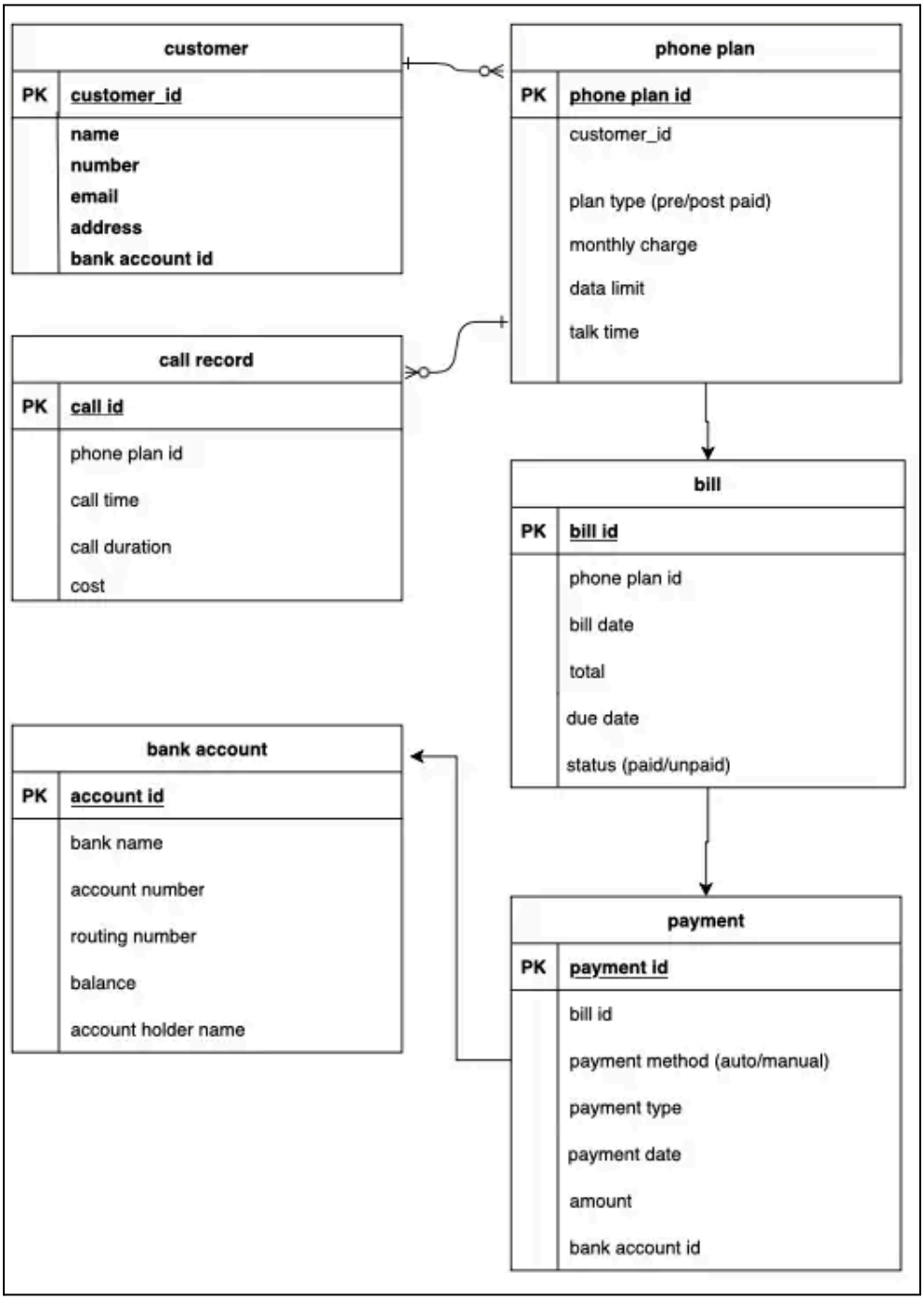
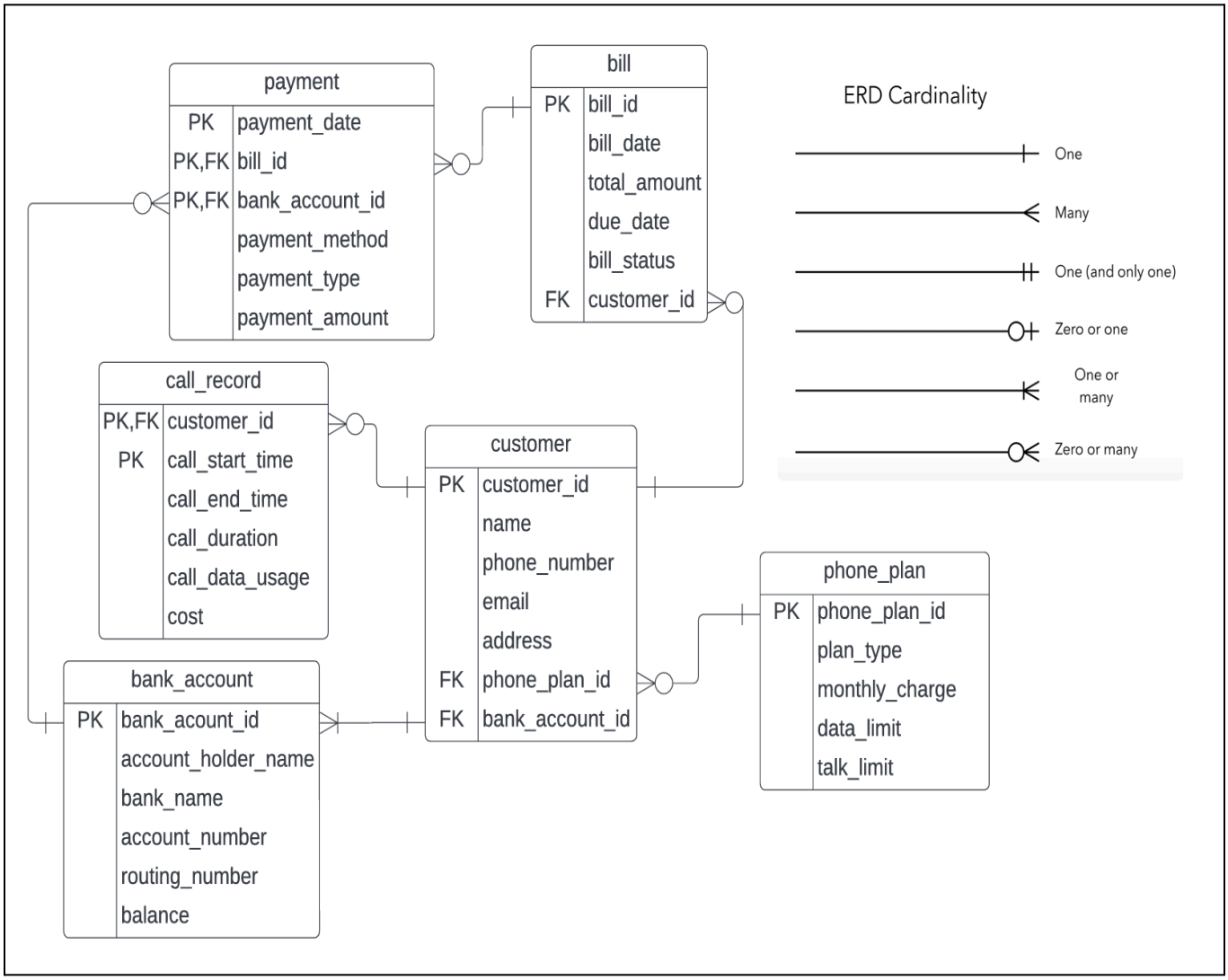
COSC3380: HW4 REPORT

ER DIAGRAMS

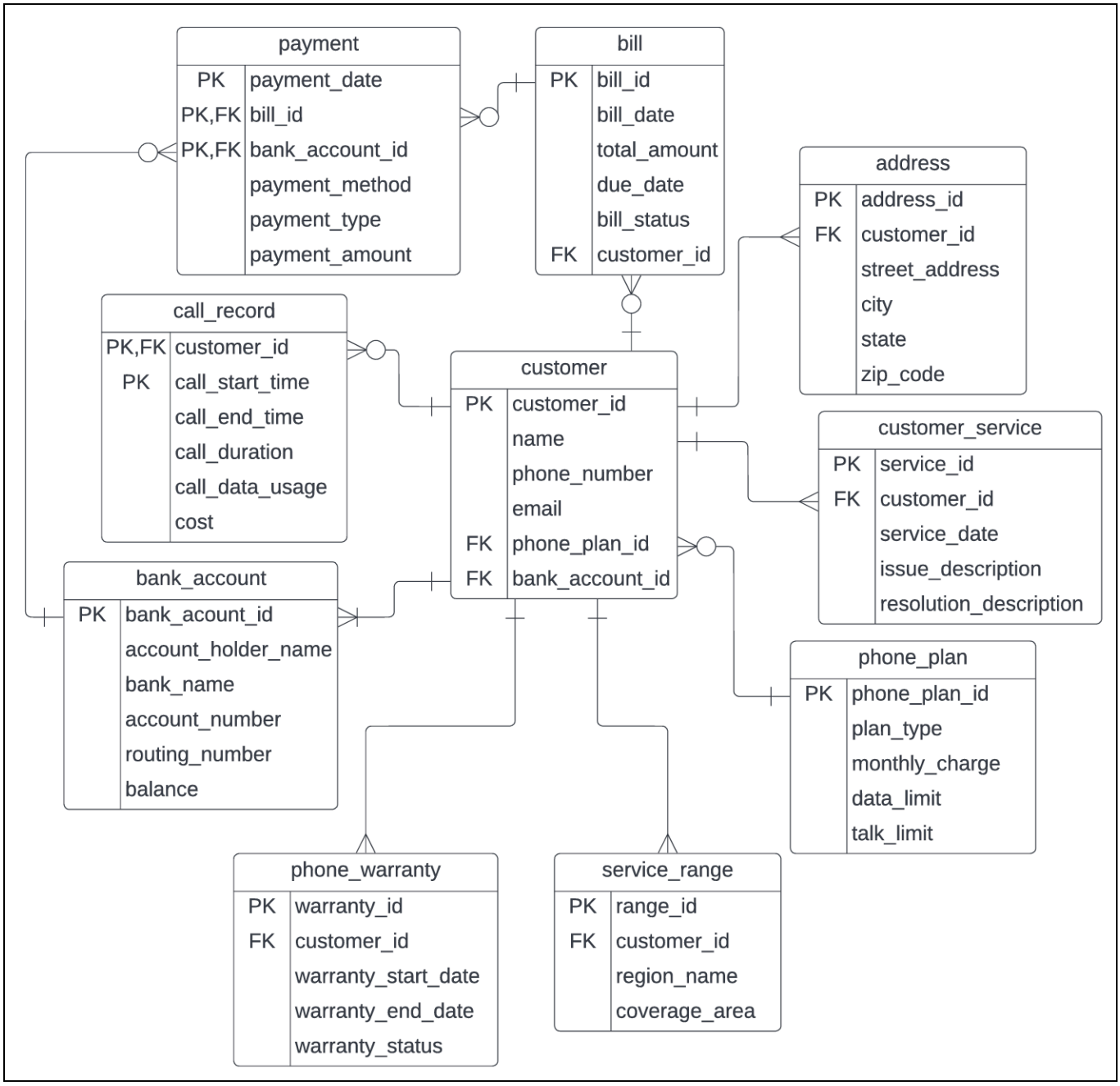
> Initial Draft ER Diagram



> Preliminary ER Diagram



> Final ER Diagram



→ Relationship from customer to phone plan: Many-to-One

- Each customer can have only one phone plan, but multiple customers can have the same phone plan this is reinforced by the phone_plan_id foreign key in the customer table.

→ Relationship from customer to bank account: One-to-Many

- Each customer can have many linked bank accounts, but each bank account belongs to one customer this is reinforced by the bank_account_id foreign key in the customer table.

→ Relationship from customer to call record: One-to-Many

- Each customer can have multiple call records, but each call record belongs to one customer this is reinforced by the customer_id foreign key in the call_record table.

→ Relationship from customer to bill: One-to-Many

- Each customer can have multiple bills, but each bill belongs to one customer this is reinforced bu the customer_id foreign key in the bill table.

→ Relationship from payment to bill: Many-to-One

- Each payment is associated with one bill, and each bill can have multiple payment this is reinforced by the bill_id foreign key in the payment table.

→ Relationship from payment to bank account: Many-to-One

- Each payment is made from one bank account, but multiple payments can be made from the same bank account this is reinforced by the bank_account_id foreign key in the payment table.

→ Relationship from customer to address: One-to-Many

- Each customer can have many addresses, but each address belongs to one customer this is reinforced by the customer_id foreign key in the address table

→ Relationship from customer to customer_service: One-to-Many

- Each customer can have many customer service occasions, but each customer_service occasion belongs to one customer this is reinforced by the customer_id foreign key in the customer_service table.

→ Relationship from customer to phone_warranty: One-to-Many

- Each customer can have many phone warranties, but each warranty belongs to one customer this is reinforced by the customer_id foreign key in the phone_warrany table.

→ Relationship from customer to service_range: One-to-Many

- Each customer can have many service ranges, but each service range belongs to one customer this is reinforced by the customer_id foreign key in the service_range table.

WEB APP SUMMARY

Our web app project is designed for efficient management of a cell phone company's customer data and operations. It allows users to create, populate, and delete database tables for customer records, phone plans, and payment details. Users can view detailed records, including customer profiles, billing summaries, total call durations, and payment histories, enabling comprehensive data insights. The app supports transaction simulations, such as bill payments and plan changes, to streamline operations. With features like billing summary generation and real-time transaction handling, the platform enhances operational efficiency and improves the overall customer experience.

➔ SQL Transaction

```
BEGIN;

DO $$
DECLARE
    v_bill_id INT;
    v_customer_id INT;
    v_bank_account_id INT;
    v_total_amount DECIMAL(10,2);
```

```
    v_balance DECIMAL(10,2);
BEGIN
    -- Get and lock an unpaid bill
    SELECT bill_id, customer_id, total_amount
    INTO v_bill_id, v_customer_id, v_total_amount
    FROM bill
    WHERE bill_status = 'Unpaid'
    LIMIT 1
    FOR UPDATE SKIP LOCKED;

    -- Exit if no unpaid bill found
    IF NOT FOUND THEN
        RAISE EXCEPTION 'No unpaid bills to process.';
    END IF;

    -- Get and lock bank account info
    SELECT ba.bank_account_id, ba.balance
    INTO v_bank_account_id, v_balance
    FROM customer c
    JOIN bank_account ba ON ba.bank_account_id = c.bank_account_id
    WHERE c.customer_id = v_customer_id
    FOR UPDATE;

    -- Check if sufficient funds are available
    IF v_balance < v_total_amount THEN
        RAISE EXCEPTION 'Insufficient funds for bill %', v_bill_id;
    END IF;

    -- Deduct amount from bank account
    UPDATE bank_account
    SET balance = balance - v_total_amount
    WHERE bank_account_id = v_bank_account_id;

    -- Mark bill as paid
    UPDATE bill
    SET bill_status = 'Paid'
    WHERE bill_id = v_bill_id;

    -- Commit the transaction
    RAISE NOTICE 'Transaction completed successfully for bill %', v_bill_id;
END $$;

-- Commit the transaction
COMMIT;
```

The transaction automates the process of paying an unpaid bill for a customer.It follows the below steps:

1. **Fetch and Lock an Unpaid Bill**
    - Selects an unpaid bill from the `bill` table, locking it for processing.
    - If no unpaid bills are available, an exception is raised, and the process stops.
2. **Fetch and Lock Bank Account Information**
    - Retrieves and locks the bank account details of the customer associated with the unpaid bill.
3. **Check for Sufficient Funds**
    - Verifies whether the bank account balance is sufficient to cover the bill's total amount.
    - If funds are insufficient, an exception is raised, and the process stops.
4. **Deduct Funds from Bank Account**
    - Updates the `bank_account` table by deducting the bill amount from the account balance.

5. **Mark Bill as Paid**
    - Updates the `bill` table to mark the selected bill as "Paid."
6. **Completion and Commit**
    - Displays a success message and commits the transaction.

The transaction ensures data consistency by using locks to prevent simultaneous processing of the same records. It also incorporates error handling for scenarios such as insufficient funds or the absence of unpaid bills.

➔ Important Queries

- **Billing Summary**: The show bill query is used to show the summary of billing information for each customer. This query includes details like the bill date, the amount due, the due date, and the status of each bill, allowing customers or administrators to view billing history or upcoming due bills for individual customers.

```
SELECT
    c.customer_id,
    c.name,
    b.bill_date,
    b.total_amount,
    b.due_date,
    b.bill_status
FROM customer c
JOIN bill b ON c.customer_id = b.customer_id
ORDER BY c.customer_id, b.bill_date;
```

- **Payment History**: Payment history query fetches details about each payment made by a customer, including information about the payment method, the bank account, and the customer who made the payment.

```
SELECT
    p.payment_method,
    p.payment_type,
    p.payment_date,
    p.payment_amount,
    ba.bank_name,
    ba.account_number,
    c.name AS customer_name
FROM payment p
JOIN bank_account ba ON p.bank_account_id = ba.bank_account_id
JOIN bill b ON p.bill_id = b.bill_id
JOIN customer c ON b.customer_id = c.customer_id
ORDER BY p.payment_date DESC;
```

- **Total call time calculation**: Query for calculating the total call time and joins tables call record and customer. This ensures that each customer's total call duration is calculated based on their call records.

```
SELECT
    RANK() OVER (ORDER BY SUM(r.call_duration) DESC) AS rank,
    c.customer_id,
```

```
    c.name,
    SUM(r.call_duration) AS total_call_time,
    ROUND(SUM(r.call_duration)::numeric / 60, 2) AS total_hours
FROM
    customer c
JOIN
    call_record r ON c.customer_id = r.customer_id
GROUP BY
    c.customer_id, c.name
ORDER BY
    total_call_time DESC;
```

Group 19 Participation

Rindy Tuy
- Transaction code, SQL queries, Table creation, Frontend, Server, README, Video

AyaanAli Lakhani
- ERD sketches/models, Transaction code, Table creation, README, Group PDF, Testing app

Cristian Herrera
- ERD sketches/models, Transaction code, Table creation, Group PDF, Testing app, SQL queries

FNU Abinanda Manoj
- ERD Models, Table creation, Transaction code, SQL queries, Group PDF, Testing app

DEMO VIDEO LINK

REFERENCES

https://dribbble.com/
https://www.w3schools.com/
https://chatgpt.com/
https://github.com/Rinzyy/Database_CS_HW4