SUMMARY

1. Motivation:

- The widespread adoption of mobile banking applications such as 'YONO' and 'Payzapp' motivated us to delve deeper into database design principles, seeking to understand how these apps efficiently store and manage vast amounts of user data.

2. Our Experiences:

- **Database Development Journey:** Developing the Banking Management System database provided us with an opportunity to explore diverse domains including complex ER design, relational schema design and normalization techniques.

- **Collaborative Efforts:** Our team worked closely together throughout the project, utilising our different skills to accomplish our shared objectives. An atmosphere of originality and cooperation was built through frequent meetings and discussions about various ideas.

3. Challenges Faced:

- **ERD and Schema Complexity:** Some of the biggest challenges in designing an effective ERD and schema was making sure it could support different banking functions while maintaining scalability and data integrity.

- **Schema Normalization:** In order to transform the schema into Boyce-Codd Normal Form (BCNF), the database structure's dependencies and anomalies had to be carefully examined. In order to ensure the removal of any redundant data and the normalisation of relations, this process required detailed analysis and reorganisation.

4. Real User Inputs:

- **Engaging with Various Users:** Our process included reaching out to a few users such as banking professionals, potential end-users and individuals (even our family members). By actively seeking feedback from these groups, we gained valuable insights into their expectations and requirements, enriching the design and functionality of our Banking Management System.

- **Improvement through User Insights:** Through continuous interaction and feedback with our users including our parents and industry experts, we iteratively refined the database's features. This collaborative approach ensured that our system aligns closely with the needs and preferences of its intended users, resulting in an increased usability.

5. Extra Efforts:

- **Welcoming Complexities:** Our dedication to delivering a comprehensive Banking Management System led us to include entities such as insurance, investment, loan, fixed deposit (FD) and various banking services. Embracing these complexities allowed us to create a versatile platform capable of addressing diverse financial needs.

- **Exploring Trigger Functions for IDs:** Exploring advanced database functionalities, we tried using trigger functions to automate the generation of relevant IDs within our system. This could not only streamline the process but also ensure accuracy in ID assignment across different relations.

- **Capturing Ideas from Leading Apps:** We examined some of the online banking apps like 'YONO', 'Payzapp' and learnt about how they were working and providing services to users. After analysing the structure of these apps, we were able to make a convenient structure of how our database should look and what relations it will be containing. (Code snippet for the same is provided on the last page of the report)

- **Using Python Code Snippets for Data Insertion:** By utilising Python's capabilities, we were able to find and apply code snippets that could generate and input massive amounts of data into our database. This automation ensured our

system's reliability by speeding up the testing process and facilitating scaling. (Pseudo Code for the same is provided on the second last page of the report)

6. Learnings:

- As the project progressed, our understanding of database has become from strong to stronger. While building optimal database we learnt about,

  a) **Data Modelling:** Through the process of creating Entity-Relationship diagrams and relational schema we gained understanding of representing real-world entities and their relations in the database.

  b) **Detecting Redundancies:** We found and removed redundant data by analysing the database structure, which resulted in a more organised and efficient database architecture.

  c) **Normalisation Techniques:** By using methods like the Boyce-Codd Normal Form (BCNF), we were able eliminate redundant data and enhance data integrity by organising the database into precise and easier-to-manage relations.

  d) **Performance Optimisation:** When relations are not in Boyce-Codd Normal Form (BCNF), redundancies may arise, increasing the likelihood of data loss during join operations. By transforming relations into BCNF, these anomalies are minimized, thus contributing to overall database performance optimization.

7. Forthcoming Project Plans:

- We have plans to implement some virtual relations, triggers and stored procedures in our project.

- We intend to make a console application or web application to handle banking management system.

TOP 3 Queries based on Practical Utility

1. Find all customers who have a fixed deposit due for maturity within the next 30 days

```
SELECT FD_no, CONCAT(fname,' ',lname) AS name, maturity_date, mobile_no

FROM Fixed_deposit

NATURAL JOIN Account

NATURAL JOIN Customer

WHERE DATE(maturity_date)>=DATE_TRUNC('month',DATE '2025-04-01')

AND DATE(maturity_date)<=DATE_TRUNC('month',DATE '2025-04-01') + INTERVAL '1 month';
```

2. List all customers who have taken a loan but missed any installments before date '2024-06-15'

```
SELECT CONCAT(fname,' ',lname) AS name, loan_app_no, loan_installment_no, due_date, due_amt

FROM Loan_repayment

NATURAL JOIN Loan_application

NATURAL JOIN Account

NATURAL JOIN Customer

WHERE DATE(due_date)<DATE '2024-06-15' AND settlement_date IS NULL;
```

3. List the transaction details of a particular account for 1 week

```
SELECT *

FROM Transaction

WHERE account_no = '1749938644158'

AND DATE(date) >= DATE_TRUNC('week', DATE '2024-04-05')

AND DATE(date) < DATE_TRUNC('week', DATE '2024-04-05') + INTERVAL '1 week';
```

Pseudo Python Code for Generating Data

```
import random

# Function to generate random names

def generate_name():

    first_names = [“Akash”, “Sachin”, “first-names”, …]

    last_names = [“Ambani”, “Tendulkar”, “last-names”, …]

    return random.choice(first_names) + ' ' + random.choice(last_names)


# Function to generate random account details

def generate_account():

    return {

        'account_no': random.randint(680010100001, 680010101000),

        'holder_name': generate_name(),

        'acc_type': random.choice(['saving', 'current']),

        'available_balance': random.randint(100000, 10000000),

        'branch_code': 'INB' + str(random.randint(1, 100)).zfill(4),

        'UUID': random.randint(100000000000, 999999999999)

    }


# Generates INSERT queries for Account table

def insert_account_data(num_records):

    for _ in range(num_records):

        account_data = generate_account()

        query = f"INSERT INTO Account (account_no, holder_name, acc_type,
available_balance, branch_code, UUID) VALUES ({account_data['account_no']},
'{account_data['holder_name']}', '{account_data['acc_type']}',
{account_data['available_balance']}, '{account_data['branch_code']}',
{account_data['UUID']});"

        print(query)

insert_account_data(100)  # We can change 100 to the desired number of
records
```

Code Snippet for Generating Trigger Function

```
-- Creates a sequence to generate unique account numbers starting from 1
CREATE SEQUENCE acc_no_seq START 1;


-- Creates a trigger function to automatically generate account numbers
CREATE OR REPLACE FUNCTION generate_account_no()
RETURNS TRIGGER AS
$$
BEGIN
     -- Concatenates the prefix 'ACC’ to the value, left-padded with zeroes to ensure a consistent length of 12 (9 + 3(‘ACC’) = 12) characters
    NEW.account_no := CONCAT('ACC', LPAD(nextval('acc_no_seq')::TEXT, 9, '0'));
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;


-- Creates a trigger that executes the generate_account_no function before each insertion into the table
CREATE TRIGGER trigger_generate_account_no
BEFORE INSERT ON Account
FOR EACH ROW EXECUTE PROCEDURE generate_account_no();
```