Berkan AKIN
[illegible]

5)

1) Patient: This user is primarily interest in accessing their own health records booking appointments, and checking test results. They may also want to see their billing information or insurance coverage. Requirements for this user might include.

- Secure access to personal health data
- The ability to book, change or cancel appointments
- The option to communicate with healthcare providers.
- Access to information on medication and treatment plans.

Doctor: Doctors need comprehensive access to their patients health record. They also need to record their observations, diagnosis, and treatment plans.

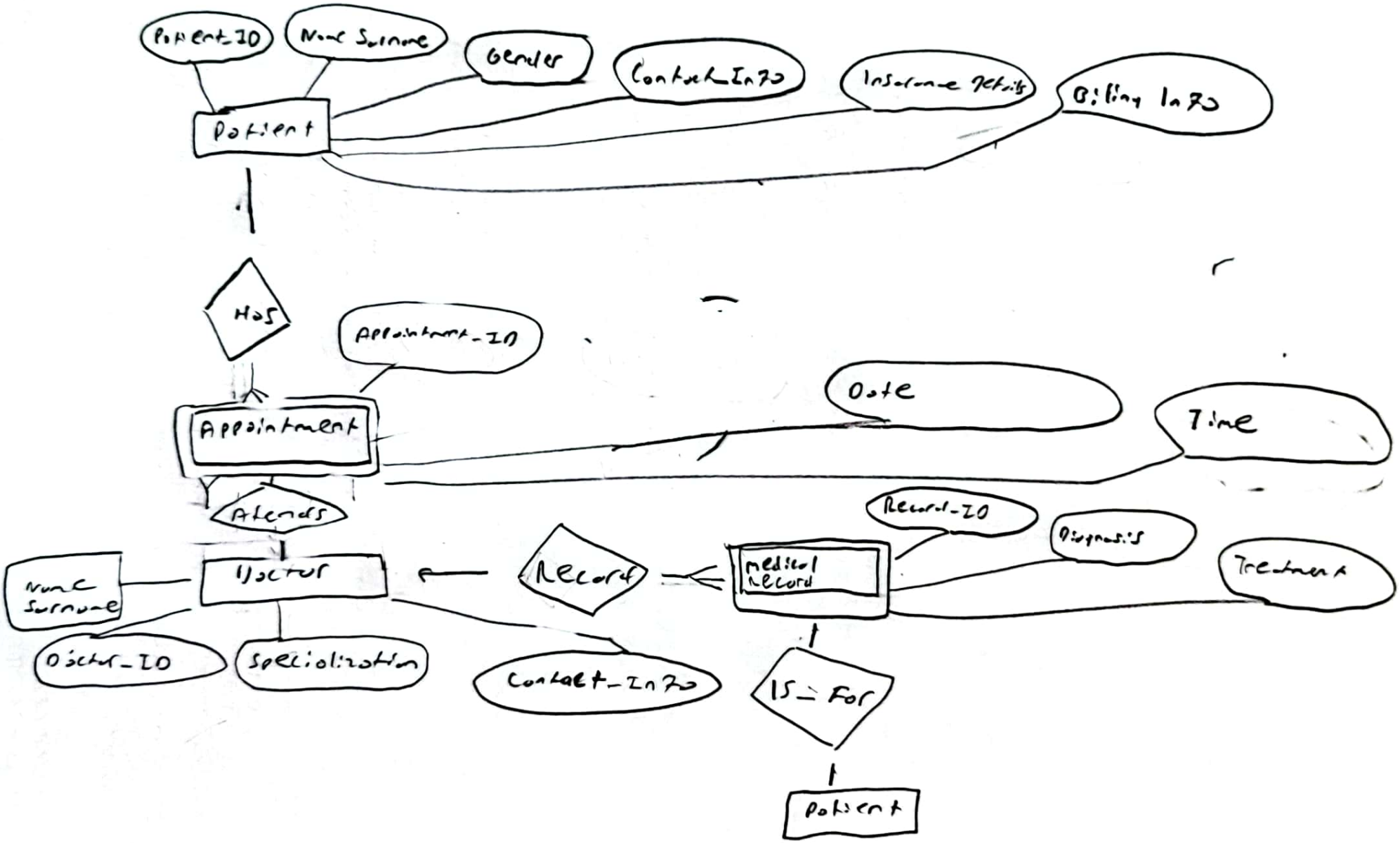
2) Doctor: Doctor need comprehensive access to their patients health records. They also need to record their observations, diagnosis, and treatment plans. other requirements

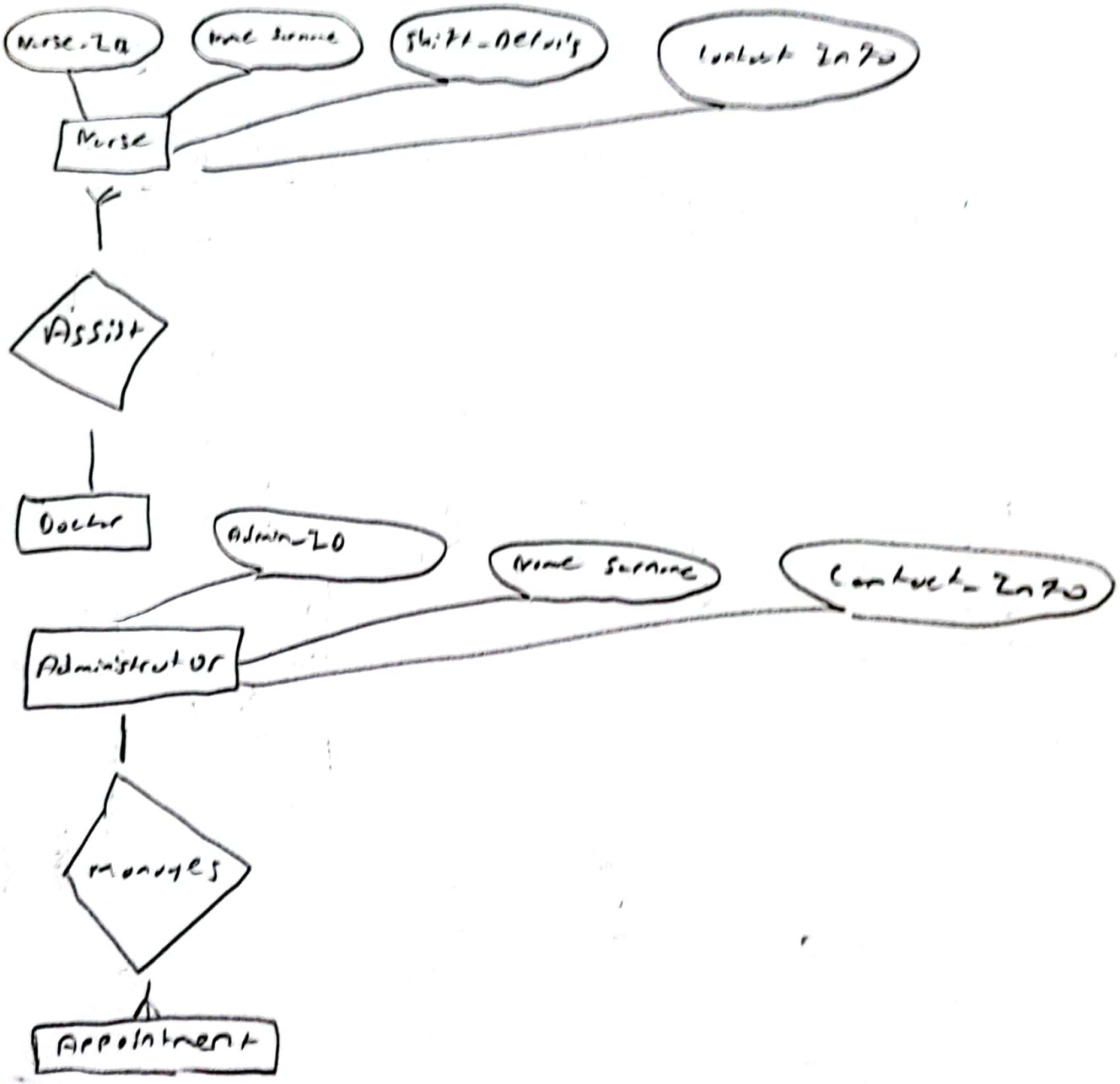
3) Nurses play a crucial role in patient care and typically need access to many of the same records as doctors, though they might not need to enter as much new data.

4) Administrator: Administrators manage the operational side of the healthcare facility. Their needs might include

- Tools for managing billing and insurance information
- The ability to manage patient data, including creating, updating and deleting record
- Tools for managing the schedules of doctors and nurses
- Access to system performance data, to optimize the operations.

ER - Diagram





C. List the functional dependencies

1. Patient_ID → 'name-surname', 'contact-info', Insurance_Details, Billing_Info

2. Doctor_ID → 'Name_Surname', specialization, 'Contact_Info'

3. Nurse_ID → 'Name_Surname', 'Shift_Detail', 'Contact_Info'

4. Admin_ID → 'Contact_Info' 'Name-Surname'

5. Appointment_ID → 'Patient_ID', Date 'Time', 'Name_Surname'

6. Appointment_ID, 'Doctor_ID' → 'Medical_Record'

7. Medical_Record → Record_ID, Diagnosis, 'Treatment'

8. Patient_ID Doctor_ID → Appointment_ID

9. Appointment_ID, Admin_ID → 'Billing_Info'

d)

d) Table : Doctor

Doctor_ID	Doctor_Name	Specialization	Contact_Info

Functional Dependencies

'Doctor_ID' -> Specialization, 'Contact_Info' -> 'Name Surname'

BCNF

For a relation to be in BCNF for any dependency A -> B A should be a super key. In our case Doctor_ID is a SuperKey

3NF

For a relation to be in 3NF, every non-prime attribute it should be fully functionally dependent on the primary key for each attribute must be primary In our case, Specialization and contact_Info are fully functionally dependent on the primary key Doctor_ID

Table : Patient

Patient_ID	Name Surname	Gender	Contact_Info	Billing_Info

Functional Dependencies

Patient_ID -> 'Name_Surname', 'Gender', 'Contact_Info', 'Insurance_details', 'Billing_Info'

BCNF

For the same reasons as above, the Patient table is in BCNF

3NF

All the non-prime attributes (Name surname, Gender, Contact_Info, Insurance details, Billing_Info) are fully functionally dependent on primary key Patient_ID. Hence, table is in 3NF.

Table: Appointment

Appointment_ID	Patient_ID	Date	Time

Functional Dependencies

Appointment_ID → Patient_ID, Date, Time

This table is also in BCNF and 3NF

BCNF

The only determinant is the primary key, so this table is in BCNF

3NF

The non-prime attributes (Patient_ID, Date, Time) are fully functionally dependent on primary key Appointment_ID. Hence, the table is in 3NF

Note Remember that achieving BCNF and 3NF also implies that the tables are 1NF and 2NF, because the rules for those forms are less strict. The aim of normalization is to reduce redundancy and risk of anomalies and these three tables meet the criteria.

E)

7.) Table Sql Function

This function retrieve all the appointments for specific doctor within a specific date range

Sql

create Function 'GetDoctorAppointments' (doctor_id INT, start_date end_date DATE)
RETURN TABLE
RETURN.
SELECT Appointment_ID, Date, Time
FROM Appointment
WHERE Doctor_ID = doctor_id
And Date Between start_date And end_date;

2SQl Function with a Loop

MySql does not natively support FOR loops in its function or procedure syntax instead, it uses a cursor in a while loop to iterate over a result set. Here's an example of a stored procedure that calculates the total number of patients a doctor has seen within a specific date range using a loop

DELIMITER $$
create PROCEDURE countPatients(IN doctor_id INT, IN start_date DATE, IN end_date DATE)

DECLARE done INT DEFAULT FALSE,
DECLARE patientCount INT DEFAULT 0;
DECLARE appointment_date DATE;
DECLARE cur Cursor FOR
SELECT DATE FROM Appointment where Doctor_ID = doctor_id
AND Date BETWEEN start_date AND end_date
Declare Continue Handler for Not Found SET done = TRUE
OPEN cur;
read_loop: Loop
Fetch cur INTO appointment_date;
IF done THEN
LEAVE read_loop;
END IF;

```
SET patient_count = patient_count + 1;
END LOOP;
CLOSE cur;
SELECT patient_count AS 'Total Patients';
END //
DELIMITER ;
```

You can call this stored procedure like CALL CountPatients(date_id, '2023-07-01', '2023-12-31');"

3. SQL Function with Input Variable Temporary Variable and Output Variable

Let's create a function that calculates the total bill for a patient. The function will take a patient's ID as input and return the total bill amount

```
DELIMITER //
CREATE FUNCTION CalculateTotalBill(p_id INT)
RETURN DECIMAL(10,2)
DETERMINISTIC
BEGIN
  DECLARE total_bill DECIMAL(10,2);
  SELECT SUM(bill_Amount) INTO total_bill
  FROM billing
  WHERE patient_ID = p_id;
  RETURN total_bill;
END //
DELIMITER
```

"You can call this function in SQL statements like this 'SELECT patient_ID, CalculateTotalBill(patient_ID) AS TotalBill FROM patients;"

7)

7) Triger using 'OLD' and NEW

In MySQL old and new row values in a triger are refered to as 'OLD.column_name' and 'NEW.column_name' respectively

Here triger that updates as doctor total appointments whenever a new appointment is added

```
Delimiter //
create TRIGER update_appointment_count
AFTER INSERT ON Appointment
For Each Row
BEGIN
 UPDATE Doctor
 SET Total_Appointments = Total_Appointments + 1
 where Doctor_ID = NEW.Doctor_ID;

END
DELIMITER ;
```

2) TRIGER with IF Condition

Here is triger that checks whether an appointment is being made within working hours. If not it raises an error

```
DELIMITER //
CREATE TRIGER check_appointment_time
BEFORE INSERT ON Appointment
FOR EACH ROW
BEGIN
 IF NEW.Time < '09:00:00' OR NEW.Time > '17:00:00' THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Cannot schedule appointment
      outside of working hours.';

  END IF;
 END //
DELIMITER ;
```

3) TRIGER with 'FOR EACH ROW'

All trigers in MySql are row-level triggers. meanig they execute 'For each ROW' Here's a triger that updates the patient's appointment date whenever a new appointment is added

```
Delimiter //
create TRIG update_last_appointment
AFTER Insert ON Appointment
For Insert on Appointment
For Each Row
BEgin
  Update patient
  SET LAST_Appointment_Date = New.Date
  where Patient_ID = NEw.Patient_ID

End //
DELIMITER ;
```

c) Triger with 'For EACH CASE'

MySql doest not support statement-level trigers or the 'For Each Case' syntax as of my knowledge cut off in september 2027. However, you can simulate a statement-level triggers by using a stored procedure or by managing state within a row-level triger.

For the sake of completeness here is an exmple of a triggers that sports. This exmple raises an exception if a doctor is deleted who still has future appointments

```
Create Trigger check_future_appointments
Before DELETE ON Doctor
For Each CASE
WHEN (SELECT COUNT(*) FROM APPOINTMENT WHERE
Doctor_ID = OLD.Doctor_ID AND DATE ) Current_DATE) >0
THEN
  RAISE Exception 'Doctor has future appointments';
END
```

9)

1. Adding a new patient scheduling their first appointment

This transaction adds a new patient to the Patient table and schedules their 1st appointment in the Appointment table

Start TRANSACTION
Insert into Patient (Patient_ID, Name_Surname, Gender_, Contact_info, Insurance_Details, Billing_info)
VALUES (new_patient_ID, new_Name_surname, new_gender, new_contact, new_insurance, new_billing)

Insert Into Appointment (Appointment_ID, Patient_ID, Doctor_ID, DATE, Time)

VALUES (new_appointment_id, new_patient_id, new_doctor_id, new_date, new_time)

Commit;

2) Transfering a patient from one doctor to another.
This transaction update all feature appointment for patient to change their doctors

Start Transaction
UPDATE Appointment
SET Doctor_ID = new_doctor_id
where Patient_ID = Patient_id_to_transfer AND Date > current_date;

UPDATE Patient
SET primary_Doctor_ID = new_doctor_id
where Patient_ID = patient_id_to_transfer;
Commit