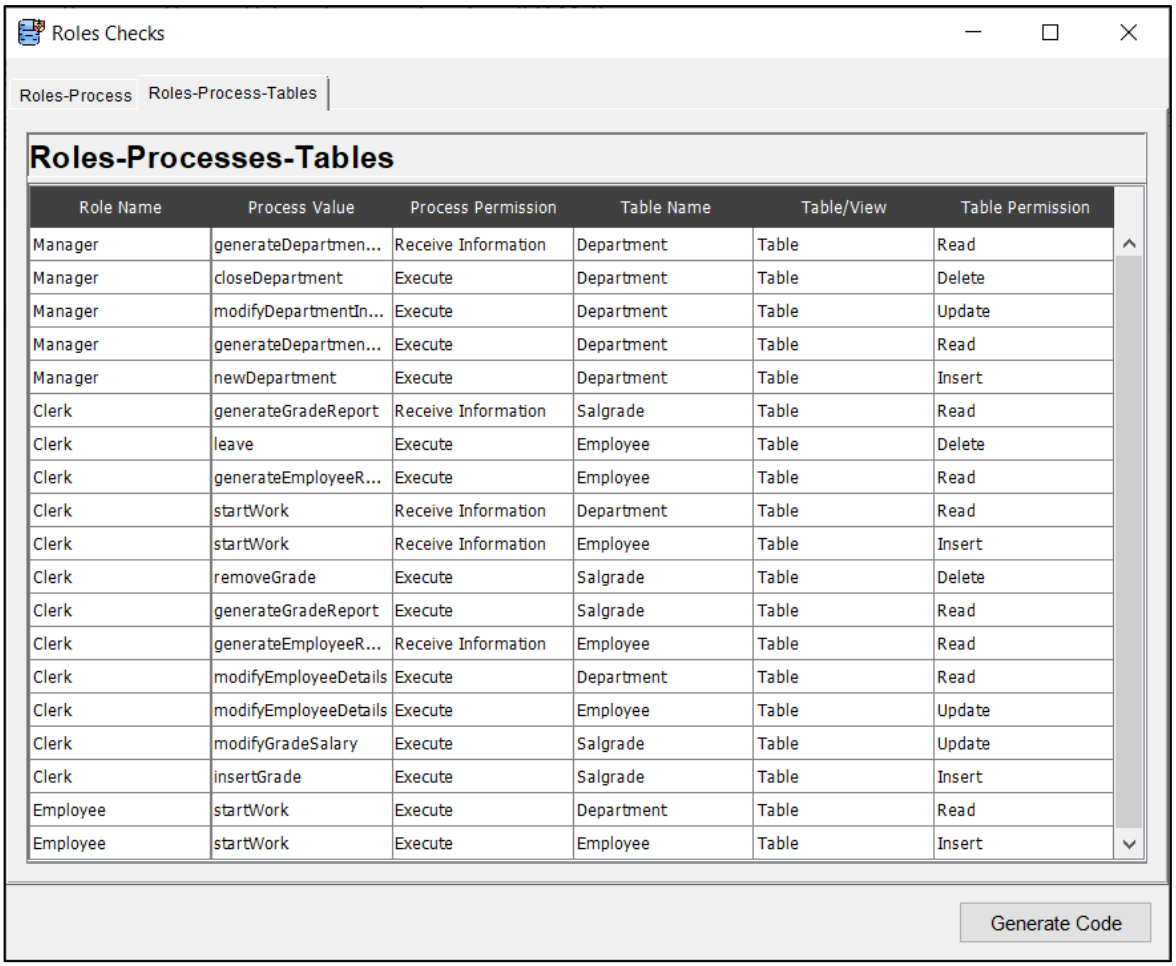


Figure C.1.12: Expanded CRUD Matrix for ‘Scott’ System

The first tuple in Figure C.1.12 explicitly indicates that the 'Manager' role has the required authorisation to execute the 'closeDepartment' process. However, this authorisation is contingent upon possessing DELETE privileges on the 'Department' table. Similarly, the second tuple indicates that the 'Manager' role is also capable of executing the 'modifyDepartmentInformation' process. However, in order to successfully perform this task, the role must have UPDATE privileges on the previously mentioned table.

C.1.6 Matrix extension to include horizontal and vertical partitioning

Upon completion of the 3D matrix, analogous to the approach adopted in the 'Banking System' case study (Chapter 7), several role partitions were formulated and integrated into the system. The file contains specifications for both horizontal and vertical partitions applied to the 'Clerk' role. The vertical partition restricts roles categorised as 'Clerk' to only accessing the 'name' and 'job' columns within the 'Employee' table. In

addition, a horizontal partition is in place that imposes row-level access restrictions within the same table for the same role, based on the values in the 'job' column. Furthermore, there is an additional horizontal partitioning requirement specific to the 'Clerk' role, which pertains to the 'Department' table and is based on the values in the 'location' column.

C.1.7 Tenant-specific specifications and generation of SQL Code

This stage involved the provisioning of tenant specific requirements and the generation of the final secure SQL code. For this case study, the requirements were as follows: separate database multi-tenant approach, 'UTF8' database encoding, 'md5' authentication, and a password expiry set for November 2 Following the input and validation of these requirements, the process concluded with the successful generation of the corresponding SQL code for a secure multi-tenant database system.

Figure C.1.13 below depicts a portion of the generated code that grants INSERT privileges to the respective roles. It can be noted that the ‘Employee’ role can insert tuples in the ‘Employee’ table. On the other hand INSERT operations on the ‘Department’ table are only allowed for managerial roles and those on 'Salgrade', to clerical employees.

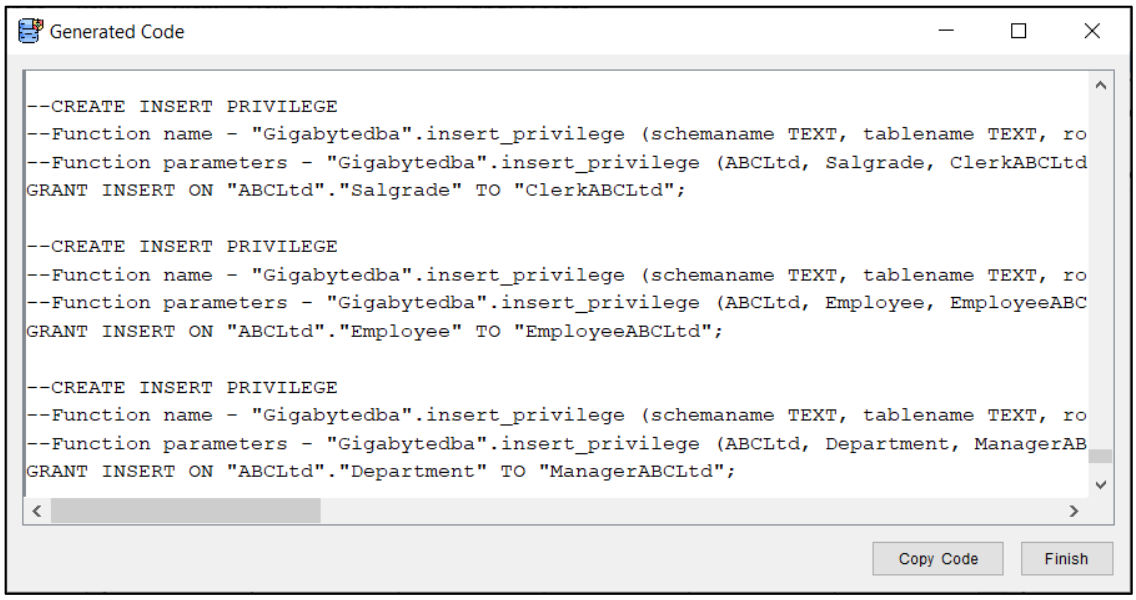


Figure C.1.13: Generated code denoting INSERT privileges

Figure C.1.14 illustrates another part of the code related to the creation of an RLS policy on the 'userconditions' table within the tenant's schema. This policy is specifically assigned to the 'Manager' role and serves the purpose of granting access to tuples in the 'userconditions' table that correspond to the assigned user (i.e., where the username matches their own).

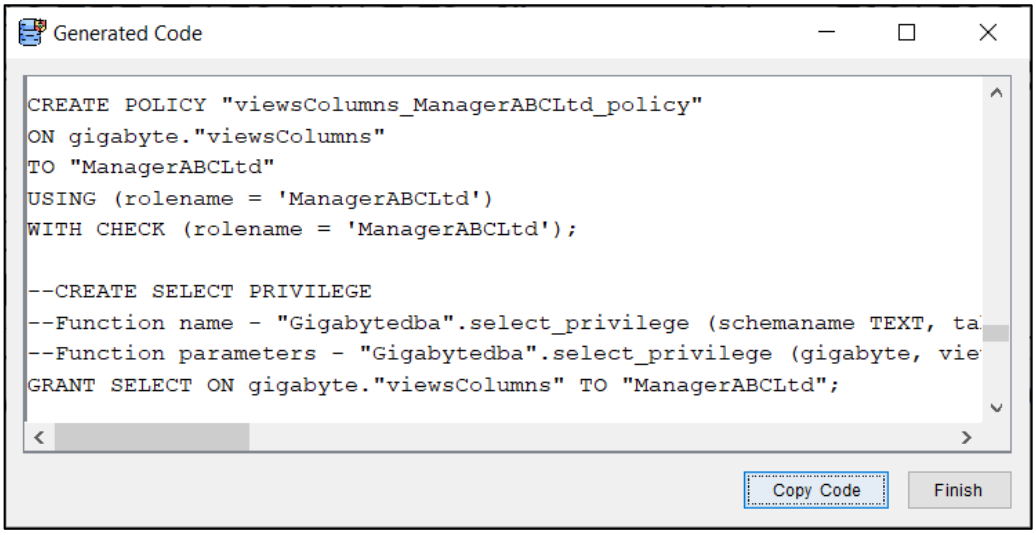


Figure C.1.14: Generated code related to RLS policy

C.1.8 User-role associations and possible partition conditions

Similar to the case study described in Chapter 7, the SQL code report generated in the prior section was executed on both local and Google Cloud platforms and the tenant environment was set up. Following successful completion of this process, the tenant DBA proceeded to the created user accounts and assigned them to suitable roles. In this particular case study, three user accounts were established: 'Laura' was assigned the role of 'Clerk', 'Pierre' was assigned the role of 'Manager', and 'Violet' was assigned the role of 'Employee'.

During the process of assigning users to roles, specific conditions were defined for horizontally partitioned roles. As mentioned in Section C.1.6, only the 'Clerk' role is subject to partitioning in this case study. Therefore, the conditions specified were related to 'Laura' (a 'Clerk'), granting her access exclusively to tuples in the 'Employees' table with the 'job' type 'Clerk' and to departments located in Sliema.

C.1.9 User login and function execution

After the previous section is completed, tenant users are allowed to log in and actively utilise the system. The credential associated with user ‘Laura’ (a ‘Clerk’) were inputted for testing purposes. Subsequently, the processes 'generateEmployeeReport' and 'insertGrade' were executed, and their outcomes were observed. Following inspection of the DFD, it is evident that the first process is associated with the 'Employee' table, while the second process is related to 'Salgrade'.

As discussed in Section C.1.6, Laura's access to the 'Employee' table is limited to the 'employeename' and 'job' columns, specifically for rows where the job is identified as 'Clerk'. Therefore, following execution of the first process, the following result was obtained as expected.

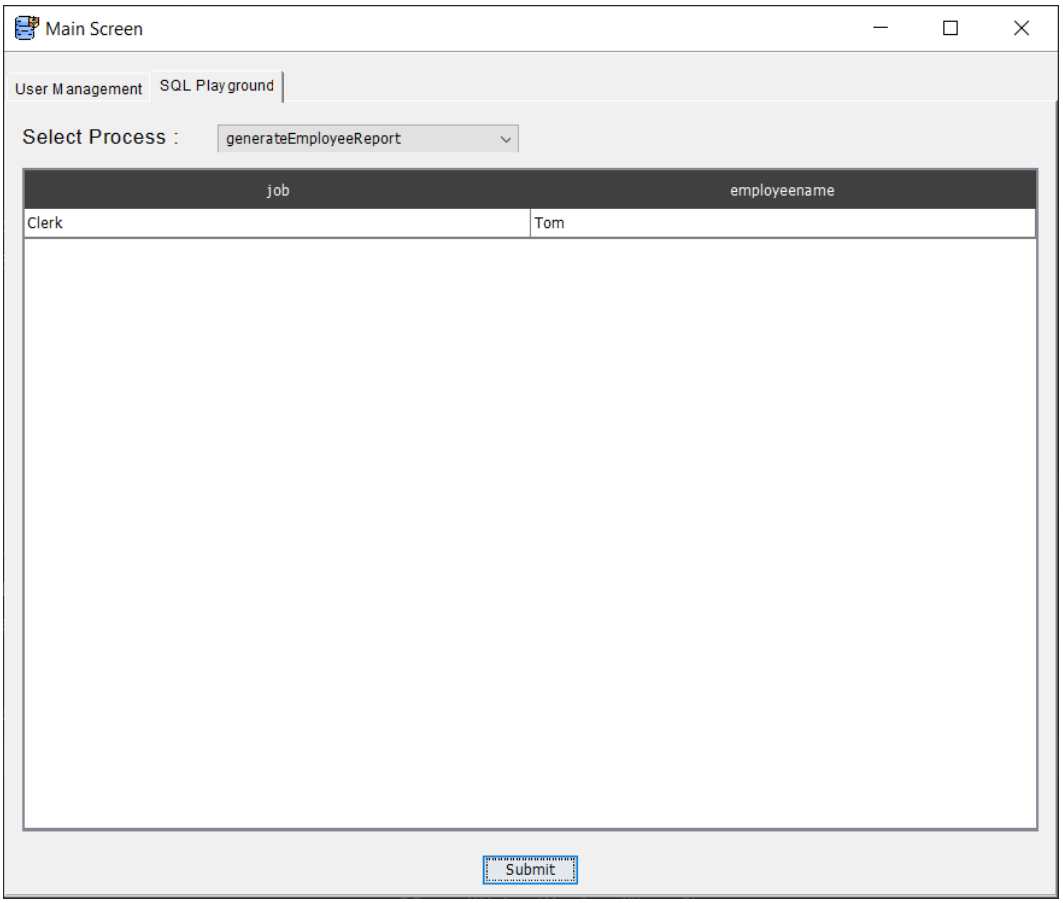


Figure C.1.15: Execution of 'generateEmployeeReport'

On the contrary, user ‘Laura’ encountered no limitations when inserting tuples into the 'Salgrade' table (by means of the ‘insertGrade’ process), as there are no partitions associated with it.

C.2 School Management System Case Study

This case study focuses on a school management system which is more complex compared to the previous case studies implemented. The system comprises six entities (two weak and four strong) that store crucial information regarding the relationships among students, lecturers, departments, units, and transcripts. The student entity holds data regarding enrolled students, while the lecturer entity captures information about the teaching staff. Additionally, the department entity encompasses details about the academic departments operating within the system. Furthermore, the unit entity stores information on academic units or courses, and the transcript entity maintains records of students' academic progress and achievements.

C.2.1 System Design Diagrams

Figure C.2.1 presents a visual representation of the table relationships in the 'School Management System' database through ERM diagram, which was supplied to the proposed tool in JSON format. Following that, the different states of each entity were depicted by constructing individual ELH diagrams for each entity. Following that, the DFD diagram displayed in Figure C.2.2 was created, and presented to the tool in JSON format.

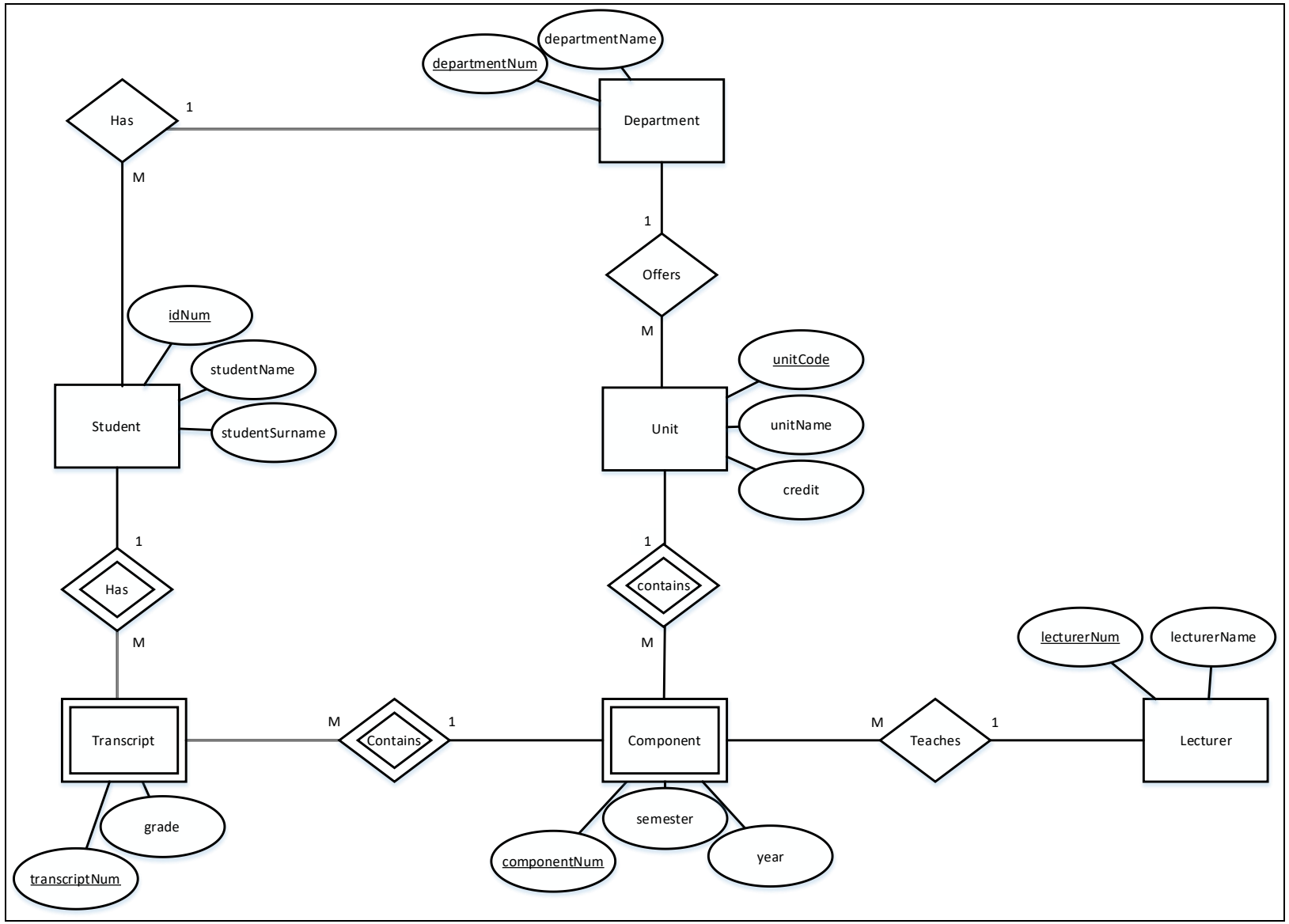


Figure C.2.1: ERM for 'School Management System'

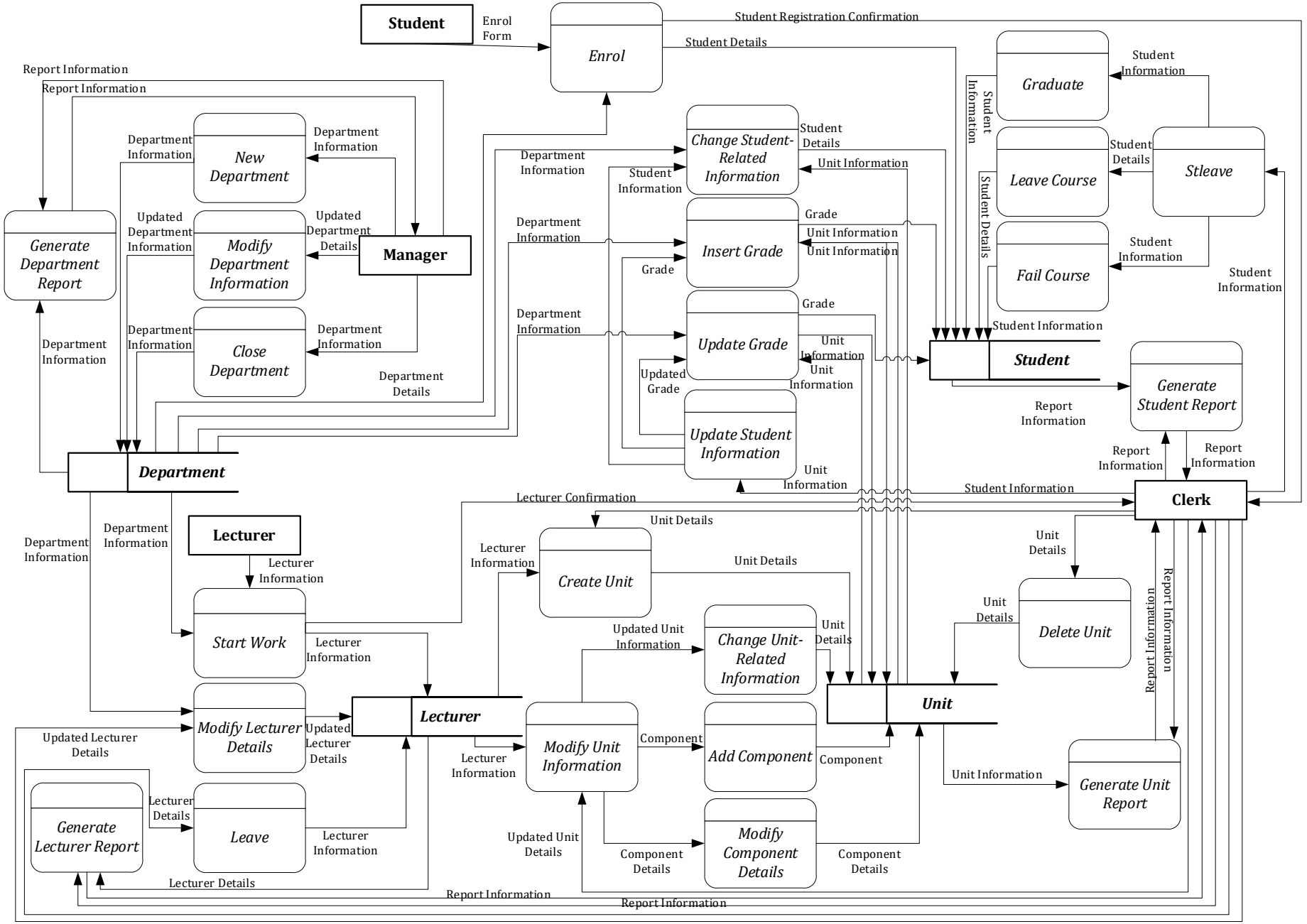


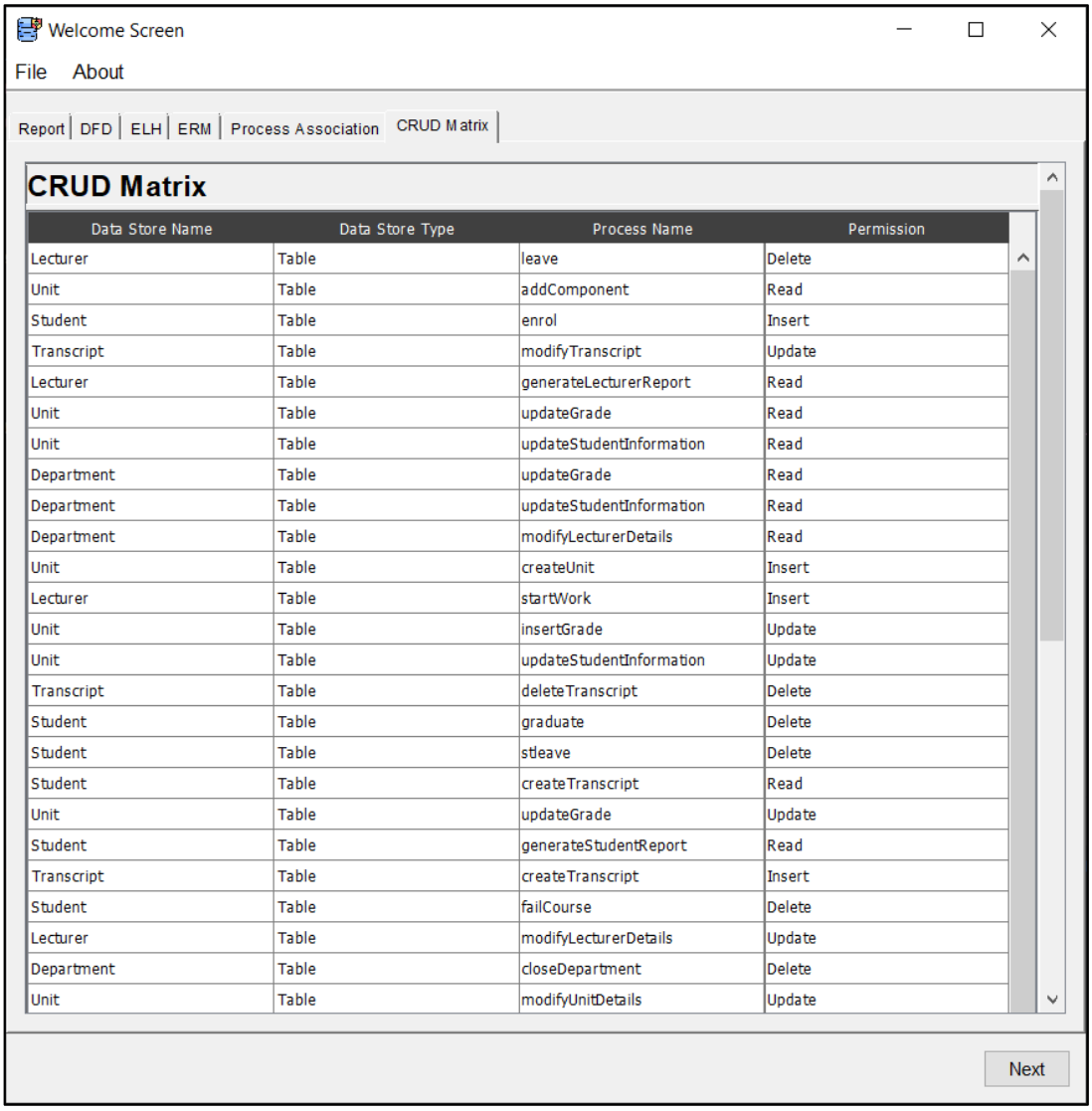
Figure C.2.2: ERM for 'School Management System'

C.2.2 Pre and Post checks

Similarly to the previous case study, the next phase of the proposed tool revolves around the conduction of a series of security checks based on the three diagrams presented. The checks are conducted on each diagram individually and then in a collective manner. The execution of these checks resulted in either a 'Good' or 'Not found' outcome indicating that the check passed or no instances of the checked database object were found in the presented systems.

C.2.3 Generation of first-level CRUD Matrix

The CRUD matrix produced for the 'School Management System' based on the presented system diagrams is depicted hereunder.



Data Store Name	Data Store Type	Process Name	Permission
Lecturer	Table	leave	Delete
Unit	Table	addComponent	Read
Student	Table	enrol	Insert
Transcript	Table	modifyTranscript	Update
Lecturer	Table	generateLecturerReport	Read
Unit	Table	updateGrade	Read
Unit	Table	updateStudentInformation	Read
Department	Table	updateGrade	Read
Department	Table	updateStudentInformation	Read
Department	Table	modifyLecturerDetails	Read
Unit	Table	createUnit	Insert
Lecturer	Table	startWork	Insert
Unit	Table	insertGrade	Update
Unit	Table	updateStudentInformation	Update
Transcript	Table	deleteTranscript	Delete
Student	Table	graduate	Delete
Student	Table	stleave	Delete
Student	Table	createTranscript	Read
Unit	Table	updateGrade	Update
Student	Table	generateStudentReport	Read
Transcript	Table	createTranscript	Insert
Student	Table	failCourse	Delete
Lecturer	Table	modifyLecturerDetails	Update
Department	Table	closeDepartment	Delete
Unit	Table	modifyUnitDetails	Update

Figure C.2.3: First- level CRUD matrix for 'School Management System'

The proposed tool proceeds by conducting a thorough evaluation through a set of checks on the developed CRUD matrix. All the obtained results consistently showed positive outcomes, serving as strong indicators that the CRUD matrix has been thoroughly verified and validated.

C.2.4 CRUD matrix expansion and vertical or horizontal partitioning

After successfully completing the checks detailed in the preceding section, the system discerns the essential roles and their corresponding processes necessary for the efficient execution of their respective responsibilities. The roles-processes matrix generated for the 'School Management System' can be viewed in Figure C.2.4 below.

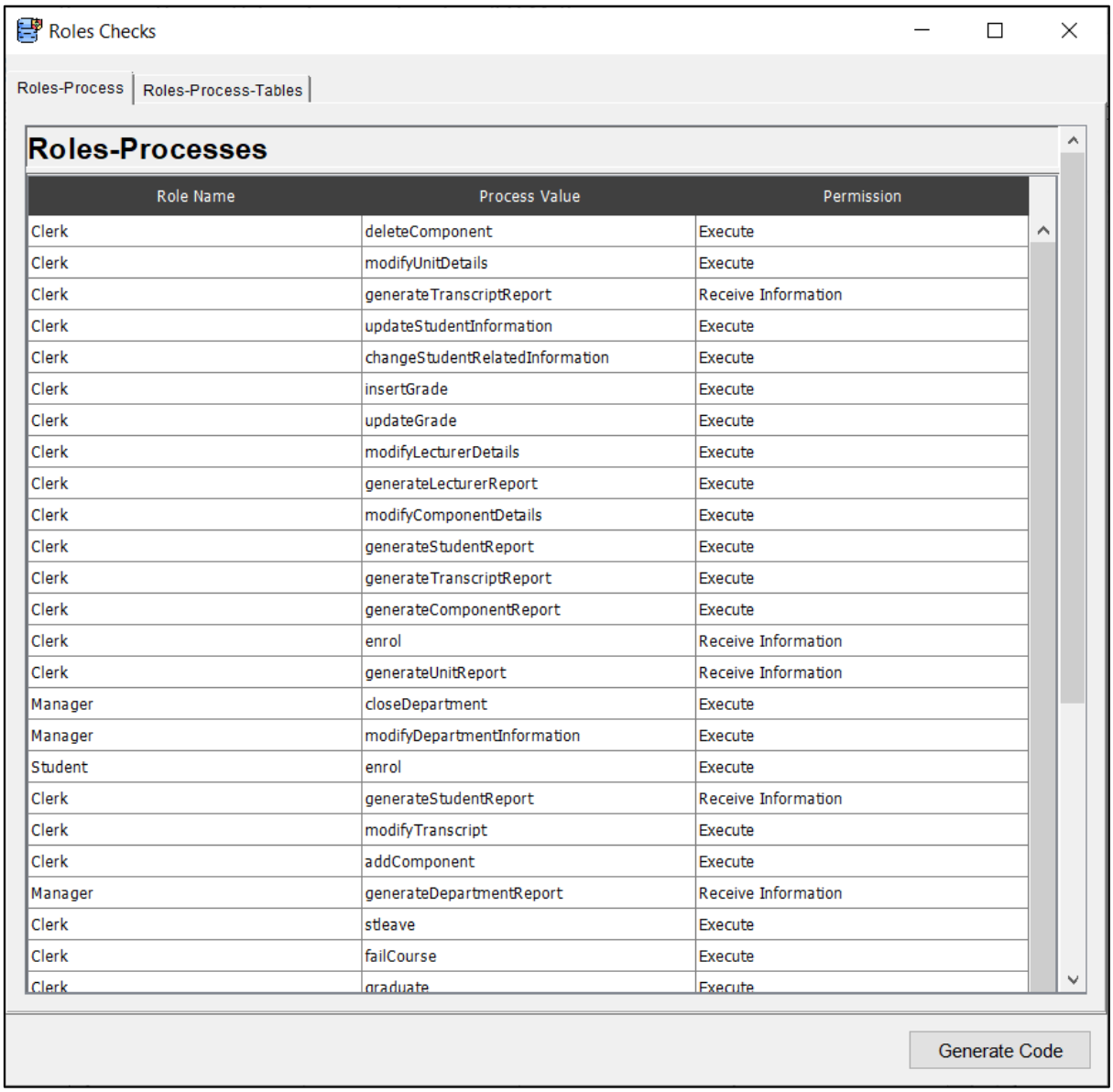


Figure C.2.4: Association of roles and processes for ‘School Management System’

Subsequently, the initial-level CRUD matrix evolved into a 3D matrix through the inclusion of an additional dimension dedicated to roles. Furthermore, in the context of this case study, specific modifications were made, including the introduction of both vertical and horizontal partitions within the 'Lecturer' role on the 'Lecturer' table. Each of these alterations were then accurately reflected in the matrix.

C.2.5 Tenant-specific requirements and generation of SQL Code

Finally, the provisioning of tenant-specific requirements and the generation of the ultimate secure SQL code were addressed. The same set of requirements employed in the previous CASE study served as the basis for this phase as well. After inputting and validating these requirements, the process reached its conclusion with the successful creation of the corresponding SQL code for a secure multi-tenant database system

Appendix D

D.1 Banking System ELHs

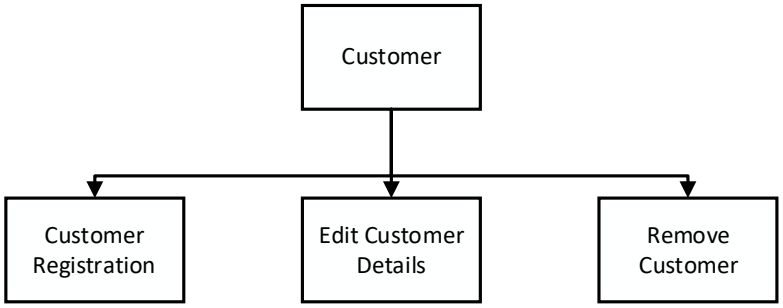


Figure D.1.1: 'Customer' ELH

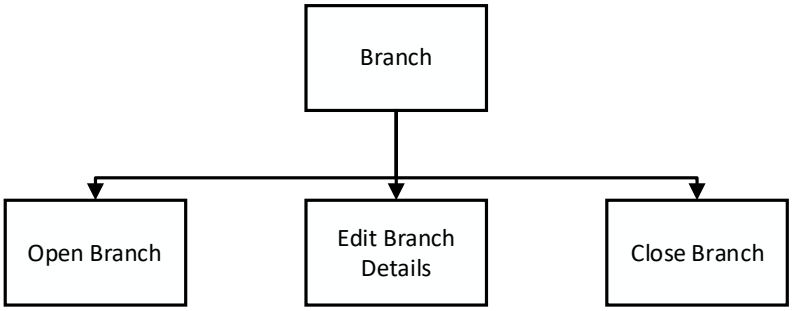


Figure D.1.2: 'Branch' ELH

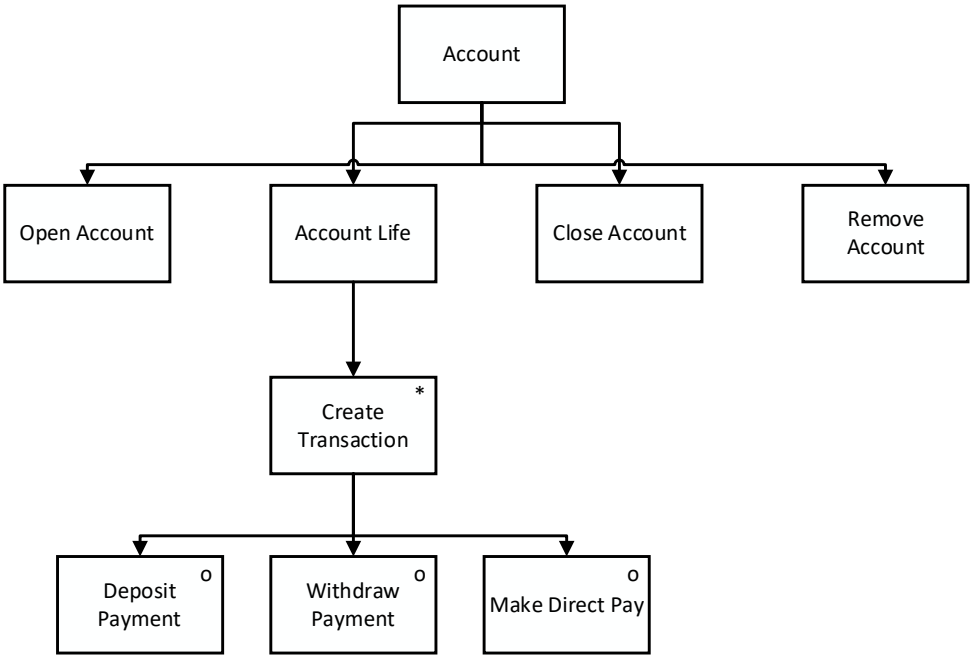


Figure D.1.3 'Account' ELH

D.2 Banking System DFD

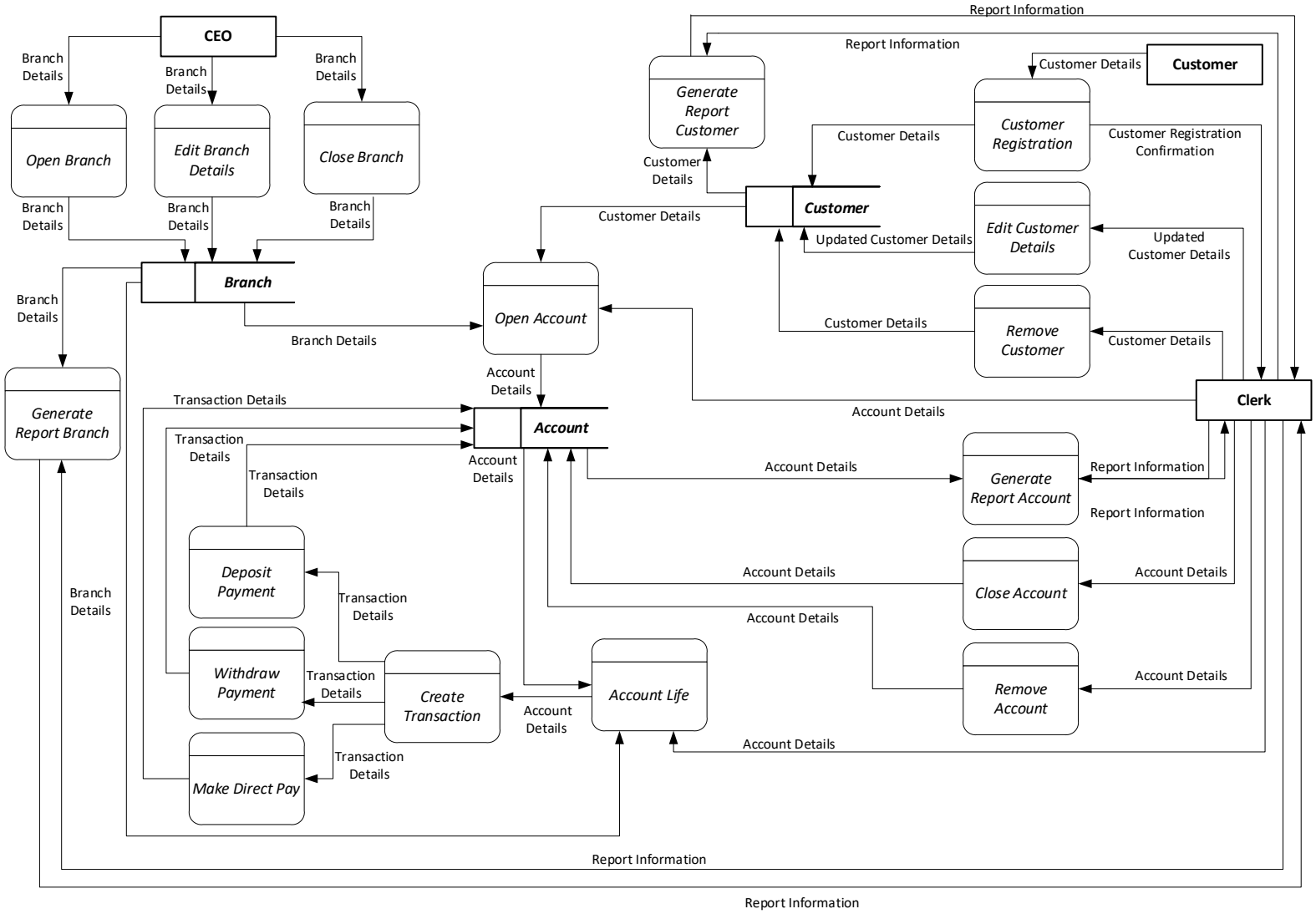


Figure D.2.1 'Banking System' DFD

D.3 Banking System Post-checks

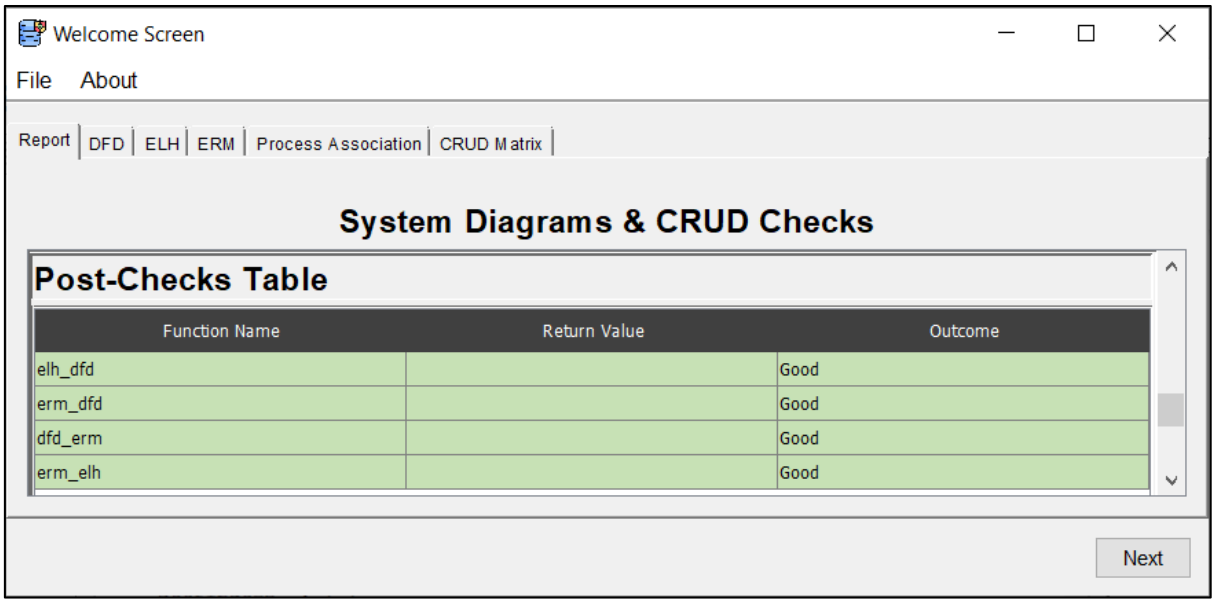


Figure D.3.1: Post checks – 'Banking System'

D.4 Banking System CRUD checks

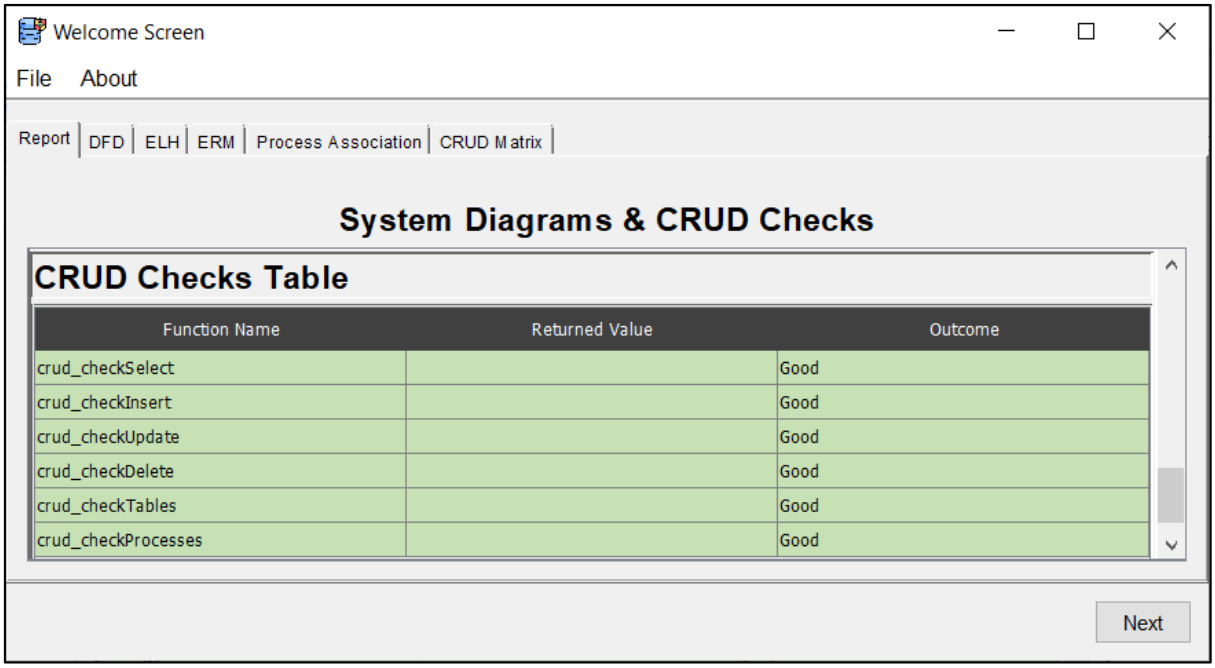


Figure D.4.1: CRUD Checks: 'Banking System'

D.5 SQL generated code – Separate database

```
CREATE ROLE bank WITH
INHERIT
NOCREATEDB
CREATEROLE
LOGIN
CONNECTION LIMIT -1
ENCRYPTED PASSWORD 'bank'
VALID UNTIL '08-03-2024'
IN ROLE tenant;

CREATE DATABASE bank WITH OWNER gigabytedba ENCODING = 'UTF8' LC_COLLATE = 'English_Malta.1252' LC_CTYPE = 'English_Malta.1252' CONNECTION LIMIT = -1;

REVOKE ALL ON DATABASE bank FROM public;

GRANT CONNECT ON DATABASE bank TO bank WITH GRANT OPTION;

CREATE EXTENSION IF NOT EXISTS pgcrypto;

CREATE SCHEMA IF NOT EXISTS spschema AUTHORIZATION gigabytedba;

CREATE TABLE IF NOT EXISTS spschema.roles(rolename varchar, partitioned bool,
CONSTRAINT pkroles PRIMARY KEY(rolename));

CREATE TABLE IF NOT EXISTS spschema.views(viewname text, tablename text, canInsert bool, canUpdate bool, canDelete bool, rolename varchar,
CONSTRAINT pkviews PRIMARY KEY(viewname, rolename),
CONSTRAINT views_rolename_rolenamefk FOREIGN KEY(rolename) REFERENCES spschema.roles(rolename) ON DELETE CASCADE);

CREATE TABLE IF NOT EXISTS spschema."viewsColumns"(vcSn serial, viewname text, rolename text, columnname text, horizpart bool,
CONSTRAINT pkviewsColumns PRIMARY KEY(vcSn),
CONSTRAINT "viewsColumns_viewnamerolename_viewnamerolenamefk" FOREIGN KEY(viewname,rolename) REFERENCES spschema.views(viewname,rolename) ON DELETE CASCADE);

CREATE TABLE IF NOT EXISTS spschema.tenant(sn serial, tenant_name text, tenant_pswd text,
CONSTRAINT pktenant PRIMARY KEY(sn));

CREATE TABLE IF NOT EXISTS spschema.log_activity(logid serial, schemaname text, relationname text, username text, currenttime time with time zone, activitytime time with time zone, clientaddress inet, clientport int, queryPerformed text, typeofquery text, currentdata JSON, altereddata JSON,
CONSTRAINT pklog_activity PRIMARY KEY(logid));
ALTER TABLE spschema.log_activity ALTER COLUMN schemaname SET NOT NULL;
ALTER TABLE spschema.log_activity ALTER COLUMN relationname SET NOT NULL;
ALTER TABLE spschema.log_activity ALTER COLUMN currenttime SET NOT NULL;
ALTER TABLE spschema.log_activity ALTER COLUMN activitytime SET NOT NULL;
ALTER TABLE spschema.log_activity ALTER COLUMN typeofquery SET NOT NULL;

CREATE OR REPLACE FUNCTION spschema.accessibility(spschemaname text, tenantschema text)
RETURNS JSON AS $$
DECLARE
r text;
ro RECORD;
ta RECORD;
j integer;
priv text;
privArr text[];
retVal boolean;
retArray text[];
rArr text[];
res text;
fnctResult JSON;

BEGIN
```

```
59.
60. j = 0;
61. privArr := '{''SELECT'', ''INSERT'', ''UPDATE'', ''DELETE''}';
62. EXECUTE format('SELECT ARRAY_AGG(rolename) FROM %I.roles', spschema) INTO rArr;
63. FOR ta IN
64. (SELECT table_name, table_schema
65. FROM information_schema.tables
66. WHERE table_schema = tenantschema
67. AND table_name != 'user'
68. AND table_name != 'usercondition')
69. LOOP
70. FOREACH ro IN ARRAY rArr
71. LOOP
72. FOREACH priv in ARRAY privArr
73. LOOP
74. EXECUTE FORMAT('SELECT has_table_privilege (%L, %s, %L)', ro.rolename, (ta.table_schema
    ||'.'|| ta.table_name), priv) INTO retVal;
75. IF !(retVal) THEN
76. retArray[j] := retArray[j] || ta.table_name;
77. j = j+1;
78. END IF;
79. END LOOP;
80. END LOOP;
81. END LOOP;
82. IF retArray = '{}' THEN
83.   RAISE NOTICE 'Check accessibility: Correct';
84.   res = 'Good';
85.   ELSE
86.   FOREACH r IN ARRAY retArray
87.   LOOP
88.   RAISE NOTICE 'Check accessibility: Entity %, cannot be accessed', r;
89.   END LOOP;
90.   res = 'Warning';
91.   END IF;
92.   fnctResult = '{"acessibility", array_to_string(retArray, chr(10)), res}';
93.   return fnctResult;
94.
95.   END;
96.   $$ LANGUAGE plpgsql;
97.
98.
99.   CREATE OR REPLACE FUNCTION spschema.reachability(spschemaname text, tenantschema text)
100.  RETURNS JSON AS $$
101.  DECLARE
102.  r text;
103.  ro RECORD;
104.  ta RECORD;
105.  j integer;
106.  priv text;
107.  privArr text[];
108.  retVal boolean;
109.  retArray text[];
110.  rArr text[];
111.  proArr text[];
112.  pa text;
113.  res text;
114.  fnctResult JSON;
115.
116.  BEGIN
117.
118.  j = 0;
119.  EXECUTE format('SELECT ARRAY_AGG(rolename) FROM %I.roles', spschema) INTO rArr;
120.  FOR ta IN
121.  (SELECT table_name, table_schema
122.  FROM information_schema.tables
123.  WHERE table_schema = tenantschema
124.  AND table_name != 'user'
125.  AND table_name != 'usercondition')
126.  LOOP
127.  EXECUTE format('SELECT array_agg(proname)
```

```
128.   FROM pg_proc
129.   WHERE prosrc LIKE ''% %s %''', ta) INTO proArr;
130.   FOREACH pa IN ARRAY proArray
131.   LOOP
132.   FOREACH ro IN ARRAY rArr
133.   LOOP
134.   EXECUTE FORMAT('SELECT has_function_privilege(%L, %L, ''EXECUTE'');', ro, pa) INTO ret-
       Val;
135.   IF !(retVal) THEN
136.   retArray[j] := retArray[j] || ta.table_name;
137.   j = j+1;
138.   END IF;
139.   END LOOP;
140.   END LOOP;
141.   END LOOP;
142.   IF retArray = '{}' THEN
143.   RAISE NOTICE 'Check reachability: Correct';
144.   res = 'Good';
145.   ELSE
146.   FOREACH r IN ARRAY retArray
147.   LOOP
148.   RAISE NOTICE 'Check reachability: Entity %, is not reachable', r;
149.   END LOOP;
150.   res = 'Warning';
151.   END IF;
152.   fnctResult = '{"reachability", array_to_string(retArray, chr(10)), res}';
153.   RETURN fnctResult;
154.
155.   END;
156.   $$ LANGUAGE plpgsql;
157.
158.
159.   CREATE OR REPLACE FUNCTION spschema."redundantRoles"(spschemaname text, tenantschema
       text)
160.   RETURNS JSON AS $$
161.   DECLARE
162.   r text;
163.   j integer;
164.   retVal boolean;
165.   retArray text[];
166.   rArr text[];
167.   res text;
168.   fnctResult JSON;
169.
170.   BEGIN
171.
172.   j = 0;
173.   EXECUTE format('SELECT ARRAY_AGG(rolename) FROM %I.roles', spschema) INTO rArr;
174.   FOREACH r IN ARRAY rArr
175.   LOOP
176.   EXECUTE format ('SELECT EXISTS (SELECT username FROM %I.user WHERE rolename = %L)',
       spschema, r) INTO retVal;
177.   IF !(retVal) THEN
178.   retArray[j] := retArray[j] || r;
179.   j = j+1;
180.   END IF;
181.   END LOOP;
182.   IF retArray = '{}' THEN
183.   RAISE NOTICE 'Check redundant roles: Correct';
184.   res = 'Good';
185.   ELSE
186.   FOREACH r IN ARRAY retArray
187.   LOOP
188.   RAISE NOTICE 'Check redundant roles: Role %, is redundant', r;
189.   END LOOP;
190.   res = 'Warning';
191.   END IF;
192.   fnctResult = '{"redundantRoles", array_to_string(retArray, chr(10)), res}';
193.   RETURN fnctResult;
194.
```

```
195.  END;
196.  $$ LANGUAGE plpgsql;
197.
198.
199.  CREATE OR REPLACE FUNCTION spschema.inference(spschemaname text, tenantschema text)
200.  RETURNS JSON AS $$
201.  DECLARE
202.  r text;
203.  ro RECORD;
204.  ta RECORD;
205.  j integer;
206.  priv text;
207.  retVal boolean;
208.  retArray text[];
209.  rArr text[];
210.  selPriv boolean;
211.  insertPriv boolean;
212.  deletePriv boolean;
213.  res text;
214.  fnctResult JSON;
215.
216.  BEGIN
217.
218.  j = 0;
219.  EXECUTE format('SELECT ARRAY_AGG(rolename) FROM %I.roles', spschema) INTO rArr;
220.  FOR ta IN
221.  (SELECT table_name, table_schema
222.  FROM information_schema.tables
223.  WHERE table_schema = tenantschema
224.  AND table_name != 'user'
225.  AND table_name != 'usercondition')
226.  LOOP
227.  FOREACH ro IN ARRAY rArr
228.  LOOP
229.  EXECUTE FORMAT('SELECT has_table_privilege (%L, %s, ''SELECT'')', ro.rolename, (ta.ta-
      ble_schema ||'.'|| ta.table_name)) INTO selPriv;
230.  IF !(selPriv) THEN
231.  EXECUTE FORMAT('SELECT has_table_privilege (%L, %s, ''INSERT'')', ro.rolename, (ta.ta-
      ble_schema ||'.'|| ta.table_name)) INTO insertPriv;
232.  EXECUTE FORMAT('SELECT has_table_privilege (%L, %s, ''DELETE'')', ro.rolename, (ta.ta-
      ble_schema ||'.'|| ta.table_name)) INTO deletePriv;
233.  IF (insertPriv OR deletePriv) THEN
234.  retArray[j] := retArray[j] || ta.table_name;
235.  j = j+1;
236.  END IF;
237.  END IF;
238.  END LOOP;
239.  END LOOP;
240.  IF retArray = '{}' THEN
241.  RAISE NOTICE 'Check inference: Correct';
242.  res = 'Good';
243.  ELSE
244.  FOREACH r IN ARRAY retArray
245.  LOOP
246.  RAISE NOTICE 'Check inference: Entity %, cannot be accessed', r;
247.  END LOOP;
248.  res = 'Warning';
249.  END IF;
250.  fnctResult = '{"inference", array_to_string(retArray, chr(10)), res}';
251.  RETURN fnctResult;
252.
253.  END;
254.  $$ LANGUAGE plpgsql;
255.
256.
257.  CREATE OR REPLACE FUNCTION spschema.diff_json(firstJsonValue JSON, secondJsonValue JSON)
258.  RETURNS JSON AS $$
259.  DECLARE
260.  outcome jsonb;
261.  jsonElem RECORD;
```

```
262.
263.   BEGIN
264.
265.   outcome = firstJsonValue::jsonb;
266.   FOR jsonElem IN
267.   SELECT * FROM json_each(secondJsonValue)
268.   LOOP
269.   IF outcome @> jsonb_build_object(jsonElem.key, jsonElem.value)
270.   THEN outcome = outcome - jsonElem.key;
271.    ELSIF outcome ? jsonElem.key THEN CONTINUE;
272.   ELSE
273.   outcome = outcome || json_build_object(jsonElem.key,'null');
274.   END IF;
275.   END LOOP;
276.   RETURN outcome::json;
277.
278.   END;
279.   $$ LANGUAGE plpgsql;
280.
281.
282.   CREATE OR REPLACE FUNCTION spschema.log_trigger()
283.   RETURNS trigger AS $$
284.   DECLARE
285.   logrow spschema.log_activity;
286.   oldData JSON;
287.   newData JSON;
288.   modifiedColumns JSON;
289.
290.   BEGIN
291.
292.   logrow = ROW(nextval('spschema.log_activity_logid_seq'::regclass), TG_TABLE_SCHEMA::text,
      TG_TABLE_NAME::text, session_user::text, current_timestamp, statement_timestamp(),
      inet_client_addr(), inet_client_port(), current_query(), TG_OP, NULL, NULL);
293.   IF TG_ARGV[0]::boolean = 'f'::boolean THEN
294.   logrow.queryperformed = NULL;
295.   END IF;
296.   IF TG_ARGV[1] IS NOT NULL THEN
297.   modifiedColumns = TG_ARGV[1]::json;
298.   END IF;
299.   IF (TG_LEVEL = 'ROW' AND TG_OP = 'INSERT') THEN
300.   logrow.currentdata = spschema.diff_json(row_to_json(NEW.*), modifiedColumns);
301.   ELSIF (TG_LEVEL = 'ROW' AND TG_OP = 'UPDATE') THEN
302.   logrow.currentdata = spschema.diff_json(row_to_json(OLD.*), modifiedColumns);
303.   logrow.altereddata = spschema.diff_json(spschema.diff_json(row_to_json(NEW.*),
       logrow.currentdata), modifiedColumns);
304.   ELSIF (TG_LEVEL = 'ROW' AND TG_OP = 'DELETE') THEN
305.   logrow.currentdata = spschema.diff_json(row_to_json(OLD.*), modifiedColumns);
306.   ELSE
307.   RAISE EXCEPTION 'Operation performed is not stored in audit file';
308.   RETURN NULL;
309.   END IF;
310.   INSERT INTO spschema.log_activity VALUES (logrow.*);
311.   RETURN NULL;
312.
313.   END;
314.   $$ LANGUAGE plpgsql;
315.   CREATE TRIGGER trigger_spschema_roles
316.   AFTER INSERT OR DELETE OR UPDATE
317.   ON spschema.roles FOR EACH ROW
318.   EXECUTE FUNCTION spschema.log_trigger();
319.
320.   CREATE TRIGGER trigger_spschema_views
321.   AFTER INSERT OR DELETE OR UPDATE
322.   ON spschema.views FOR EACH ROW
323.   EXECUTE FUNCTION spschema.log_trigger();
324.
325.   CREATE TRIGGER "trigger_spschema_viewsColumns"
326.   AFTER INSERT OR DELETE OR UPDATE
327.   ON spschema."viewsColumns" FOR EACH ROW
328.   EXECUTE FUNCTION spschema.log_trigger();
```

```
329.
330.    CREATE TRIGGER trigger_spschema_tenant
331.    AFTER INSERT OR DELETE OR UPDATE
332.    ON spschema.tenant FOR EACH ROW
333.    EXECUTE FUNCTION spschema.log_trigger();
334.
335.    REVOKE ALL ON SCHEMA spschema FROM public;
336.    REVOKE ALL ON ALL TABLES IN SCHEMA spschema FROM public;
337.    REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA spschema FROM public;
338.    REVOKE ALL ON SCHEMA spschema FROM bank;
339.    GRANT USAGE ON SCHEMA spschema TO bank;
340.    REVOKE ALL ON ALL TABLES IN SCHEMA spschema FROM bank;
341.    REVOKE ALL ON ALL FUNCTIONS IN SCHEMA spschema FROM bank;
342.    ALTER TABLE spschema.roles ENABLE ROW LEVEL SECURITY;
343.
344.    CREATE POLICY roles_bank_policy
345.    ON spschema.roles
346.    TO bank
347.    USING (tenantid = 4)
348.    WITH CHECK (tenantid = 4);
349.
350.    ALTER TABLE spschema.views ENABLE ROW LEVEL SECURITY;
351.
352.    CREATE POLICY views_bank_policy
353.    ON spschema.views
354.    TO bank
355.    USING (true);
356.
357.    ALTER TABLE spschema."viewsColumns" ENABLE ROW LEVEL SECURITY;
358.
359.    CREATE POLICY "viewsColumns_bank_policy"
360.    ON spschema."viewsColumns"
361.    TO bank
362.    USING (true);
363.
364.    GRANT SELECT ON ALL TABLES IN SCHEMA spschema TO bank;
365.
366.    CREATE SCHEMA IF NOT EXISTS bank AUTHORIZATION gigabytedba;
367.
368.    CREATE TABLE IF NOT EXISTS bank."user"(username varchar, pswd varchar, enbl bool, email
       varchar, created_at timestamp, rolename varchar,
369.    CONSTRAINT pkuser PRIMARY KEY(username, rolename),
370.    CONSTRAINT user_rolename_rolenamefk FOREIGN KEY(rolename) REFERENCES
       spschema.roles(rolename) ON DELETE CASCADE);
371.
372.    CREATE TABLE IF NOT EXISTS bank.userconditions(sn serial, username varchar, rolename var
       char, ucviewname text, op text, val text,
373.    CONSTRAINT pkuserconditions PRIMARY KEY(sn),
374.    CONSTRAINT userconditions_usernamerolename_usernamerolenamefk FOREIGN
       KEY(username,rolename) REFERENCES bank.user(username,rolename) ON DELETE CASCADE,
375.    CONSTRAINT userconditions_rolenameucviewname_rolenameviewnamefk FOREIGN
       KEY(rolename,ucviewname) REFERENCES spschema.views(rolename,viewname) ON DELETE CASCADE);
376.
377.    CREATE OR REPLACE FUNCTION bank.insert_user(rn text, usname text, pass text, em text)
378.    RETURNS void AS $$
379.    DECLARE
380.    rolExists bool;
381.
382.    BEGIN
383.
384.    EXECUTE FORMAT('SELECT EXISTS(SELECT * FROM spschema.roles WHERE rolename = %L andtiq )',
       rn) INTO rolExists;
385.    IF rolExists THEN
386.    EXECUTE FORMAT('INSERT INTO bank.user(username, pswd, enbl, email, created_at, rolename
       ) VALUES (%L, %L, %s, %L, %L, %L )', usname, crypt(pass, gen_salt('bf')), 'true', em,
       (SELECT NOW()), rn  );
387.    EXECUTE FORMAT('SELECT * FROM bank.user_creation(%L, %s, %s, %s, %L, %L)', usname,
       'false', 'false', -1, crypt(pass, gen_salt('bf')), rn);
388.    ELSE
389.    RAISE EXCEPTION 'Role not found';
```

```
390.    END IF;
391.
392.    END;
393.    $$ LANGUAGE plpgsql;
394.
395.
396.    CREATE OR REPLACE FUNCTION bank.insert_csvdata(schemaname text, tablename text,
       csvfilepath text)
397.    RETURNS text AS $$
398.
399.    BEGIN
400.
401.    EXECUTE FORMAT('COPY %I.%I FROM %L WITH (FORMAT csv)', schemaname, tablename,
       csvfilepath);
402.    EXCEPTION WHEN undefined_file THEN
403.    RAISE EXCEPTION 'File not found';
404.    WHEN others THEN
405.    RAISE EXCEPTION 'An error occured whilst copying csv data from file to table';
406.
407.    END;
408.    $$ LANGUAGE plpgsql;
409.
410.
411.    CREATE OR REPLACE FUNCTION bank.user_creation(username text, databasecreation bool,
       rolecreation bool, connlimit int, pswd text, associatedrole text, databasenm text)
412.    RETURNS void AS $$
413.    DECLARE
414.    dbCreation text;
415.    rolCreation text;
416.    j int;
417.
418.    BEGIN
419.
420.    j = 0;
421.    IF databaseCreation THEN
422.    dbCreation = 'CREATEDB';
423.    ELSE dbCreation = 'NOCREATEDB';
424.    END IF;
425.    IF roleCreation THEN
426.    rolCreation = 'CREATEROLE';
427.    ELSE rolCreation = 'NOCREATEROLE';
428.    END IF;
429.    EXECUTE format ('CREATE ROLE %I WITH INHERIT %s %s LOGIN CONNECTION LIMIT %s ENCRYPTED
       PASSWORD %L VALID UNTIL 30/10/2023 IN ROLE %I;', username, dbCreation, rolCreation,
       connLimit, pswd, associatedRole);
430.    EXECUTE format ('GRANT CONNECT ON DATABASE %s TO %I;', databasenm, username);
431.
432.    END;
433.    $$ LANGUAGE plpgsql;
434.
435.
436.    CREATE OR REPLACE FUNCTION bank.update_user(currentusname text, currentrn text, newusname
       text, newpass text, newem text)
437.    RETURNS void AS $$
438.    DECLARE
439.    userExists bool;
440.
441.    BEGIN
442.
443.    EXECUTE FORMAT('SELECT EXISTS(SELECT * FROM bank.user WHERE username = %L and rolename =
       %L andtiq )', currentusname, currentrn) INTO userExists;
444.    IF userExists THEN
445.    EXECUTE FORMAT('UPDATE bank.user SET pswd = %L, email = %L WHERE username = %L AND
       rolename = %L andtiq ', crypt(newpass, gen_salt('bf')), newem, currentusname, currentrn);
446.    ELSE RAISE EXCEPTION 'User not found';
447.    END IF;
448.
449.    END;
450.    $$ LANGUAGE plpgsql;
451.
```

```
452.
453.   CREATE OR REPLACE FUNCTION bank.delete_user(usname text, rn text)
454.   RETURNS void AS $$
455.   DECLARE
456.   userExists bool;
457.
458.   BEGIN
459.
460.   EXECUTE FORMAT('SELECT EXISTS(SELECT * FROM bank.user WHERE username = %L and rolename =
      %L)', usname, rn) INTO userExists;
461.   IF userExists THEN
462.   EXECUTE FORMAT('DELETE FROM bank.user WHERE username = %L AND rolename = %L andtiq ', us-
      name, rn);
463.   EXECUTE FORMAT('SELECT * FROM bank.user_deletion(%L)', usname);
464.   ELSE
465.   RAISE EXCEPTION 'No such user found';
466.   END IF;
467.
468.   END;
469.   $$ LANGUAGE plpgsql;
470.
471.
472.   CREATE OR REPLACE FUNCTION bank.user_deletion(usname text)
473.   RETURNS void AS $$
474.
475.   BEGIN
476.
477.   EXECUTE format ('DROP ROLE %I;', usname);
478.
479.   END;
480.   $$ LANGUAGE plpgsql;
481.
482.
483.   CREATE OR REPLACE FUNCTION bank."delete_userCondition"(ssn int)
484.   RETURNS void AS $$
485.   DECLARE
486.   userCondExists bool;
487.
488.   BEGIN
489.
490.   EXECUTE FORMAT('SELECT EXISTS(SELECT * FROM bank.userconditions WHERE sn = %s)', ssn)
      INTO userCondExists;
491.   IF userCondExists THEN
492.   EXECUTE FORMAT('DELETE FROM bank.userconditions WHERE sn = %s', ssn);
493.   ELSE
494.   RAISE EXCEPTION 'No such user condition found';
495.   END IF;
496.
497.   END;
498.   $$ LANGUAGE plpgsql;
499.
500.
501.   CREATE OR REPLACE FUNCTION bank."getRoles"()
502.   RETURNS text[] AS $$
503.   DECLARE
504.   rolArray text[];
505.
506.   BEGIN
507.
508.   rolArray := (SELECT ARRAY_AGG(rolename) FROM gigabyte.roles  );
509.   IF rolArray IS NULL THEN
510.   RAISE EXCEPTION 'No roles found';
511.   ELSE RETURN rolArray;
512.   END IF;
513.
514.   END;
515.   $$ LANGUAGE plpgsql;
516.
517.
518.   CREATE OR REPLACE FUNCTION bank.enable_tenantusers(usrname text, rolename text)
```

```
519.   RETURNS void AS $$
520.
521.   BEGIN
522.
523.   IF EXISTS(SELECT FORMAT ('SELECT * FROM bank.users WHERE username = %L AND rolename = %L
       andtiq '), usrname, (rlname || tenantname)) THEN
524.   EXECUTE format('UPDATE bank.users SET enbl = true WHERE username = %L AND rolename = %L
       andtiq ;', usrname, (rlname || tenantname));
525.   RAISE NOTICE 'Tenant user % enabled', usrname;
526.   ELSE RAISE EXCEPTION 'Tenant user does not exist';
527.   END IF;
528.
529.   END;
530.   $$ LANGUAGE plpgsql;
531.
532.
533.   CREATE OR REPLACE FUNCTION bank.disable_tenantusers(usrname text, rolname text)
534.   RETURNS void AS $$
535.
536.   BEGIN
537.
538.   IF EXISTS(SELECT FORMAT ('SELECT * FROM bank.users WHERE username = %L AND rolename = %L
        andtiq '), usrname, (rlname || tenantname)) THEN
539.   EXECUTE format('UPDATE bank.users SET enbl = false WHERE username = %L AND rolename = %L
       andtiq ;', usrname, (rlname || tenantname));
540.   RAISE NOTICE 'Tenant user % disabled', usrname;
541.   ELSE RAISE EXCEPTION 'Tenant user does not exist';
542.   END IF;
543.
544.   END;
545.   $$ LANGUAGE plpgsql;
546.
547.
548.   CREATE OR REPLACE FUNCTION bank."getProcesses"(username text, tenantname text)
549.   RETURNS text[] AS $$
550.   DECLARE
551.   procArray text[];
552.   exist boolean;
553.
554.   BEGIN
555.
556.   EXECUTE FORMAT('SELECT EXISTS (SELECT tenant_name FROM gigabyte.tenant WHERE tenant_name
       = %L)', username) INTO exist;
557.   IF exist THEN
558.   EXECUTE FORMAT('SELECT ARRAY_AGG(p.proname) FROM pg_proc p, pg_namespace n WHERE n.oid =
       p.pronamespace AND has_function_privilege(%L, p.oid, ''execute'') AND n.nspname = %L AND
       n.nspname = %L AND p.proname != ''getProcesses'' AND p.proname != ''getProcessParams'' ',
       username, tenantname, 'gigabyte') INTO procArray;
559.   ELSE
560.   EXECUTE FORMAT('SELECT ARRAY_AGG(p.proname) FROM pg_proc p, pg_namespace n WHERE n.oid =
       p.pronamespace AND has_function_privilege(%L, p.oid, ''execute'') AND n.nspname = %L AND
       p.proname != ''getProcesses'' AND p.proname != ''getProcessParams'' ', username, tenant-
       name) INTO procArray;
561.   END IF;
562.   RETURN procArray;
563.
564.   END;
565.   $$ LANGUAGE plpgsql;
566.
567.
568.   CREATE OR REPLACE FUNCTION bank."getProcessParams"(processname text)
569.   RETURNS TABLE(argumentname text, argumenttype text) AS $$
570.   DECLARE
571.   procArray text[];
572.   elem text;
573.
574.   BEGIN
575.
```

```
576.    procArray := (SELECT string_to_array(pg_get_function_arguments(p.oid), ',') FROM pg_proc
         p, pg_namespace n WHERE p.pronamespace = n.oid AND n.nspname != 'pg_catalog' AND
         n.nspname != 'information_schema' AND p.proname = processname);
577.    FOREACH elem IN ARRAY procArray LOOP
578.     argumentname := (split_part(elem, ' ', 1));
579.    argumenttype := (split_part(elem, ' ', 2));
580.    RETURN NEXT;
581.    END LOOP;
582.
583.    END;
584.    $$ LANGUAGE plpgsql;
585.
586.    CREATE TRIGGER trigger_bank_user
587.    AFTER INSERT OR DELETE OR UPDATE
588.    ON bank."user" FOR EACH ROW
589.    EXECUTE FUNCTION spschema.log_trigger();
590.
591.    CREATE TRIGGER trigger_bank_userconditions
592.    AFTER INSERT OR DELETE OR UPDATE
593.    ON bank.userconditions FOR EACH ROW
594.    EXECUTE FUNCTION spschema.log_trigger();
595.
596.    REVOKE ALL ON SCHEMA bank FROM public;
597.    REVOKE ALL ON ALL TABLES IN SCHEMA bank FROM public;
598.    REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA bank FROM public;
599.    REVOKE ALL ON SCHEMA bank FROM bank;
600.    GRANT USAGE ON SCHEMA bank TO bank;
601.    REVOKE ALL ON ALL TABLES IN SCHEMA bank FROM bank;
602.    REVOKE ALL ON ALL FUNCTIONS IN SCHEMA bank FROM bank;
603.    GRANT ALL ON ALL TABLES IN SCHEMA bank TO bank;
604.    GRANT ALL ON ALL FUNCTIONS IN SCHEMA bank TO bank;
605.
606.    CREATE TABLE IF NOT EXISTS bank."Transaction"(amount double precision, transactionDate
       date, transactionType varchar, transactionNum integer,
607.    CONSTRAINT pkTransaction PRIMARY KEY (transactionNum));
608.    ALTER TABLE bank."Transaction" ALTER COLUMN amount SET NOT NULL;
609.    ALTER TABLE bank."Transaction" ALTER COLUMN transactionDate SET NOT NULL;
610.    ALTER TABLE bank."Transaction" ALTER COLUMN transactionType SET NOT NULL;
611.    ALTER TABLE bank."Transaction" ALTER COLUMN transactionNum SET NOT NULL;
612.
613.    CREATE TRIGGER "trigger_bank_Transaction"
614.    AFTER INSERT OR DELETE OR UPDATE
615.    ON bank."Transaction" FOR EACH ROW
616.    WHEN (pg_trigger_depth() < 1)EXECUTE FUNCTION spschema.log_trigger();
617.
618.    CREATE TABLE IF NOT EXISTS bank."Account"(accountType varchar, balance double precision,
       status varchar, accountNum integer,
619.    CONSTRAINT pkAccount PRIMARY KEY (accountNum));
620.    ALTER TABLE bank."Account" ALTER COLUMN accountType SET NOT NULL;
621.    ALTER TABLE bank."Account" ALTER COLUMN balance SET NOT NULL;
622.    ALTER TABLE bank."Account" ALTER COLUMN status SET NOT NULL;
623.    ALTER TABLE bank."Account" ALTER COLUMN accountNum SET NOT NULL;
624.
625.    CREATE TRIGGER "trigger_bank_Account"
626.    AFTER INSERT OR DELETE OR UPDATE
627.    ON bank."Account" FOR EACH ROW
628.    WHEN (pg_trigger_depth() < 1)EXECUTE FUNCTION spschema.log_trigger();
629.
630.    CREATE TABLE IF NOT EXISTS bank."Branch"(branchNum integer, location varchar,
631.    CONSTRAINT pkBranch PRIMARY KEY (branchNum));
632.
633.    ALTER TABLE bank."Branch" ALTER COLUMN location SET NOT NULL;
634.    ALTER TABLE bank."Branch" ALTER COLUMN branchNum SET NOT NULL;
635.
636.    CREATE TRIGGER "trigger_bank_Branch"
637.    AFTER INSERT OR DELETE OR UPDATE
638.    ON bank."Branch" FOR EACH ROW
639.    WHEN (pg_trigger_depth() < 1)EXECUTE FUNCTION spschema.log_trigger();
640.
```

```
641.  CREATE TABLE IF NOT EXISTS bank."Customer"(sex char, name varchar, addresslocality var-
      char, addressstreet varchar, contactNum varchar, surname varchar, customerNum integer,
642.  CONSTRAINT pkCustomer PRIMARY KEY (customerNum));
643.  ALTER TABLE bank."Customer" ALTER COLUMN customerNum SET NOT NULL;
644.  ALTER TABLE bank."Customer" ALTER COLUMN contactNum SET NOT NULL;
645.  ALTER TABLE bank."Customer" ALTER COLUMN name SET NOT NULL;
646.  ALTER TABLE bank."Customer" ALTER COLUMN surname SET NOT NULL;
647.  ALTER TABLE bank."Customer" ALTER COLUMN addresslocality SET NOT NULL;
648.  ALTER TABLE bank."Customer" ALTER COLUMN addressstreet SET NOT NULL;
649.
650.  CREATE TRIGGER "trigger_bank_Customer"
651.  AFTER INSERT OR DELETE OR UPDATE
652.  ON bank."Customer" FOR EACH ROW
653.  WHEN (pg_trigger_depth() < 1)EXECUTE FUNCTION spschema.log_trigger();
654.
655.  CREATE TABLE IF NOT EXISTS bank."Customer_Account"(customerNum integer, accountNum inte-
      ger,
656.  CONSTRAINT pk_Customer_Account PRIMARY KEY (customerNum, accountNum));
657.  ALTER TABLE bank."Customer_Account" DROP CONSTRAINT IF EXISTS fkCustomer_Customer_Ac-
      count;
658.  ALTER TABLE bank."Customer_Account" ADD CONSTRAINT fkCustomer_Customer_Account FOREIGN
       KEY(customerNum) REFERENCES bank."Customer"(customerNum)
659.  ON UPDATE CASCADE ON DELETE CASCADE;
660.  ALTER TABLE bank."Customer_Account" DROP CONSTRAINT IF EXISTS fkAccount_Customer_Account;
661.  ALTER TABLE bank."Customer_Account" ADD CONSTRAINT fkAccount_Customer_Account FOREIGN
       KEY(accountNum) REFERENCES bank."Account"(accountNum)
662.  ON UPDATE CASCADE ON DELETE CASCADE;
663.
664.  ALTER TABLE bank."Account" ADD COLUMN IF NOT EXISTS branchNumRef integer NOT NULL;
665.  ALTER TABLE bank."Account" DROP CONSTRAINT IF EXISTS uniqueAccount_branchNumRef;
666.  ALTER TABLE bank."Account" ADD CONSTRAINT uAccount_branchNumRef UNIQUE (branchNumRef);
667.  ALTER TABLE bank."Account" DROP CONSTRAINT IF EXISTS fk_Account_Branch;
668.  ALTER TABLE bank."Account" ADD CONSTRAINT fk_Account_Branch FOREIGN KEY(branchNumRef)
      REFERENCES bank."Branch"(branchNum)
669.  ON UPDATE CASCADE ON DELETE CASCADE;
670.
671.  ALTER TABLE bank."Transaction" ADD COLUMN IF NOT EXISTS accountNumRef integer NOT NULL;
672.  ALTER TABLE bank."Transaction" DROP CONSTRAINT IF EXISTS uniqueTransaction_accountNumRef;
673.  ALTER TABLE bank."Transaction" ADD CONSTRAINT uTransac_accNumRef UNIQUE(accountNumRef);
674.  ALTER TABLE bank."Transaction" DROP CONSTRAINT IF EXISTS fk_Transaction_Account;
675.  ALTER TABLE bank."Transaction" ADD CONSTRAINT fk_Transaction_Account FOREIGN KEY(account-
      NumRef) REFERENCES bank."Account"(accountNum)
676.  ON UPDATE CASCADE ON DELETE CASCADE;
677.
678.  INSERT INTO spschema.roles(rolename, partitioned)
679.  VALUES ('Clerkbank', true);
680.
681.  INSERT INTO spschema.views(viewname, tablename, canInsert, canUpdate, canDelete,
      rolename)
682.  VALUES ('Clerkbank_Transaction', 'Transaction', false, false, true, 'Clerkbank');
683.  INSERT INTO spschema."viewsColumns"(viewname, columnname, horizpart)
684.  VALUES ('Clerkbank_Transaction', 'transactionType', false);
685.  INSERT INTO spschema."viewsColumns"(viewname, columnname, horizpart)
686.  VALUES ('Clerkbank_Transaction', 'transactionType', true);
687.
688.  INSERT INTO spschema.views(viewname, tablename, canInsert, canUpdate, canDelete,
      rolename)
689.  VALUES ('Clerkbank_Account', 'Account', true, true, true, 'Clerkbank');
690.  INSERT INTO spschema."viewsColumns"(viewname, columnname, horizpart)
691.  VALUES ('Clerkbank_Account', 'status', true);
692.
693.  INSERT INTO spschema.roles(rolename, partitioned)
694.  VALUES ('CEObank', true);
695.
696.  INSERT INTO spschema.views(viewname, tablename, canInsert, canUpdate, canDelete,
      rolename)
697.  VALUES ('CEObank_Branch', 'Branch', true, true, true, 'CEObank');
698.  INSERT INTO spschema."viewsColumns"(viewname, columnname, horizpart)
699.  VALUES ('CEObank_Branch', 'location', true);
700.
```

```
701.  CREATE VIEW bank."Clerkbank_Transaction" WITH (security_invoker = true) AS SELECT trans-
      actionType FROM bank."Transaction";
702.
703.  INSERT INTO spschema.roles(rolename, partitioned)
704.  VALUES ('Customerbank', false);
705.
706.  INSERT INTO spschema.tenant(sn, tenant_name, tenant_pswd)
707.  VALUES (4, 'bank', 'bank');
708.
709.  CREATE ROLE "Clerkbank" WITH
710.  NOSUPERUSER
711.  NOCREATEDB
712.  NOCREATEROLE
713.  NOINHERIT
714.  NOREPLICATION
715.  NOBYPASSRLS
716.  NOLOGIN;
717.
718.  REVOKE ALL ON DATABASE bank FROM "Clerkbank";
719.  GRANT CONNECT ON DATABASE bank TO "Clerkbank" ;
720.  REVOKE ALL ON SCHEMA spschema FROM "Clerkbank";
721.  GRANT USAGE ON SCHEMA spschema TO "Clerkbank";
722.  REVOKE ALL ON ALL TABLES IN SCHEMA spschema FROM "Clerkbank";
723.  REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA spschema FROM "Clerkbank";
724.  REVOKE ALL ON SCHEMA bank FROM "Clerkbank";
725.  GRANT USAGE ON SCHEMA bank TO "Clerkbank";
726.  REVOKE ALL ON ALL TABLES IN SCHEMA bank FROM "Clerkbank";
727.  REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA bank FROM "Clerkbank";
728.  GRANT EXECUTE ON FUNCTION bank."getProcesses" TO "Clerkbank";
729.  GRANT EXECUTE ON FUNCTION bank."getProcessParams" TO "Clerkbank";
730.
731.  CREATE ROLE "CEObank" WITH
732.  NOSUPERUSER
733.  NOCREATEDB
734.  NOCREATEROLE
735.  NOINHERIT
736.  NOREPLICATION
737.  NOBYPASSRLS
738.  NOLOGIN;
739.
740.  REVOKE ALL ON DATABASE bank FROM "CEObank";
741.  GRANT CONNECT ON DATABASE bank TO "CEObank" ;
742.  REVOKE ALL ON SCHEMA spschema FROM "CEObank";
743.  GRANT USAGE ON SCHEMA spschema TO "CEObank";
744.  REVOKE ALL ON ALL TABLES IN SCHEMA spschema FROM "CEObank";
745.  REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA spschema FROM "CEObank";
746.  REVOKE ALL ON SCHEMA bank FROM "CEObank";
747.  GRANT USAGE ON SCHEMA bank TO "CEObank";
748.  REVOKE ALL ON ALL TABLES IN SCHEMA bank FROM "CEObank";
749.  REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA bank FROM "CEObank";
750.  GRANT EXECUTE ON FUNCTION bank."getProcesses" TO "CEObank";
751.  GRANT EXECUTE ON FUNCTION bank."getProcessParams" TO "CEObank";
752.
753. CREATE ROLE "Customerbank" WITH
754.  NOSUPERUSER
755.  NOCREATEDB
756.  NOCREATEROLE
757.  NOINHERIT
758.  NOREPLICATION
759.  NOBYPASSRLS
760.  NOLOGIN;
761.
762.  REVOKE ALL ON DATABASE bank FROM "Customerbank";
763.  GRANT CONNECT ON DATABASE bank TO "Customerbank" ;
764.  REVOKE ALL ON SCHEMA spschema FROM "Customerbank";
765.  GRANT USAGE ON SCHEMA spschema TO "Customerbank";
766.  REVOKE ALL ON ALL TABLES IN SCHEMA spschema FROM "Customerbank";
767.  REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA spschema FROM "Customerbank";
768.  REVOKE ALL ON SCHEMA bank FROM "Customerbank";
769.  GRANT USAGE ON SCHEMA bank TO "Customerbank";
```

```
770.  REVOKE ALL ON ALL TABLES IN SCHEMA bank FROM "Customerbank";
771.  REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA bank FROM "Customerbank";
772.  GRANT EXECUTE ON FUNCTION bank."getProcesses" TO "Customerbank";
773.  GRANT EXECUTE ON FUNCTION bank."getProcessParams" TO "Customerbank";
774.
775.  CREATE FUNCTION bank."depositPayment"(attValPair JSON)
776.  RETURNS text
777.  AS
778.  $body$
779.  #variable_conflict use_variable
780.  DECLARE
781.  --include required variables
782.  returnResult integer;
783.
784.  BEGIN
785.  --body of function
786.
787.    INSERT INTO bank."Clerkbank_Transaction"
788.    SELECT * FROM json_populate_record(NULL::bank."Clerkbank_Transaction", attValPair)
789.    RETURNING * INTO returnResult;
790.
791.    IF returnResult > 0 THEN
792.    RETURN 'Insert operation conducted successfully';
793.    ELSE
794.    RAISE EXCEPTION 'ERROR';
795.    END IF;
796.  END
797.  $body$
798.  LANGUAGE plpgsql;
799.
800.  CREATE FUNCTION bank."withdrawPayment"(attValPair JSON)
801.  RETURNS text
802.  AS
803.  $body$
804.  #variable_conflict use_variable
805.  DECLARE
806.  --include required variables
807.  returnResult integer;
808.
809.  BEGIN
810.  --body of function
811.
812.    INSERT INTO bank."Clerkbank_Transaction"
813.    SELECT * FROM json_populate_record(NULL::bank."Clerkbank_Transaction", attValPair)
814.    RETURNING * INTO returnResult;
815.
816.    IF returnResult > 0 THEN
817.    RETURN 'Insert operation conducted successfully';
818.    ELSE
819.    RAISE EXCEPTION 'ERROR';
820.    END IF;
821.  END
822.  $body$
823.  LANGUAGE plpgsql;
824.
825.  CREATE FUNCTION bank."makeDirectPay"(attValPair JSON)
826.  RETURNS text
827.  AS
828.  $body$
829.  #variable_conflict use_variable
830.  DECLARE
831.  --include required variables
832.  returnResult integer;
833.
834.  BEGIN
835.  --body of function
836.
837.    INSERT INTO bank."Clerkbank_Transaction"
838.    SELECT * FROM json_populate_record(NULL::bank."Clerkbank_Transaction", attValPair)
839.    RETURNING * INTO returnResult;
```

```
840.
841.     IF returnResult > 0 THEN
842.     RETURN 'Insert operation conducted successfully';
843.     ELSE
844.     RAISE EXCEPTION 'ERROR';
845.     END IF;
846.   END
847.   $body$
848.   LANGUAGE plpgsql;
849.
850.   CREATE FUNCTION bank."createTransaction"(childSelection JSON, finalChildParams JSON)
851.   RETURNS text
852.   AS
853.   $body$
854.   #variable_conflict use_variable
855.   DECLARE
856.   --include required variables
857.   returnResult text;
858.
859.   BEGIN
860.   --body of function
861.    IF (SELECT value FROM json_each_text(childSelection->0) LIMIT 1)::INTEGER =   THEN
862.     returnResult := (SELECT * FROM bank."depositPayment"(finalChildParams));
863.
864.     ELSEIF (SELECT value FROM json_each_text(childSelection->0) LIMIT 1)::INTEGER = 2 THEN
865.     returnResult := (SELECT * FROM bank."makeDirectPay"(finalChildParams));
866.
867.     ELSEIF (SELECT value FROM json_each_text(childSelection->0) LIMIT 1)::INTEGER = 3 THEN
868.     returnResult := (SELECT * FROM bank."withdrawPayment"(finalChildParams));
869.     ELSE RAISE NOTICE 'Selection inputted does not exist';
870.     END IF;
871.
872.    RETURN returnResult;
873.   END
874.   $body$
875.   LANGUAGE plpgsql;
876.
877.   CREATE FUNCTION bank."reverseTransaction"(transactionNum integer)
878.   RETURNS text
879.   AS
880.   $body$
881.   #variable_conflict use_variable
882.   DECLARE
883.   --include required variables
884.   returnResult integer;
885.
886.   BEGIN
887.   --body of function
888.    DELETE FROM bank."Clerkbank_Transaction"
889.    WHERE "Clerkbank_Transaction".transactionNum = transactionNum
890.    RETURNING * INTO returnResult;
891.
892.    IF returnResult > 0 THEN
893.    RETURN 'Delete operation conducted successfully';
894.    ELSE
895.    RAISE EXCEPTION 'ERROR';
896.    END IF;
897.   END
898.   $body$
899.   LANGUAGE plpgsql;
900.
901.   CREATE FUNCTION bank."updateTransaction"(attValPair JSON, transactionNum integer)
902.   RETURNS text
903.   AS
904.   $body$
905.   #variable_conflict use_variable
906.   DECLARE
907.   --include required variables
908.   returnResult integer;
909.
```

```
910.  BEGIN
911.  --body of function
912.   IF !(SELECT canUpdate FROM spschema.views WHERE viewname = 'Clerkbank_Transaction') THEN
913.    RAISE EXCEPTION 'Update operation cannot take place';
914.   ELSE
915.    UPDATE bank."Clerkbank_Transaction"
916.    SET "transactionType" = json_record."transactionType"
917.    FROM json_populate_record(NULL::bank."Clerkbank_Transaction", attValPair) AS json_rec-
        ord
918.    WHERE "Clerkbank_Transaction"."transactionNum" = transactionNum
919.    RETURNING * INTO returnResult; END IF;
920.
921.
922.    IF returnResult > 0 THEN
923.    RETURN 'Insert operation conducted successfully';
924.    ELSE
925.    RAISE EXCEPTION 'ERROR';
926.    END IF;
927.  END
928.  $body$
929.  LANGUAGE plpgsql;
930.
931.  CREATE FUNCTION bank."openAccount"(attValPair JSON)
932.  RETURNS text
933.  AS
934.  $body$
935.  #variable_conflict use_variable
936.  DECLARE
937.  --include required variables
938.  returnResult integer;
939.
940.  BEGIN
941.  --body of function
942.
943.    INSERT INTO bank."Account"
944.    SELECT * FROM json_populate_record(NULL::bank."Account", attValPair)
945.    RETURNING * INTO returnResult;
946.
947.    IF returnResult > 0 THEN
948.    RETURN 'Insert operation conducted successfully';
949.    ELSE
950.    RAISE EXCEPTION 'ERROR';
951.    END IF;
952.  END
953.  $body$
954.  LANGUAGE plpgsql;
955.
956.  CREATE FUNCTION bank."openBranch"(attValPair JSON)
957.  RETURNS text
958.  AS
959.  $body$
960.  #variable_conflict use_variable
961.  DECLARE
962.  --include required variables
963.  returnResult integer;
964.
965.  BEGIN
966.  --body of function
967.
968.    INSERT INTO bank."Branch"
969.    SELECT * FROM json_populate_record(NULL::bank."Branch", attValPair)
970.    RETURNING * INTO returnResult;
971.
972.    IF returnResult > 0 THEN
973.    RETURN 'Insert operation conducted successfully';
974.    ELSE
975.    RAISE EXCEPTION 'ERROR';
976.    END IF;
977.  END
978.  $body$
```

```
   LANGUAGE plpgsql;

   CREATE FUNCTION bank."editBranchDetails"(attValPair JSON, branchNum integer)
   RETURNS text
   AS
   $body$
   #variable_conflict use_variable
   DECLARE
   --include required variables
   returnResult integer;

   BEGIN
   --body of function
      UPDATE bank."Branch"
     SET "branchNum" = json_record."branchNum", location = json_record.location
     FROM json_populate_record(NULL::bank."Branch", attValPair) AS json_record
     WHERE "Branch"."branchNum" = branchNum
     RETURNING * INTO returnResult;

     IF returnResult > 0 THEN
     RETURN 'Insert operation conducted successfully';
     ELSE
     RAISE EXCEPTION 'ERROR';
     END IF;
   END
   $body$
   LANGUAGE plpgsql;

   CREATE FUNCTION bank."closeBranch"(branchNum integer)
   RETURNS text
   AS
   $body$
   #variable_conflict use_variable
   DECLARE
   --include required variables
   returnResult integer;

   BEGIN
   --body of function
    DELETE FROM bank."Branch"
    WHERE "Branch".branchNum = branchNum
    RETURNING * INTO returnResult;

    IF returnResult > 0 THEN
    RETURN 'Delete operation conducted successfully';
    ELSE
    RAISE EXCEPTION 'ERROR';
    END IF;
   END
   $body$
   LANGUAGE plpgsql;

   CREATE FUNCTION bank."customerRegistration"(attValPair JSON)
   RETURNS text
   AS
   $body$
   #variable_conflict use_variable
   DECLARE
   --include required variables
   returnResult integer;

   BEGIN
   --body of function
     IF "Clerk" IN (SELECT pg_has_role((SELECT current_user), 'Member')) THEN
     INSERT INTO bank."Customer"
     SELECT * FROM json_populate_record(NULL::bank."Customer", attValPair)
     RETURNING * INTO returnResult;

     ELSEIF "Customer" IN (SELECT pg_has_role((SELECT current_user), 'Member')) THEN
     INSERT INTO bank."Customer"
```

```
1049.     SELECT * FROM json_populate_record(NULL::bank."Customer", attValPair)
1050.     RETURNING * INTO returnResult;
1051.
1052.     ELSE RAISE EXCEPTION 'Process cannot be executed';
1053.     END IF;
1054.
1055.
1056.     IF returnResult > 0 THEN
1057.     RETURN 'Insert operation conducted successfully';
1058.     ELSE
1059.     RAISE EXCEPTION 'ERROR';
1060.     END IF;
1061.   END
1062.   $body$
1063.   LANGUAGE plpgsql;
1064.
1065.   CREATE FUNCTION bank."editCustomerDetails"(attValPair JSON, customerNum integer)
1066.   RETURNS text
1067.   AS
1068.   $body$
1069.   #variable_conflict use_variable
1070.   DECLARE
1071.   --include required variables
1072.   returnResult integer;
1073.
1074.   BEGIN
1075.   --body of function
1076.     UPDATE bank."Customer"
1077.     SET "addresshouseNum" = json_record."addresshouseNum", name = json_record.name, sex =
          json_record.sex, "customerNum" = json_record."customerNum", "contactNum" = json_rec-
          ord."contactNum", surname = json_record.surname, addresslocality = json_record.ad-
          dresslocality, addressstreet = json_record.addressstreet
1078.     FROM json_populate_record(NULL::bank."Customer", attValPair) AS json_record
1079.     WHERE "Customer"."customerNum" = customerNum
1080.     RETURNING * INTO returnResult;
1081.
1082.     IF returnResult > 0 THEN
1083.     RETURN 'Insert operation conducted successfully';
1084.     ELSE
1085.     RAISE EXCEPTION 'ERROR';
1086.     END IF;
1087.   END
1088.   $body$
1089.   LANGUAGE plpgsql;
1090.
1091.   CREATE FUNCTION bank."removeCustomer"(customerNum integer)
1092.   RETURNS text
1093.   AS
1094.   $body$
1095.   #variable_conflict use_variable
1096.   DECLARE
1097.   --include required variables
1098.   returnResult integer;
1099.
1100.   BEGIN
1101.   --body of function
1102.    DELETE FROM bank."Customer"
1103.    WHERE "Customer".customerNum = customerNum
1104.    RETURNING * INTO returnResult;
1105.
1106.    IF returnResult > 0 THEN
1107.    RETURN 'Delete operation conducted successfully';
1108.    ELSE
1109.    RAISE EXCEPTION 'ERROR';
1110.    END IF;
1111.   END
1112.   $body$
1113.   LANGUAGE plpgsql;
1114.
1115.   CREATE FUNCTION bank."updateAccount"(attValPair JSON, accountNum integer)
```

```
1116.   RETURNS text
1117.   AS
1118.   $body$
1119.   #variable_conflict use_variable
1120.   DECLARE
1121.   --include required variables
1122.   returnResult integer;
1123.
1124.   BEGIN
1125.   --body of function
1126.     UPDATE bank."Account"
1127.     SET "branchNumRef" = json_record."branchNumRef", "accountType" = json_record."ac-
        countType", balance = json_record.balance, status = json_record.status, "accountNum" =
        json_record."accountNum"
1128.     FROM json_populate_record(NULL::bank."Account", attValPair) AS json_record
1129.     WHERE "Account"."accountNum" = accountNum
1130.     RETURNING * INTO returnResult;
1131.
1132.     IF returnResult > 0 THEN
1133.     RETURN 'Insert operation conducted successfully';
1134.     ELSE
1135.     RAISE EXCEPTION 'ERROR';
1136.     END IF;
1137.   END
1138.   $body$
1139.   LANGUAGE plpgsql;
1140.
1141.   CREATE FUNCTION bank."closeAccount"(attValPair JSON, accountNum integer)
1142.   RETURNS text
1143.   AS
1144.   $body$
1145.   #variable_conflict use_variable
1146.   DECLARE
1147.   --include required variables
1148.   returnResult integer;
1149.
1150.   BEGIN
1151.   --body of function
1152.      UPDATE bank."Account"
1153.     SET "branchNumRef" = json_record."branchNumRef", "accountType" = json_record."ac-
         countType", balance = json_record.balance, status = json_record.status, "accountNum" =
         json_record."accountNum"
1154.     FROM json_populate_record(NULL::bank."Account", attValPair) AS json_record
1155.     WHERE "Account"."accountNum" = accountNum
1156.     RETURNING * INTO returnResult;
1157.
1158.     IF returnResult > 0 THEN
1159.     RETURN 'Insert operation conducted successfully';
1160.     ELSE
1161.     RAISE EXCEPTION 'ERROR';
1162.     END IF;
1163.   END
1164.   $body$
1165.   LANGUAGE plpgsql;
1166.
1167.   CREATE FUNCTION bank."removeAccount"(accountNum integer)
1168.   RETURNS text
1169.   AS
1170.   $body$
1171.   #variable_conflict use_variable
1172.   DECLARE
1173.   --include required variables
1174.   returnResult integer;
1175.
1176.   BEGIN
1177.   --body of function
1178.    DELETE FROM bank."Account"
1179.    WHERE "Account".accountNum = accountNum
1180.    RETURNING * INTO returnResult;
1181.
```

```
1182.    IF returnResult > 0 THEN
1183.    RETURN 'Delete operation conducted successfully';
1184.    ELSE
1185.    RAISE EXCEPTION 'ERROR';
1186.    END IF;
1187.   END
1188.   $body$
1189.   LANGUAGE plpgsql;
1190.
1191.   CREATE FUNCTION bank."generateReportAccount"()
1192.   RETURNS JSON
1193.   AS
1194.   $body$
1195.   #variable_conflict use_variable
1196.   DECLARE
1197.   --include required variables
1198.   returnResult JSON;
1199.
1200.   BEGIN
1201.   --body of function
1202.    returnResult:=(SELECT json_agg(jagg) FROM(SELECT * FROM bank."Account" as agg) as jagg);
1203.    RETURN returnResult;
1204.
1205.   END
1206.   $body$
1207.   LANGUAGE plpgsql;
1208.
1209.   CREATE FUNCTION bank."generateReportBranch"()
1210.   RETURNS JSON
1211.   AS
1212.   $body$
1213.   #variable_conflict use_variable
1214.   DECLARE
1215.   --include required variables
1216.   returnResult JSON;
1217.
1218.   BEGIN
1219.   --body of function
1220.    returnResult:=(SELECT json_agg(jagg) FROM (SELECT * FROM bank."Branch" as agg) as jagg);
1221.    RETURN returnResult;
1222.
1223.   END
1224.   $body$
1225.   LANGUAGE plpgsql;
1226.
1227.   CREATE FUNCTION bank."generateReportCustomer"()
1228.   RETURNS JSON
1229.   AS
1230.   $body$
1231.   #variable_conflict use_variable
1232.   DECLARE
1233.   --include required variables
1234.   returnResult JSON;
1235.
1236.   BEGIN
1237.   --body of function
1238.    returnResult := (SELECT json_agg(jagg) FROM (SELECT * FROM bank."Customer" as agg) as
jagg);
1239.    RETURN returnResult;
1240.
1241.   END
1242.   $body$
1243.   LANGUAGE plpgsql;
1244.
1245.   CREATE FUNCTION bank."generateReportTransaction"(attValPair JSON, transactionNum integer)
1246.   RETURNS text
1247.   AS
1248.   $body$
1249.   #variable_conflict use_variable
1250.   DECLARE
```

```
1251.   --include required variables
1252.   returnResult integer;
1253.
1254.   BEGIN
1255.   --body of function
1256.    IF !(SELECT canUpdate FROM spschema.views WHERE viewname = 'Clerkbank_Transaction') THEN
1257.     RAISE EXCEPTION 'Update operation cannot take place';
1258.    ELSE
1259.      UPDATE bank."Clerkbank_Transaction"
1260.     SET "transactionType" = json_record."transactionType"
1261.     FROM json_populate_record(NULL::bank."Clerkbank_Transaction", attValPair) AS json_rec-
         ord
1262.     WHERE "Clerkbank_Transaction"."transactionNum" = transactionNum
1263.     RETURNING * INTO returnResult; END IF;
1264.
1265.     IF returnResult > 0 THEN
1266.     RETURN 'Insert operation conducted successfully';
1267.     ELSE
1268.     RAISE EXCEPTION 'ERROR';
1269.     END IF;
1270.   END
1271.   $body$
1272.   LANGUAGE plpgsql;
1273.
1274.
1275.   ALTER TABLE spschema.views ENABLE ROW LEVEL SECURITY;
1276.
1277.   CREATE POLICY "views_Clerkbank_policy"
1278.   ON spschema.views
1279.   TO "Clerkbank"
1280.   USING (rolename = 'Clerkbank')
1281.   WITH CHECK (rolename = 'Clerkbank');
1282.
1283.   GRANT SELECT ON spschema.views TO "Clerkbank";
1284.
1285.   CREATE POLICY "views_CEObank_policy"
1286.   ON spschema.views
1287.   TO "CEObank"
1288.   USING (rolename = 'CEObank')
1289.   WITH CHECK (rolename = 'CEObank');
1290.
1291.   GRANT SELECT ON spschema.views TO "CEObank";
1292.
1293.   CREATE POLICY "views_Customerbank_policy"
1294.   ON spschema.views
1295.   TO "Customerbank"
1296.   USING (rolename = 'Customerbank')
1297.   WITH CHECK (rolename = 'Customerbank');
1298.
1299.   GRANT SELECT ON spschema.views TO "Customerbank";
1300.
1301.   ALTER TABLE spschema."viewsColumns" ENABLE ROW LEVEL SECURITY;
1302.
1303.   CREATE POLICY "viewsColumns_Clerkbank_policy"
1304.   ON spschema."viewsColumns"
1305.   TO "Clerkbank"
1306.   USING (rolename = 'Clerkbank')
1307.   WITH CHECK (rolename = 'Clerkbank');
1308.
1309.   GRANT SELECT ON spschema."viewsColumns" TO "Clerkbank";
1310.
1311.   CREATE POLICY "viewsColumns_CEObank_policy"
1312.   ON spschema."viewsColumns"
1313.   TO "CEObank"
1314.   USING (rolename = 'CEObank')
1315.   WITH CHECK (rolename = 'CEObank');
1316.
1317.   GRANT SELECT ON spschema."viewsColumns" TO "CEObank";
1318.
1319.   CREATE POLICY "viewsColumns_Customerbank_policy"
```

```
1320.   ON spschema."viewsColumns"
1321.   TO "Customerbank"
1322.   USING (rolename = 'Customerbank')
1323.   WITH CHECK (rolename = 'Customerbank');
1324.
1325.   GRANT SELECT ON spschema."viewsColumns" TO "Customerbank";
1326.
1327.   ALTER TABLE bank.userconditions ENABLE ROW LEVEL SECURITY;
1328.   ALTER TABLE bank.userconditions FORCE ROW LEVEL SECURITY;
1329.
1330.   CREATE POLICY "policy_Clerkbank_userconditions"
1331.   ON bank.userconditions
1332.   TO "Clerkbank"
1333.   USING (rolename = 'Clerkbank' AND username = (SELECT current_user))
1334.   WITH CHECK (rolename = 'Clerkbank' AND username = (SELECT current_user));
1335.
1336.   CREATE POLICY "policy_CEObank_userconditions"
1337.   ON bank.userconditions
1338.   TO "CEObank"
1339.   USING (rolename = 'CEObank' AND username = (SELECT current_user))
1340.   WITH CHECK (rolename = 'CEObank' AND username = (SELECT current_user));
1341.
1342.   GRANT SELECT ON bank.userconditions TO "CEObank";
1343.
1344.   CREATE POLICY "policy_Customerbank_userconditions"
1345.   ON bank.userconditions
1346.   TO "Customerbank"
1347.   USING (rolename = 'Customerbank' AND username = (SELECT current_user))
1348.   WITH CHECK (rolename = 'Customerbank' AND username = (SELECT current_user));
1349.
1350.   GRANT SELECT ON bank.userconditions TO "Customerbank";
1351.
1352.   ALTER TABLE bank."Transaction" ENABLE ROW LEVEL SECURITY;
1353.   ALTER TABLE bank."Transaction" FORCE ROW LEVEL SECURITY;
1354.   ALTER TABLE bank."Account" ENABLE ROW LEVEL SECURITY;
1355.   ALTER TABLE bank."Account" FORCE ROW LEVEL SECURITY;
1356.   ALTER TABLE bank."Branch" ENABLE ROW LEVEL SECURITY;
1357.   ALTER TABLE bank."Branch" FORCE ROW LEVEL SECURITY;
1358.
1359.   CREATE POLICY "policy_Transaction_Clerkbank"
1360.   ON bank."Transaction"
1361.   TO "Clerkbank"
1362.   USING (
1363.   "Transaction".transactionType = ANY (SELECT CAST(val AS varchar)
1364.      FROM bank.userconditions
1365.      WHERE ucviewname = (SELECT DISTINCT views.viewname
1366.                          FROM spschema.views, spschema."viewsColumns"
1367.                          WHERE tablename = 'Transaction'
1368.                          AND columnname = 'transactionType'
1369.                          AND horizpart
1370.                          AND views.viewname = "viewsColumns".viewname)
1371.      AND op = '='
1372.      AND username = (SELECT current_user))
1373.   AND "Transaction".transactionType != ANY (SELECT CAST(val AS varchar)
1374.      FROM bank.userconditions
1375.      WHERE ucviewname = (SELECT DISTINCT views.viewname
1376.                          FROM spschema.views, spschema."viewsColumns"
1377.                          WHERE tablename = 'Transaction'
1378.                          AND columnname = 'transactionType'
1379.                          AND horizpart
1380.                          AND views.viewname = "viewsColumns".viewname)
1381.      AND op = '!='
1382.      AND username = (SELECT current_user))
1383.   )
1384.   WITH CHECK ("Transaction".transactionType = ANY (SELECT CAST(val AS varchar)
1385.      FROM bank.userconditions
1386.      WHERE ucviewname = (SELECT DISTINCT views.viewname
1387.                          FROM spschema.views, spschema."viewsColumns"
1388.                          WHERE tablename = 'Transaction'
1389.                          AND columnname = 'transactionType'
```

```
                        AND horizpart
                        AND views.viewname = "viewsColumns".viewname)
      AND op = '='
      AND username = (SELECT current_user))
    AND "Transaction".transactionType != ANY (SELECT CAST(val AS varchar)
      FROM bank.userconditions
      WHERE ucviewname = (SELECT DISTINCT views.viewname
                        FROM spschema.views, spschema."viewsColumns"
                        WHERE tablename = 'Transaction'
                        AND columnname = 'transactionType'
                        AND horizpart
                        AND views.viewname = "viewsColumns".viewname)
      AND op = '!='
      AND username = (SELECT current_user))
   );

   CREATE POLICY "policy_Account_Clerkbank"
   ON bank."Account"
   TO "Clerkbank"
   USING (
   "Account".status = ANY (SELECT CAST(val AS varchar)
      FROM bank.userconditions
      WHERE ucviewname = (SELECT DISTINCT views.viewname
                        FROM spschema.views, spschema."viewsColumns"
                        WHERE tablename = 'Account'
                        AND columnname = 'status'
                        AND horizpart
                        AND views.viewname = "viewsColumns".viewname)
      AND op = '='
      AND username = (SELECT current_user))
   AND "Account".status != ANY (SELECT CAST(val AS varchar)
      FROM bank.userconditions
      WHERE ucviewname = (SELECT DISTINCT views.viewname
                        FROM spschema.views, spschema."viewsColumns"
                        WHERE tablename = 'Account'
                        AND columnname = 'status'
                        AND horizpart
                        AND views.viewname = "viewsColumns".viewname)
      AND op = '!='
      AND username = (SELECT current_user))
   )
   WITH CHECK ("Account".status = ANY (SELECT CAST(val AS varchar)
      FROM bank.userconditions
      WHERE ucviewname = (SELECT DISTINCT views.viewname
                        FROM spschema.views, spschema."viewsColumns"
                        WHERE tablename = 'Account'
                        AND columnname = 'status'
                        AND horizpart
                        AND views.viewname = "viewsColumns".viewname)
      AND op = '='
      AND username = (SELECT current_user))
    AND "Account".status != ANY (SELECT CAST(val AS varchar)
      FROM bank.userconditions
      WHERE ucviewname = (SELECT DISTINCT views.viewname
                        FROM spschema.views, spschema."viewsColumns"
                        WHERE tablename = 'Account'
                        AND columnname = 'status'
                        AND horizpart
                        AND views.viewname = "viewsColumns".viewname)
      AND op = '!='
      AND username = (SELECT current_user))
   );

   CREATE POLICY "policy_Branch_CEObank"
   ON bank."Branch"
   TO "CEObank"
   USING (
   "Branch".location = ANY (SELECT CAST(val AS varchar)
      FROM bank.userconditions
      WHERE ucviewname = (SELECT DISTINCT views.viewname
```

```
                        FROM spschema.views, spschema."viewsColumns"
                        WHERE tablename = 'Branch'
                        columnname = 'location'
                        AND horizpart
                        AND views.viewname = "viewsColumns".viewname)
      AND op = '='
      AND username = (SELECT current_user))
   AND "Branch".location != ANY (SELECT CAST(val AS varchar)
      FROM bank.userconditions
      WHERE ucviewname = (SELECT DISTINCT views.viewname
                        FROM spschema.views, spschema."viewsColumns"
                        WHERE tablename = 'Branch'
                        AND columnname = 'location'
                        AND horizpart
                        AND views.viewname = "viewsColumns".viewname)
      AND op = '!='
      AND username = (SELECT current_user))
   )
   WITH CHECK ("Branch".location = ANY (SELECT CAST(val AS varchar)
      FROM bank.userconditions
      WHERE ucviewname = (SELECT DISTINCT views.viewname
                        FROM spschema.views, spschema."viewsColumns"
                        WHERE tablename = 'Branch'
                        AND columnname = 'location'
                        AND horizpart
                        AND views.viewname = "viewsColumns".viewname)
      AND op = '='
      AND username = (SELECT current_user))
    AND "Branch".location != ANY (SELECT CAST(val AS varchar)
      FROM bank.userconditions
      WHERE ucviewname = (SELECT DISTINCT views.viewname
                        FROM spschema.views, spschema."viewsColumns"
                        WHERE tablename = 'Branch'
                        AND columnname = 'location'
                        AND horizpart
                        AND views.viewname = "viewsColumns".viewname)
      AND op = '!='
      AND username = (SELECT current_user))
   );

   GRANT EXECUTE ON FUNCTION bank."createTransaction"(JSON, JSON) TO "Clerkbank";
   GRANT EXECUTE ON FUNCTION bank."depositPayment"(JSON) TO "Clerkbank";
   GRANT EXECUTE ON FUNCTION bank."makeDirectPay"(JSON) TO "Clerkbank";
   GRANT EXECUTE ON FUNCTION bank."withdrawPayment"(JSON) TO "Clerkbank";
   GRANT EXECUTE ON FUNCTION bank."editCustomerDetails"(JSON, integer) TO "Clerkbank";
   GRANT EXECUTE ON FUNCTION bank."generateReportCustomer"() TO "Clerkbank";
   GRANT EXECUTE ON FUNCTION bank."generateReportTransaction"(JSON, integer) TO "Clerkbank";
   GRANT EXECUTE ON FUNCTION bank."generateReportBranch"() TO "Clerkbank";
   GRANT EXECUTE ON FUNCTION bank."updateTransaction"(JSON, integer) TO "Clerkbank";
   GRANT EXECUTE ON FUNCTION bank."customerRegistration"(JSON) TO "Customerbank";
   GRANT EXECUTE ON FUNCTION bank."reverseTransaction"(integer) TO "Clerkbank";
   GRANT EXECUTE ON FUNCTION bank."removeCustomer"(integer) TO "Clerkbank";
   GRANT EXECUTE ON FUNCTION bank."removeAccount"(integer) TO "Clerkbank";
   GRANT EXECUTE ON FUNCTION bank."openAccount"(JSON) TO "Clerkbank";
   GRANT EXECUTE ON FUNCTION bank."closeBranch"(integer) TO "CEObank";
   GRANT EXECUTE ON FUNCTION bank."updateAccount"(JSON, integer) TO "Clerkbank";
   GRANT EXECUTE ON FUNCTION bank."generateReportAccount"() TO "Clerkbank";
   GRANT EXECUTE ON FUNCTION bank."closeAccount"(JSON, integer) TO "Clerkbank";
   GRANT EXECUTE ON FUNCTION bank."editBranchDetails"(JSON, integer) TO "CEObank";

   GRANT SELECT ON bank."Customer" TO "Clerkbank";
   GRANT SELECT ON bank."Branch" TO "Clerkbank";
   GRANT SELECT ON bank."Clerkbank_Transaction" TO "Clerkbank";
   GRANT SELECT ON bank."Account" TO "Clerkbank";
   GRANT INSERT ON bank."Customer" TO "Customerbank";
   GRANT INSERT ON bank."Customer" TO "Clerkbank";
   GRANT INSERT ON bank."Clerkbank_Transaction" TO "Clerkbank";
   GRANT INSERT ON bank."Branch" TO "CEObank";
   GRANT INSERT ON bank."Account" TO "Clerkbank";
```

```
1530.   GRANT UPDATE ON bank."Customer" TO "Clerkbank";
1531.   GRANT UPDATE ON bank."Clerkbank_Transaction" TO "Clerkbank";
1532.   GRANT UPDATE ON bank."Branch" TO "CEObank";
1533.   GRANT UPDATE ON bank."Account" TO "Clerkbank";
1534.
1535.   GRANT DELETE ON bank."Customer" TO "Clerkbank";
1536.   GRANT DELETE ON bank."Clerkbank_Transaction" TO "Clerkbank";
1537.   GRANT DELETE ON bank."Branch" TO "CEObank";
1538.   GRANT DELETE ON bank."Account" TO "Clerkbank";
```

Appendix E

E.1 User guide

To execute the developed CASE tool, a series of prerequisites and setup steps must be conducted. This includes downloading and installing PostgreSQL, along with essential extensions like 'pg_crypto,' which are necessary for script execution. Moreover, the installation of mandatory Java packages and the importation of essential Java Archive (JAR) files, such as those associated with the JDBC connection are required.

Moreover, before executing the GUI program, it's imperative to configure the environment. This configuration process involves several crucial steps including:

- Creation of ‘pg_pass’ file and modification of related paths
- Setting up the CSP environment via the execution of the *csp.bat* file located in the ‘CSP’ folder
- Establishing the software provider environment via the execution of the *SoftwareProvider_SETUP.bat* file located in the ‘SoftwareProvider’ folder
- Inclusion of Java and PostgreSQL to the system environmental variables

Furthermore, it should be noted that the file paths specified within these setup files must be carefully tailored to align with the specifics of the computer configuration on which the CASE tool is being executed.

Following successful completion of the above-mentioned steps, the Java program consisting of two main parts - setting up the initial environment and handling subsequent tenant testing and operations – can be executed.

Appendix F

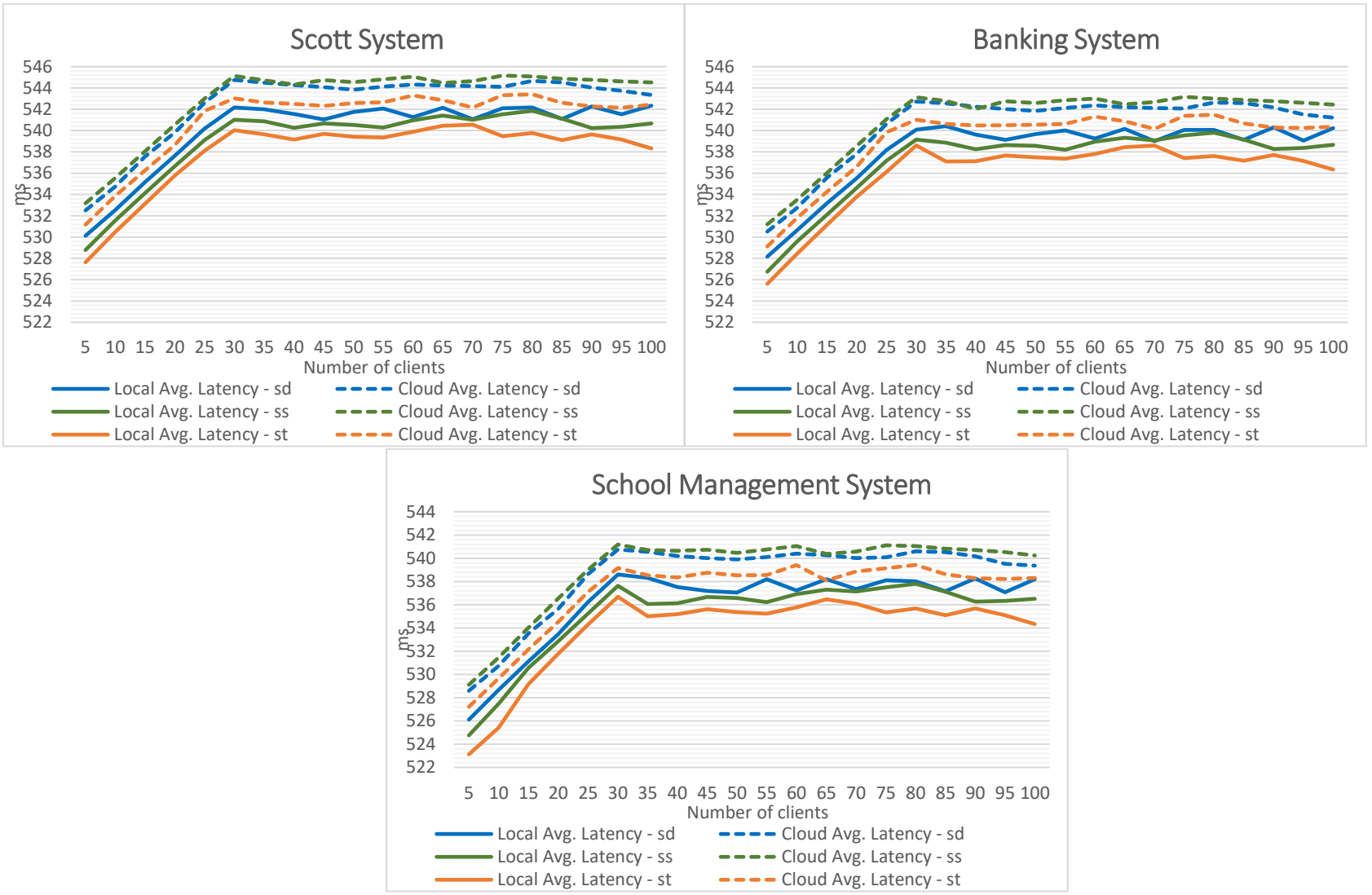


Figure F.1: Performance Benchmark with Security – Average Throughput

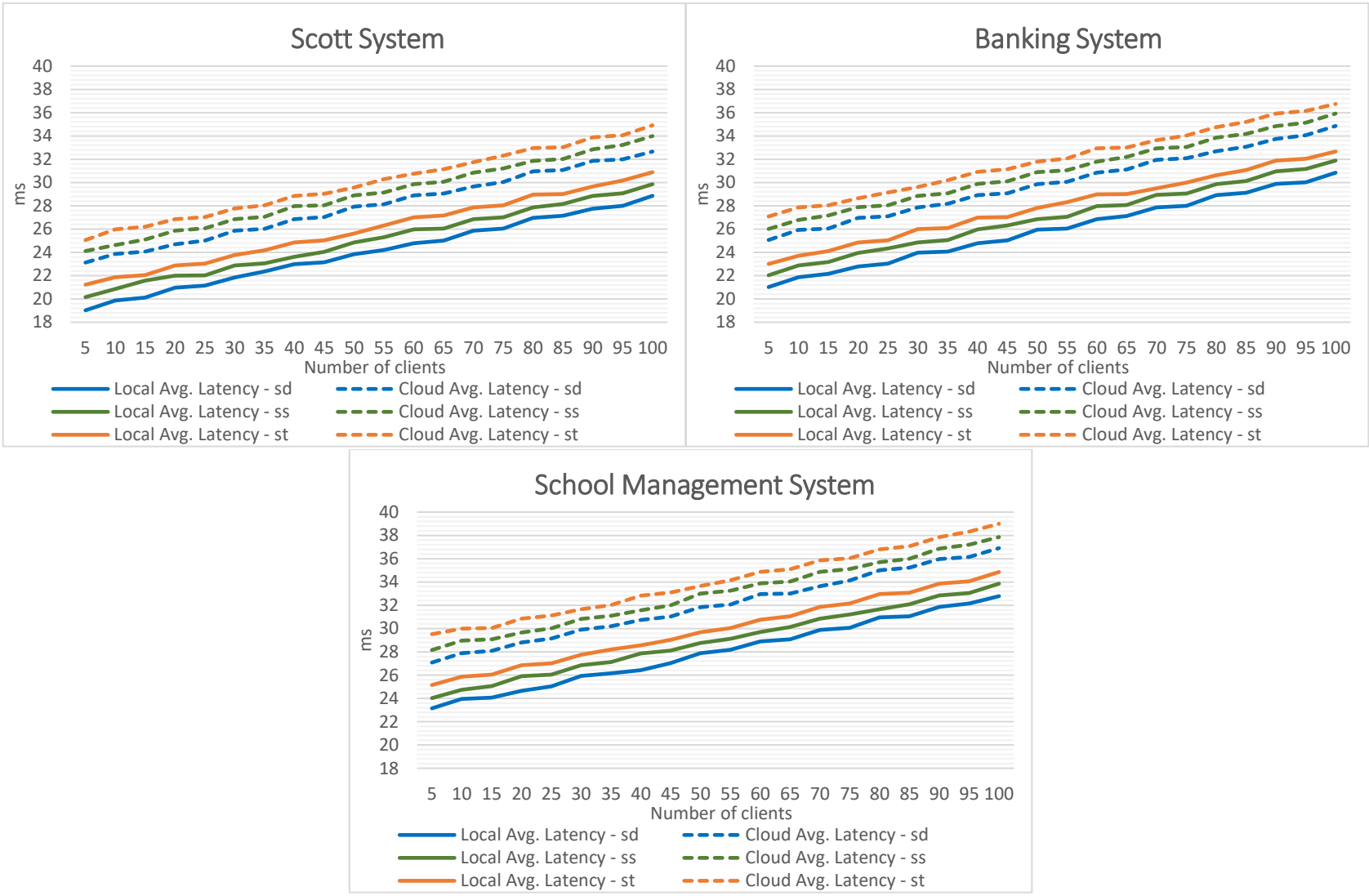


Figure F.2: Performance Benchmark with Security – Average Latency