Information Management

Coursework 2

u5551408

Table of Contents

1 Database Design

To avoid any anomalies when querying data, the database will be set up in 3rd normal form. This means that each value will be atomic, and will depend solely on the primary key of the table.

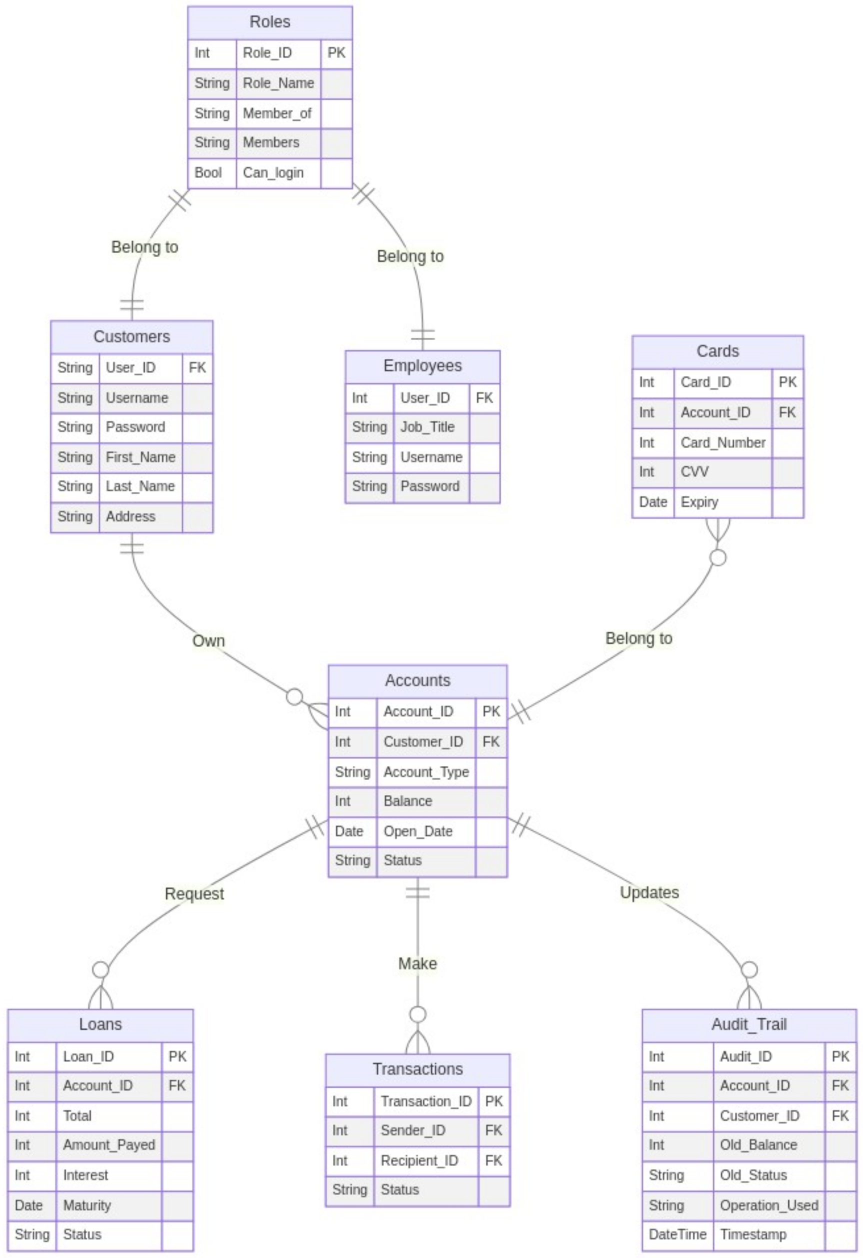


Figure 1: Entity Relationship Diagram for BankDB

1.1 Table Designs

This section will describe and explain each table in the database in reference to the Entity Relationship Diagram (See Figure 1).

1.1.1 Roles

The Roles table stores information about each role in the database. In implementation, these are stored by PostgreSQL as group / user roles, but it has been added to the ERD to demonstrate how it fits in with the users.

- Primary Key: Role_ID

1.1.2 Employees

This table stores all important information about each employee, including their 'Job_Title' which will reflect the role they are given.

- Primary Key: Employee_ID

1.1.3 Customers

The Customers table holds all relevant PII about individual users, including their first names, last names, and address. This is important for identifying a user either over the phone or in a branch, however this is sensitive information and must be kept secure.

- Primary Key: Customer_ID

1.1.4 Accounts

A customer can open many accounts, these are stored here. It shows the balance, they account type, and is the reference point for transactions, loans, and cards.

- Primary Key: Account_ID
- Foreign Key: Customer_ID (→ Customers.Customer_ID)

1.1.5 Cards

Cards which are assigned to an account are stored here. These will be encrypted with a secret key to achieve compliance with PCI DSS.

- Primary Key: Card_ID
- Foreign Key: Account_ID (→ Accounts.Account_ID)

1.1.6 Transactions

Transactions will show the history of payments between accounts within the bank. It has a Sender_ID, and a Recipient_ID, which refer to the accounts in which the transaction took place.

- Primary Key: Transaction_ID
- Foreign Key: Sender_ID ($\rightarrow$ Accounts.Account_ID)
- Foreign Key: Recipient_ID ($\rightarrow$ Accounts.Account_ID)

1.1.7 Loans

Loans should be requested by the user. The Loan Officer should be able to see these, and approve or deny the request. The table will store information like the total amount due, the amount paid so far to allow the customer to pay in instalments, and the maturity date which is the final pay date before fines are incurred.

- Primary Key: Loan_ID
- Foreign Key: Account_ID ($\rightarrow$ Accounts.Account_ID)

1.1.8 Audit_Trail

The audit trail will store a log history of all updates to the database. This includes past transactions and loans (successful or declined), as well as updates to a user's account and changes in their balance. This is done to provide a way to monitor for suspicious activity.

- Primary Key: Audit_ID
- Transaction Type: ('Update', 'Transaction', 'Loan')
- Foreign Key: (ID dependent on Transaction Type)

1.2 Security Requirements

Given the sensitive nature of the information that a bank will store, security requirements are of utmost importance, and will be considered with three principles: confidentiality, integrity, and availability. Confidentiality of data will be maintained by following regulatory compliance with GDPR (UK GDPR, 2016) and PCI DSS (as detailed in the next paragraph). Integrity and availability of the data will be managed with strict access controls to enforce authentication and authorisation of users, and ensure that they have just enough access to carry out their necessary tasks. These three principles are crucial in order to maintain customer trust, and keep the the threat of an exploit to a minimum.

In order to comply with the Payment Card Industry Data Security Standard, the database must fit a series of security requirements (as stated in (PCI DSS, 2024)). These can be seen in Table 1, along with the security measures implemented to address the requirement.

No.	Requirement	Implementation
3	Protect Stored Account Data	Sensitive information will be encrypted, and credentials will be stored as a hash
4	Protect Cardholder Data with Strong Cryptography During Transmission Over Open, Public Networks	An SSL key and certificate will be attached with the database to allow for encrypted communication
7	Restrict Access to System Components and Cardholder Data by Business Need to Know	Strict role-based access controls will restrict access to authorised users only
8	Identify Users and Authenticate Access to System Components	All users will require a password to log in
10	Log and Monitor All Access to System Components and Cardholder Data	Any queries and updates to the database will be logged and recorded in the Audit_Trail table

Table 1: A list of all relevant PCI DSS requirements and the security measures taken to align with them

2 Access Control

2.1 Roles

To control access over the data, the database will feature role-based access control (as described in Section 3.1). Here, CRUD operations will be granted or restricted to employees based on their job title, ensuring that each employee can only access the data required for their specific job, while allowing job roles to easily be updated and added for scalability. Customers will have their own role to manage their access. The four roles which have been implemented are described in Table 5.

This will be enforced using a combination of row-level security, views, and functions. Row-level security will restrict a user to only view and operate on their own records in a table. Views will be used to restrict the columns in which a user has access to. This is crucial to prevent users from having access to data which is not required for their jobs. Functions will be assigned to users which will grant update, insert, and delete features, limiting the access even further.

2.2 Row-Level Security

Row level security will be implemented to restrict the user on which records they are able to view. This is hugely important in keeping an individual's data private, and will be applied on the following tables as seen in Table 2.

Tables	Affected Role	Row-level security policy
Customers	Customers	Can only see their own information
Accounts	Customers	Can only see accounts they own
Transactions	Customers	Can only see transactions affecting them
Loans	Customers	Can only see their own active loans
Employees	Employees	Can only see their own data
Audit Trail	N/A	N/A

Table 2: Row-level security policies for each table

The figure below shows the SQL query used to implement row-level security on the "customers" table. This was repeated on each schema as described in Table 2.

```
ALTER TABLE IF EXISTS public.accounts ENABLE ROW LEVEL SECURITY;
CREATE POLICY account_rls ON public.customers FOR SELECT TO
"Customer" USING (username = CURRENT_USER);
```

Row-level security policy for "customers" table

```
CREATE POLICY loans_rls ON public.loans FOR SELECT TO "Customer"
USING ( account_id IN (
    SELECT accounts.account_id FROM accounts
    WHERE (accounts.customer_id = customer_id_from_username()))
);
```

Row-level security policy for "loans" table

2.3 Views

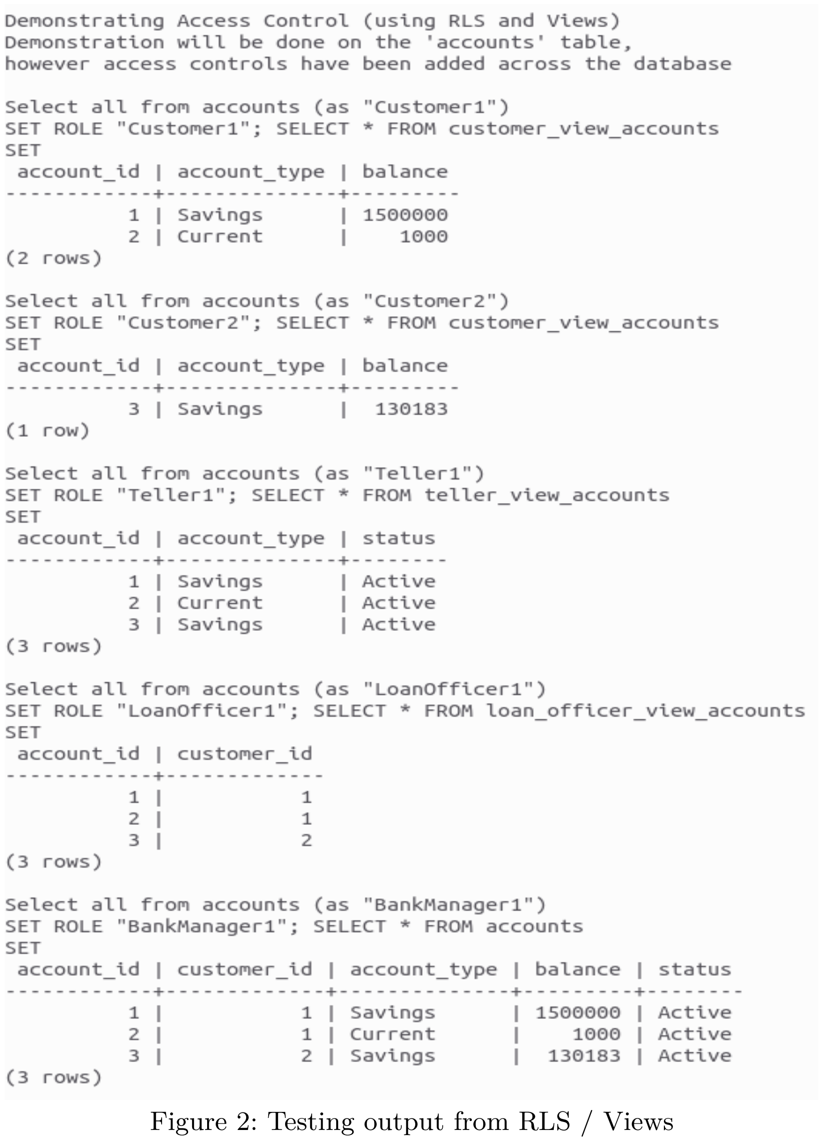
Views will be used to restrict which columns a user has access to. This has been updated from the table in the brief, so customers are able to see their own account number and account type, this will be useful for conversations with a teller in person or over the phone.

Accounts Table	Customer	Teller	Loan Officer	Bank Manager
Account_id	C, R	R	R	C, R, D
Customer_id	C	-	R	C, R, D
Account_type	C, R	R	-	C, R, U, D
Balance	C, R	-	-	C, R, D
Open_date	C	-	R	C, R, D
Status	C	R, U	-	C, R, U, D

Table 3: Role-based access to customer "accounts" table (Create, Read, Update, Delete)

2.3.1 Testing RLS and Views

The output code seen in Figure 2 shows different roles accessing the same table with RLS enabled and using their respective views.



```
Demonstrating Access Control (using RLS and Views)
Demonstration will be done on the 'accounts' table,
however access controls have been added across the database

Select all from accounts (as "Customer1")
SET ROLE "Customer1"; SELECT * FROM customer_view_accounts
SET
 account_id | account_type | balance
------------+--------------+---------
          1 | Savings      | 1500000
          2 | Current      |    1000
(2 rows)

Select all from accounts (as "Customer2")
SET ROLE "Customer2"; SELECT * FROM customer_view_accounts
SET
 account_id | account_type | balance
------------+--------------+---------
          3 | Savings      |  130183
(1 row)

Select all from accounts (as "Teller1")
SET ROLE "Teller1"; SELECT * FROM teller_view_accounts
SET
 account_id | account_type | status
------------+--------------+--------
          1 | Savings      | Active
          2 | Current      | Active
          3 | Savings      | Active
(3 rows)

Select all from accounts (as "LoanOfficer1")
SET ROLE "LoanOfficer1"; SELECT * FROM loan_officer_view_accounts
SET
 account_id | customer_id
------------+-------------
          1 |           1
          2 |           1
          3 |           2
(3 rows)

Select all from accounts (as "BankManager1")
SET ROLE "BankManager1"; SELECT * FROM accounts
SET
 account_id | customer_id | account_type | balance | status
------------+-------------+--------------+---------+--------
          1 |           1 | Savings      | 1500000 | Active
          2 |           1 | Current      |    1000 | Active
          3 |           2 | Savings      |  130183 | Active
(3 rows)
```

Figure 2: Testing output from RLS / Views

2.4 Functions

Access to write operations (like UPDATE and INSERT) would be done via the use of functions, on which users can be granted execute access based on their role. This would prevent an unauthorised user from accessing operations which they should not be allowed to perform. Table 4 shows how they would be implemented.

Role Name	Access	Members (with login)
Customer	Edit personal information Can create accounts Can request loans Can make transactions	Customer1 Customer2
Teller	Can authorise large payments	Teller1
Loan Officer	Can authorise loans	LoanOfficer1
Bank Manager	Superuser	BankManager1

Table 5: List of roles and their respective access

```
CREATE FUNCTION public.customer_id_from_username()
    RETURNS integer LANGUAGE 'plpgsql'
AS $BODY$
DECLARE
    _customer_id integer;
BEGIN
    SELECT customers.customer_id INTO _customer_id
    FROM customers WHERE customers.username = CURRENT_USER;
    IF FOUND THEN RETURN _customer_id;
    ELSE RETURN NULL;
    END IF;
END;
$BODY$;
```

A function to get a customers' user_id from the current user's username

3 Miscellaneous

3.1 Role-Based Access Control

Role-Based security is a method in which authentication is managed by roles. These roles can be given to a user to grant or revoke access to specific operations and tables within a database. This provides a structured approach to security, with clearly defined boundaries, and enforces the principle of least privilege, where users only have access to the data they require to carry out their tasks. RBAC is very easily scalable, as roles can be managed easily, and can affect a large number of users at once. Granting and revoking access to employees can be done through their job roles within the bank.

When compared with other methods, such Mandatory Access Control (MAC), Discretionary Access Control (DAC), or even individual methods, role-based access control provides the most prominent balance between control and scalability, making it the most admirable method for a business like a bank.

3.2 Minimising Risk of Unauthorised Access

3.2.1 Data-centric mitigations

It is best practice to encrypt data before it is sent to the database, this will reduce the risk of data being stolen during both transmission and storage. As application software is out of scope for this project, encryption can be applied at the database level, using built-in postgresql options, this also means that data must be encrypted during transmission, therefore an SSL certificate and key will be attached to the database to authorise connections and encrypt data during transmission. Because of this, credentials should also be hashed when they are stored in the database, although again this should be done on the application side before it reaches the database.

3.2.2 Perimeter-centric mitigations

Perimeter-centric security is focused around securing the entry-points to the database in order to prevent unauthorised access. This includes features like firewalls to block suspicious connections, using strong password, access controls, and keeping backups secured.

3.2.3 Detection Mechanisms

Detection mechanisms are also a crucial part of security. This includes auditing (as mentioned in Section 3.3), and also Intrusion Detection Systems, as these can help spot suspicious activity early, meaning that threats can be dealt with earlier.

3.3 Data Auditing

Audits are crucial when it comes to enhancing security. They help to spot suspicious activity such as data being modified and unusual access patterns, which can help detect and prevent threats early on.

They are also mandatory for regulatory compliance; PCI DSS requires businesses to log and monitor all access to a systems components and cardholder data (PCI DSS, 2024). It is also important to protect users' personal information to align with GDPR.

3.3.1 Implementation

Data auditing has been implemented via the use of Trigger Functions. Triggers are set up on a certain data in a database, and whenever they detect a change, they will call a trigger function. In this case, the trigger functions have been used to create a new record in the 'audit_trail' table whenever a record in the 'accounts' table is inserted, updated, or deleted. This new entry contains all of the changes that were made, as well as the operation used, and a timestamp of the operation. This will keep track of every operation that has happened within the 'accounts' table.

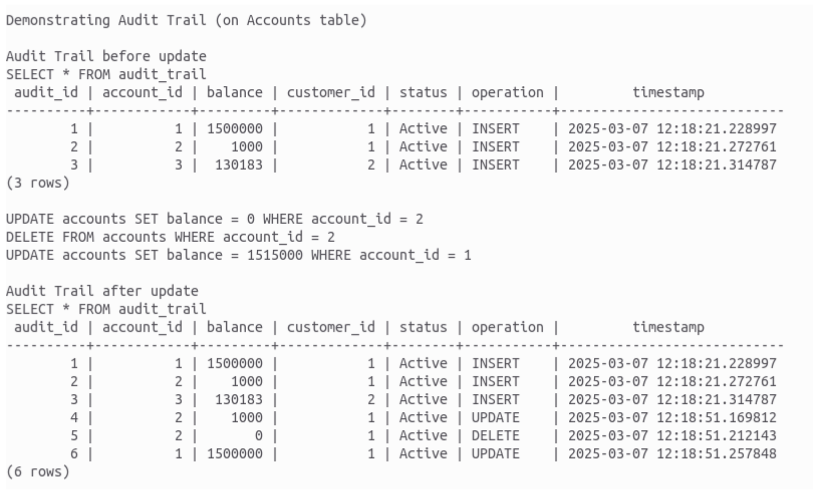
```sql
CREATE FUNCTION public.account_audit_insert()
    RETURNS trigger LANGUAGE 'plpgsql'
AS $BODY$
```

```sql
BEGIN
    INSERT INTO audit_trail (account_id, customer_id, balance,
status, operation, timestamp)
    VALUES (new.account_id, new.customer_id, new.balance,
new.status, 'INSERT', now());
    RETURN new;
END;
$BODY$;
```

Trigger Function for an INSERT operation

```sql
CREATE OR REPLACE TRIGGER account_insert
    AFTER INSERT
    ON public.accounts
    FOR EACH ROW
    EXECUTE FUNCTION public.account_audit_insert();
```

SQL code to create a Trigger after an INSERT operation



```
Demonstrating Audit Trail (on Accounts table)

Audit Trail before update
SELECT * FROM audit_trail
 audit_id | account_id | balance | customer_id | status | operation |         timestamp
----------+------------+---------+-------------+--------+-----------+----------------------------
        1 |          1 | 1500000 |           1 | Active | INSERT    | 2025-03-07 12:18:21.228997
        2 |          2 |    1000 |           1 | Active | INSERT    | 2025-03-07 12:18:21.272761
        3 |          3 |  130183 |           2 | Active | INSERT    | 2025-03-07 12:18:21.314787
(3 rows)

UPDATE accounts SET balance = 0 WHERE account_id = 2
DELETE FROM accounts WHERE account_id = 2
UPDATE accounts SET balance = 1515000 WHERE account_id = 1

Audit Trail after update
SELECT * FROM audit_trail
 audit_id | account_id | balance | customer_id | status | operation |         timestamp
----------+------------+---------+-------------+--------+-----------+----------------------------
        1 |          1 | 1500000 |           1 | Active | INSERT    | 2025-03-07 12:18:21.228997
        2 |          2 |    1000 |           1 | Active | INSERT    | 2025-03-07 12:18:21.272761
        3 |          3 |  130183 |           2 | Active | INSERT    | 2025-03-07 12:18:21.314787
        4 |          2 |    1000 |           1 | Active | UPDATE    | 2025-03-07 12:18:51.169812
        5 |          2 |       0 |           1 | Active | DELETE    | 2025-03-07 12:18:51.212143
        6 |          1 | 1500000 |           1 | Active | UPDATE    | 2025-03-07 12:18:51.257848
(6 rows)
```

Figure 3: Output from auditing trigger functions

Bibliography

UK GDPR, 2016: , Regulation (EU) 2016/679 of the European Parliament and of the Council, 2016

PCI DSS, 2024: PCI Security Standards Council, Payment Card Industry Data Security Standard, 2024