# CSE 414 DATABASE

# SPRING 2024

# HOMEWORK 2

---

MERT GÜRŞİMŞİR

1901042646

# CONTENTS

## DESIGN

In this homework, I am going to develop a management system database related to health sector step by step. For the building of database system for health sector operations, I have used PostgreSQL.

There are going to be 7 tables for my management system which are listed below:

- Patients

```sql
CREATE TABLE Patients (
    PatientID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    DateOfBirth DATE,
    Gender CHAR(1),
    Address VARCHAR(100),
    Phone VARCHAR(15)
);
```

- Doctors

```sql
CREATE TABLE Doctors (
    DoctorID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Specialization VARCHAR(50),
    Phone VARCHAR(15)
);
```

- Appointments

```sql
CREATE TABLE Appointments (
    AppointmentID INT PRIMARY KEY,
    PatientID INT,
    DoctorID INT,
    AppointmentDate DATE,
    AppointmentTime TIME,
    Reason VARCHAR(100),
    Status VARCHAR(20),
    FOREIGN KEY (PatientID) REFERENCES Patients(PatientID),
    FOREIGN KEY (DoctorID) REFERENCES Doctors(DoctorID)
);
```

- Records

```
CREATE TABLE Records (
    RecordID INT PRIMARY KEY,
    PatientID INT,
    DoctorID INT,
    DateOfRecord DATE,
    Diagnosis VARCHAR(100),
    Treatment VARCHAR(100),
    Prescription VARCHAR(100),
    FOREIGN KEY (PatientID) REFERENCES Patients(PatientID),
    FOREIGN KEY (DoctorID) REFERENCES Doctors(DoctorID)
);
```

- Departments

```
CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY,
    Name VARCHAR(50),
    Location VARCHAR(100)
);
```

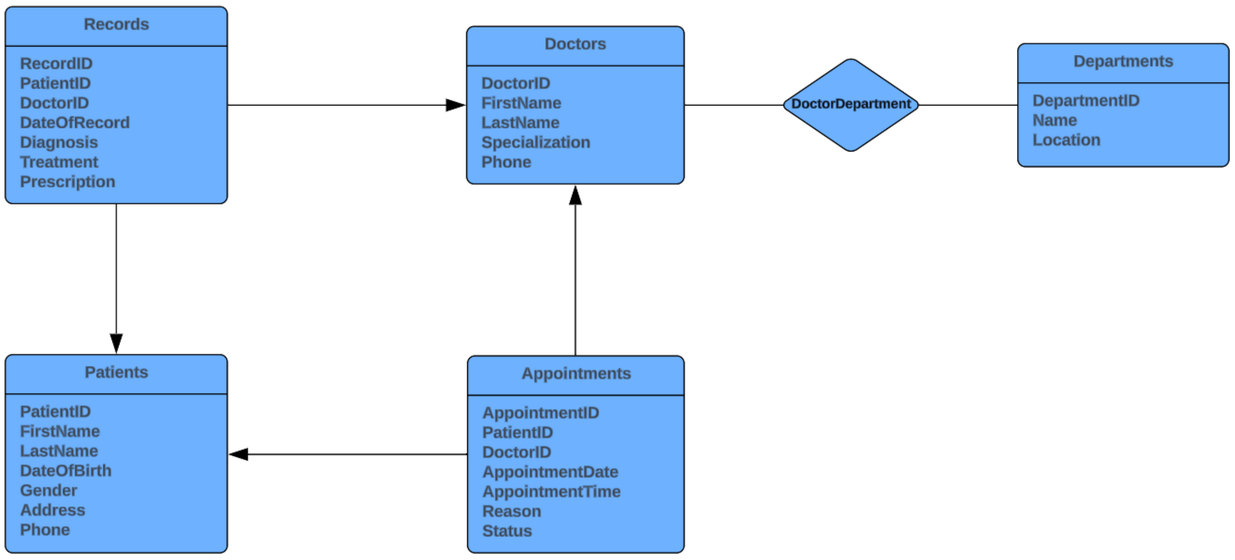
- DoctorDepartment

```
CREATE TABLE DoctorDepartment (
    DoctorID INT,
    DepartmentID INT,
    PRIMARY KEY (DoctorID, DepartmentID),
    FOREIGN KEY (DoctorID) REFERENCES Doctors(DoctorID),
    FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
);
```

Below the description of the relationships are listed:

| TABLES | CARDINALITY CONSTRAINT | DESCRIPTION |
|---|---|---|
| Doctors – Appointments | One-to-Many | A doctor can have many appointments. |
| Doctors – Records | One-to-Many | A doctor can write many records. |
| Doctors – Departments | Many-to-Many | A doctor can work in many departments and a department can have many doctors. |
| Patients – Appointments | One-to-Many | A patient can have many appointments. |
| Patients – Records | One-to-Many | A patient can have many records. |

I am representing my schema diagram below:



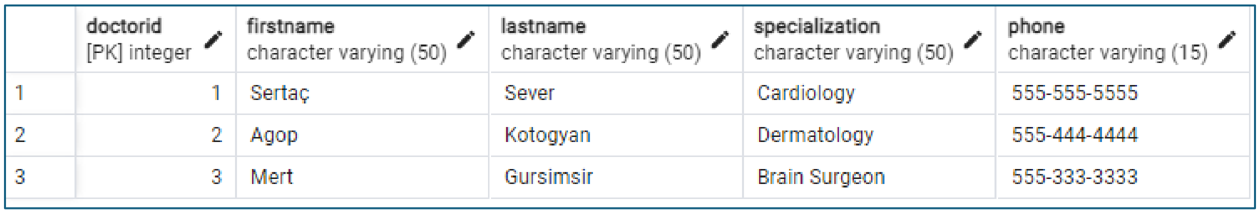
## SQL FUNCTIONS

First of all, I am adding some values to the tables related to the functions that I am going to write:

```
INSERT INTO Doctors (DoctorID, FirstName, LastName, Specialization, Phone)
VALUES
    (1, 'Sertaç', 'Sever', 'Cardiology', '555-555-5555'),
    (2, 'Agop', 'Kotogyan', 'Dermatology', '555-444-4444'),
    (3, 'Mert', 'Gursimsir', 'Brain Surgeon', '555-333-3333');

INSERT INTO Patients (PatientID, FirstName, LastName, DateOfBirth, Gender, Address, Phone)
VALUES
    (1, 'Stephen', 'King', '1947-09-21', 'M', 'Suite 419 4729 Lemke Plains, East Audria, ME 84366', '532-532-5353'),
    (2, 'Dan', 'Brown', '1964-06-22', 'M', 'Apt. 102 556 Feil Lake, Zoilatown, ID 56072-1825', '532-532-5252');

INSERT INTO Appointments (AppointmentID, PatientID, DoctorID, AppointmentDate, AppointmentTime, Reason, Status)
VALUES
    (1, 1, 3, '2024-06-30', '09:00', 'Check-up', 'Scheduled'),
    (2, 1, 2, '2024-06-30', '10:00', 'Check-up', 'Scheduled'),
    (3, 2, 1, '2024-06-30', '11:00', 'Check-up', 'Scheduled'),
    (4, 2, 3, '2024-06-30', '12:00', 'Check-up', 'Scheduled');
```

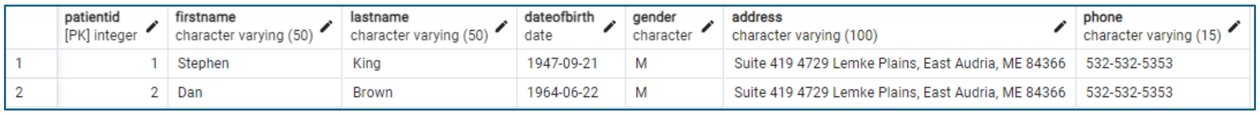
These tables look like these now:

Doctors



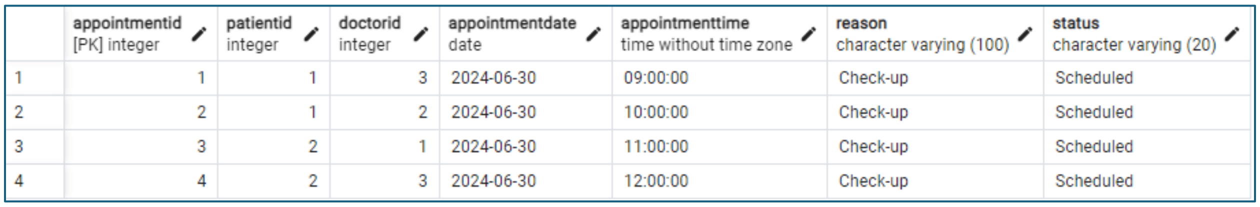
| | doctorid [PK] integer | firstname character varying (50) | lastname character varying (50) | specialization character varying (50) | phone character varying (15) |
|---|---|---|---|---|---|
| 1 | 1 | Sertaç | Sever | Cardiology | 555-555-5555 |
| 2 | 2 | Agop | Kotogyan | Dermatology | 555-444-4444 |
| 3 | 3 | Mert | Gursimsir | Brain Surgeon | 555-333-3333 |

Patients



| | patientid [PK] integer | firstname character varying (50) | lastname character varying (50) | dateofbirth date | gender character | address character varying (100) | phone character varying (15) |
|---|---|---|---|---|---|---|---|
| 1 | 1 | Stephen | King | 1947-09-21 | M | Suite 419 4729 Lemke Plains, East Audria, ME 84366 | 532-532-5353 |
| 2 | 2 | Dan | Brown | 1964-06-22 | M | Suite 419 4729 Lemke Plains, East Audria, ME 84366 | 532-532-5353 |

Appointments

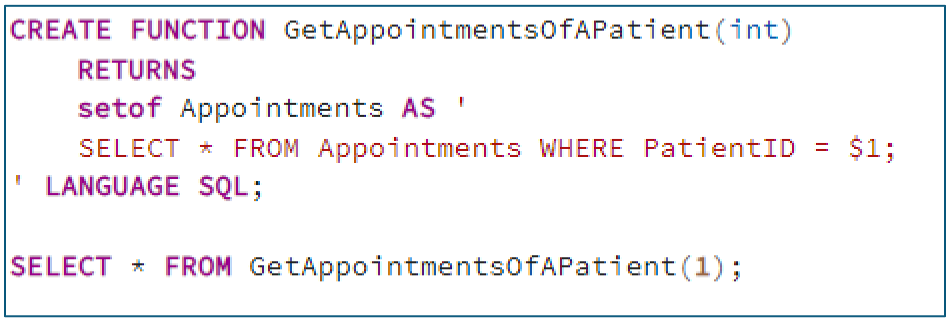


| | appointmentid [PK] integer | patientid integer | doctorid integer | appointmentdate date | appointmenttime time without time zone | reason character varying (100) | status character varying (20) |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 3 | 2024-06-30 | 09:00:00 | Check-up | Scheduled |
| 2 | 2 | 1 | 2 | 2024-06-30 | 10:00:00 | Check-up | Scheduled |
| 3 | 3 | 2 | 1 | 2024-06-30 | 11:00:00 | Check-up | Scheduled |
| 4 | 4 | 2 | 3 | 2024-06-30 | 12:00:00 | Check-up | Scheduled |

Now I am going to write functions needed for the management system one by one.

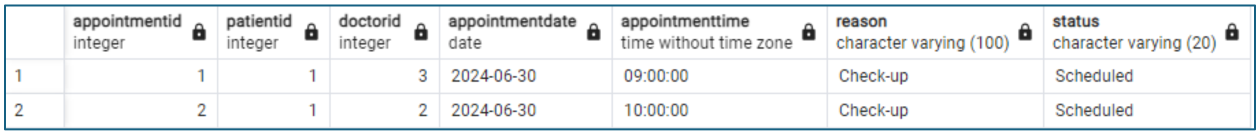
## Table SQL Function

I want to write a table SQL function that gets the ID of a patient and returns the appointments of that patient. I have written a function that is below and called it as you can see:



```
CREATE FUNCTION GetAppointmentsOfAPatient(int)
    RETURNS
    setof Appointments AS '
    SELECT * FROM Appointments WHERE PatientID = $1;
' LANGUAGE SQL;

SELECT * FROM GetAppointmentsOfAPatient(1);
```

The result is this:



| | appointmentid integer | patientid integer | doctorid integer | appointmentdate date | appointmenttime time without time zone | reason character varying (100) | status character varying (20) |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 3 | 2024-06-30 | 09:00:00 | Check-up | Scheduled |
| 2 | 2 | 1 | 2 | 2024-06-30 | 10:00:00 | Check-up | Scheduled |

## For Loop Function

I have written an SQL for loop function that loops through all the appointments and writes down the information about each of the appointments. The function is like this:

```
CREATE OR REPLACE FUNCTION loopAppointments()
RETURNS VOID AS $$
    DECLARE
    appointment RECORD;
BEGIN
    FOR appointment IN (SELECT AppointmentDate, DoctorID, PatientID FROM Appointments)
    LOOP
        RAISE NOTICE 'There is appointment on % for doctor with ID % by patient %',    appointment.AppointmentDate,
                                                                                        appointment.DoctorID,
                                                                                        appointment.PatientID;

    END LOOP;
END;
$$ LANGUAGE PLPGSQL;
```

I have called it as follows:

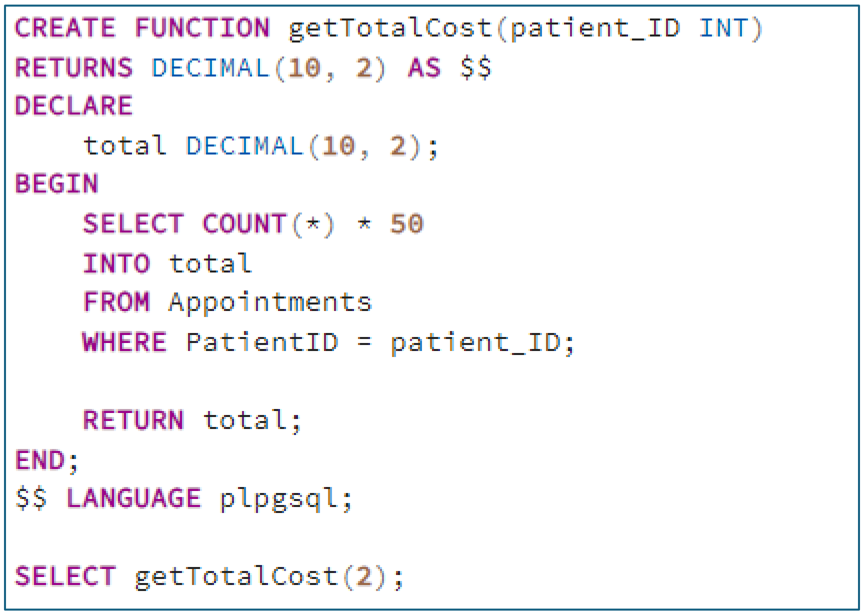
```
DO $$
BEGIN
    PERFORM loopAppointments();
END;
$$;
```

As a result, I got the following:

```
NOTICE:  There is appointment on 2024-06-30 for doctor with ID 3 by patient 1
NOTICE:  There is appointment on 2024-06-30 for doctor with ID 2 by patient 1
NOTICE:  There is appointment on 2024-06-30 for doctor with ID 1 by patient 2
NOTICE:  There is appointment on 2024-06-30 for doctor with ID 3 by patient 2
```

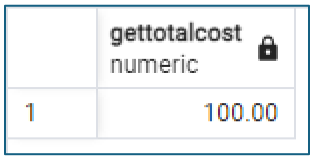
## Function with Variable

I have written a function that does the following: takes the patient ID as input, use a temporary variable that is total (means the total cost), calculates how many times a patient visits the hospital and multiply by 50 (each appointment costs 50 dollars). Then I have called the function at the bottom.



```
CREATE FUNCTION getTotalCost(patient_ID INT)
RETURNS DECIMAL(10, 2) AS $$
DECLARE
    total DECIMAL(10, 2);
BEGIN
    SELECT COUNT(*) * 50
    INTO total
    FROM Appointments
    WHERE PatientID = patient_ID;

    RETURN total;
END;
$$ LANGUAGE plpgsql;

SELECT getTotalCost(2);
```

The result is this:



|   | gettotalcost numeric |
|---|---|
| 1 | 100.00 |

# TRIGGERS

## Trigger with “OLD” and “NEW”

In PostgreSQL, there is no explicit definition such as “referencing old row as” and “referencing new row as”. The keywords OLD and NEW can be directly used to access the rows.

Therefore, I have used OLD and NEW in my trigger function.

For this trigger, I have created another table called AppointmentChanges. This table logs the appointments when their status have been changed and when they have been changed.

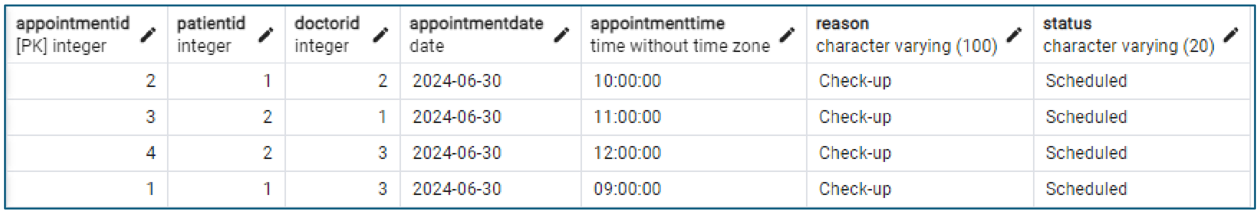
```
CREATE TABLE IF NOT EXISTS AppointmentChanges (
    ChangeID SERIAL PRIMARY KEY,
    AppointmentID INT,
    OldStatus VARCHAR(20),
    NewStatus VARCHAR(20),
    ChangeDate TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Trigger goes as follows:

```
CREATE OR REPLACE FUNCTION logAppointmentChange()
RETURNS TRIGGER AS $$
BEGIN
    IF OLD.Status IS DISTINCT FROM NEW.Status THEN
        INSERT INTO AppointmentChanges(AppointmentID, OldStatus, NewStatus)
        VALUES (NEW.AppointmentID, OLD.Status, NEW.Status);
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER logChanges
AFTER UPDATE ON Appointments
FOR EACH ROW
EXECUTE FUNCTION logAppointmentChange();
```

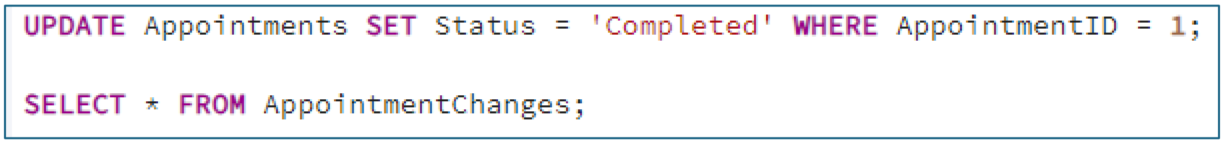
I am accessing the old row with OLD and new row with NEW.

Current Appointments table is as follows:



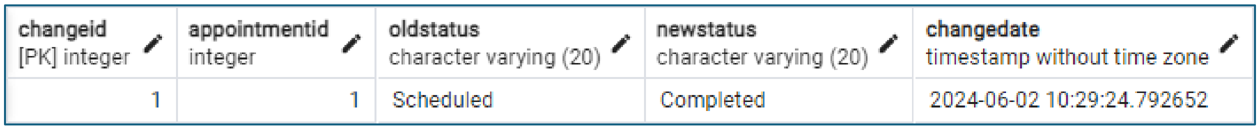
| appointmentid [PK] integer | patientid integer | doctorid integer | appointmentdate date | appointmenttime time without time zone | reason character varying (100) | status character varying (20) |
|---|---|---|---|---|---|---|
| 2 | 1 | 2 | 2024-06-30 | 10:00:00 | Check-up | Scheduled |
| 3 | 2 | 1 | 2024-06-30 | 11:00:00 | Check-up | Scheduled |
| 4 | 2 | 3 | 2024-06-30 | 12:00:00 | Check-up | Scheduled |
| 1 | 1 | 3 | 2024-06-30 | 09:00:00 | Check-up | Scheduled |

Now assume the appointment with ID 1 has been cancelled. In that case this situation is going to be logged into the AppointmentChanges table. The query:



```
UPDATE Appointments SET Status = 'Completed' WHERE AppointmentID = 1;

SELECT * FROM AppointmentChanges;
```

At the end, AppointmentChanges table looks like this:



| changeid [PK] integer | appointmentid integer | oldstatus character varying (20) | newstatus character varying (20) | changedate timestamp without time zone |
|---|---|---|---|---|
| 1 | 1 | Scheduled | Completed | 2024-06-02 10:29:24.792652 |

*NOTE*: In PostgreSQL, triggers are written in the PL/pgSQL language, which requires the use of the 'CREATE FUNCTION statement to define the trigger function. Therefore OLD and NEW keywords are used directly inside this trigger function.

## Trigger with “when” and “if”

In this trigger, I am aiming to prevent updating doctors’ specializations to a unvalid value. I have created a whitelist for the specializations and if a doctor’s specialization is going to be updated, this must be one of the specializations at the whitelist. Also the specialization must be different from the old specialization. With this way, trigger is triggered when the update is on specialization.

My trigger is as follows:

```
CREATE OR REPLACE FUNCTION checkSpecializationWhitelist()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.Specialization NOT IN ('Cardiology', 'Neurology', 'Pediatrics', 'Dermatology', 'Brain Surgeon') THEN
        RAISE EXCEPTION 'Doctor cannot change his/her specialization to given value. % is not in list.', NEW.Specialization;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER doctorUpdate
BEFORE UPDATE ON Doctors
FOR EACH ROW
WHEN (NEW.Specialization IS DISTINCT FROM OLD.Specialization)
EXECUTE FUNCTION checkSpecializationWhitelist();
```

When I try to change the specialization of the doctor to a new value that is not in the whitelist, I expect from program to give an error.

Let's update the doctor's specialization, whose ID is 1, to 'Anesthesiology' which is not in the whitelist.

```
UPDATE Doctors SET Specialization = 'Anesthesiology' WHERE DoctorID = 1;
```

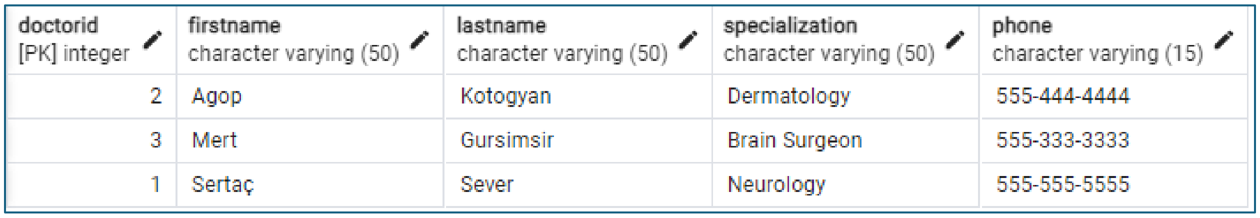
Result is as follows:

```
ERROR:  Doctor cannot change his/her specialization to given value. Anesthesiology is not in list.
CONTEXT:  PL/pgSQL function checkspecializationwhitelist() line 4 at RAISE
```

But if we try to change the specialization to a value that is in whitelist, update is going to be successful.

```
UPDATE Doctors SET Specialization = 'Neurology' WHERE DoctorID = 1;
SELECT * FROM Doctors;
```

Then we  can see specialization is changed:



| doctorid [PK] integer | firstname character varying (50) | lastname character varying (50) | specialization character varying (50) | phone character varying (15) |
|---|---|---|---|---|
| 2 | Agop | Kotogyan | Dermatology | 555-444-4444 |
| 3 | Mert | Gursimsir | Brain Surgeon | 555-333-3333 |
| 1 | Sertaç | Sever | Neurology | 555-555-5555 |

## Trigger with “for each row”

Actually, with the former 2 triggers, we have used the “for each row”. Although in this trigger, the importance of it is going to be mentioned and understood deeply.

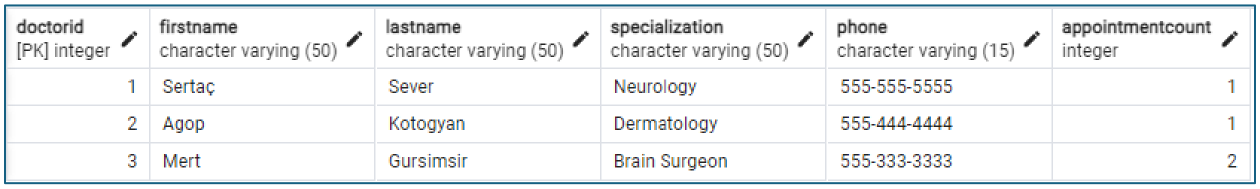
In this trigger, I am aiming to do this: Firstly I am going to add a column to the AppointmentCount. Without the trigger, when a new appointment occurs we have to increment the AppointmentCount of the doctor by hand. This is not an efficient approach.

```
ALTER TABLE Doctors ADD COLUMN AppointmentCount INT DEFAULT 0;
```

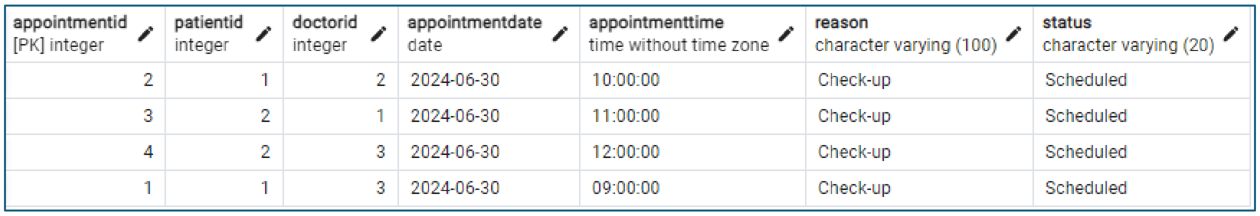
Now my aim is this: When now appointment is inserted into the Appointments table I am going to increment the related doctor’s AppointmentCount by 1 and in case of deletion of an appointment, I am going to decrease the number of AppointmentCount by 1. Also if the status of the appointment is updated, so it is not “Scheduled”, then we can decrease the AppointmentCount by 1 also.

```
CREATE OR REPLACE FUNCTION updatingAppointmentCount()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        UPDATE Doctors SET AppointmentCount = AppointmentCount + 1 WHERE DoctorID = NEW.DoctorID;
    ELSIF TG_OP = 'DELETE' THEN
        UPDATE Doctors SET AppointmentCount = AppointmentCount - 1 WHERE DoctorID = OLD.DoctorID;
    ELSIF TG_OP = 'UPDATE' THEN
        IF OLD.Status = 'Scheduled' AND NEW.Status <> 'Scheduled' THEN
            UPDATE Doctors SET AppointmentCount = AppointmentCount - 1 WHERE DoctorID = NEW.DoctorID;
        END IF;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER appointmentChange
AFTER INSERT OR DELETE OR UPDATE OF Status ON Appointments
FOR EACH ROW
EXECUTE FUNCTION updatingAppointmentCount();
```

Let’s see how the Doctors and Appointments tables looks for now:



| doctorid [PK] integer | firstname character varying (50) | lastname character varying (50) | specialization character varying (50) | phone character varying (15) | appointmentcount integer |
|---|---|---|---|---|---|
| 1 | Sertaç | Sever | Neurology | 555-555-5555 | 1 |
| 2 | Agop | Kotogyan | Dermatology | 555-444-4444 | 1 |
| 3 | Mert | Gursimsir | Brain Surgeon | 555-333-3333 | 2 |

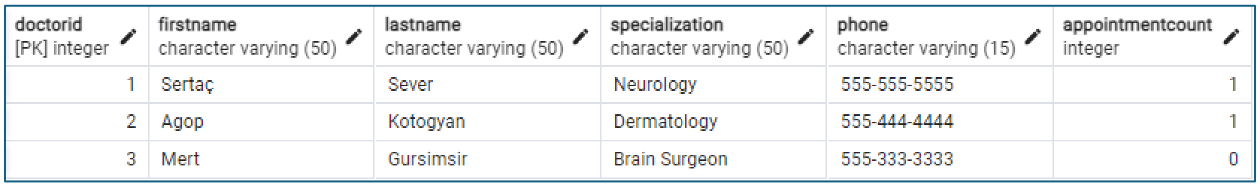


| appointmentid [PK] integer | patientid integer | doctorid integer | appointmentdate date | appointmenttime time without time zone | reason character varying (100) | status character varying (20) |
|---|---|---|---|---|---|---|
| 2 | 1 | 2 | 2024-06-30 | 10:00:00 | Check-up | Scheduled |
| 3 | 2 | 1 | 2024-06-30 | 11:00:00 | Check-up | Scheduled |
| 4 | 2 | 3 | 2024-06-30 | 12:00:00 | Check-up | Scheduled |
| 1 | 1 | 3 | 2024-06-30 | 09:00:00 | Check-up | Scheduled |

Let's cancel the appointments of doctor with ID 3. In this case AppointmentCount of Mert should be 0 because there is no appointment for him:

```
UPDATE Appointments SET Status = 'Canceled' WHERE DoctorID = 3;

SELECT * FROM Doctors;
```

The result is as follows:



| doctorid [PK] integer | firstname character varying (50) | lastname character varying (50) | specialization character varying (50) | phone character varying (15) | appointmentcount integer |
|---|---|---|---|---|---|
| 1 | Sertaç | Sever | Neurology | 555-555-5555 | 1 |
| 2 | Agop | Kotogyan | Dermatology | 555-444-4444 | 1 |
| 3 | Mert | Gursimsir | Brain Surgeon | 555-333-3333 | 0 |

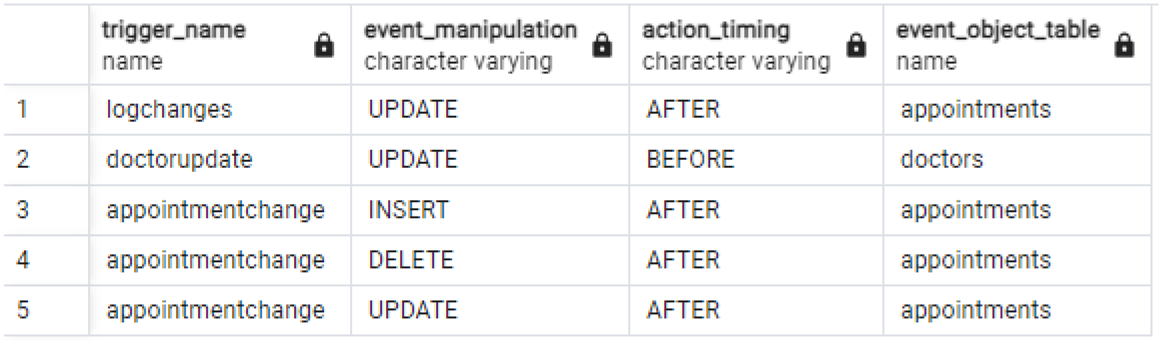
Thanks to "for each row" used in the trigger, it is executed for each row. So here 2 lines are updated at appointments and therefore trigger has executed 2 times.

## Drop a Trigger & Show Triggers

Firstly, let's see the triggers. I want to see them with name, event object table, event manipulation (update, insert, delete, etc.), and action timing (before or after):

```
SELECT trigger_name, event_manipulation, action_timing, event_object_table FROM information_schema.triggers;
```

Result:

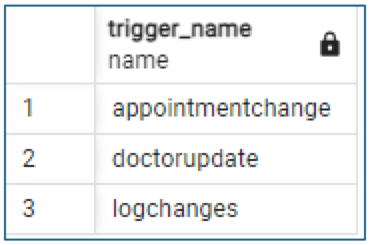


| | trigger_name name | event_manipulation character varying | action_timing character varying | event_object_table name |
|---|---|---|---|---|
| 1 | logchanges | UPDATE | AFTER | appointments |
| 2 | doctorupdate | UPDATE | BEFORE | doctors |
| 3 | appointmentchange | INSERT | AFTER | appointments |
| 4 | appointmentchange | DELETE | AFTER | appointments |
| 5 | appointmentchange | UPDATE | AFTER | appointments |

We can also see only the names of the triggers:

```
SELECT DISTINCT(trigger_name) FROM information_schema.triggers;
```

Result:

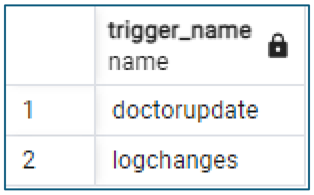


| | trigger_name<br>name |
|---|---|
| 1 | appointmentchange |
| 2 | doctorupdate |
| 3 | logchanges |

Now let's drop some triggers. Firstly, drop the appointmentchange trigger and see the triggers again:

```
DROP TRIGGER IF EXISTS appointmentChange ON Appointments;

SELECT DISTINCT(trigger_name) FROM information_schema.triggers;
```

Result:

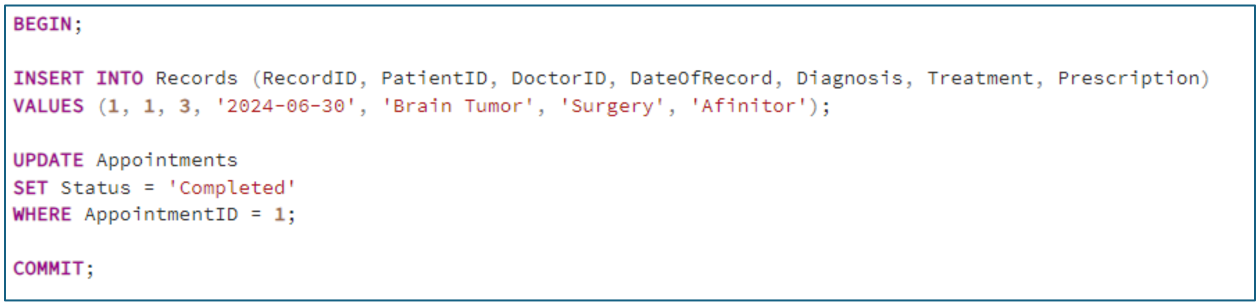


| | trigger_name<br>name |
|---|---|
| 1 | doctorupdate |
| 2 | logchanges |

# TRANSACTIONS

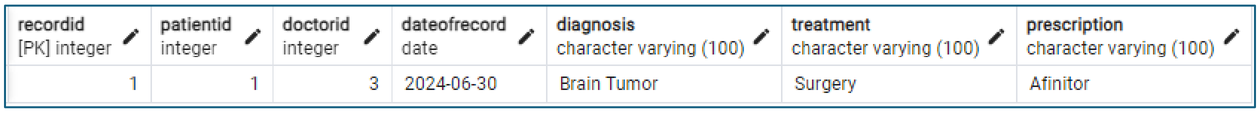
## First Atomic Transcation

With this transaction, I am aiming to inserting a new record to Records table and updating the status of an appointment to “Completed”. These both operations should be succeed together or fail together.



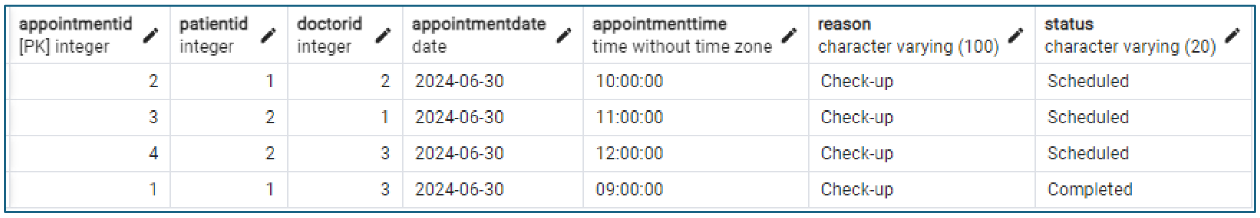
```
BEGIN;

INSERT INTO Records (RecordID, PatientID, DoctorID, DateOfRecord, Diagnosis, Treatment, Prescription)
VALUES (1, 1, 3, '2024-06-30', 'Brain Tumor', 'Surgery', 'Afinitor');

UPDATE Appointments
SET Status = 'Completed'
WHERE AppointmentID = 1;

COMMIT;
```

Records table after transaction:



| recordid [PK] integer | patientid integer | doctorid integer | dateofrecord date | diagnosis character varying (100) | treatment character varying (100) | prescription character varying (100) |
|---|---|---|---|---|---|---|
| 1 | 1 | 3 | 2024-06-30 | Brain Tumor | Surgery | Afinitor |

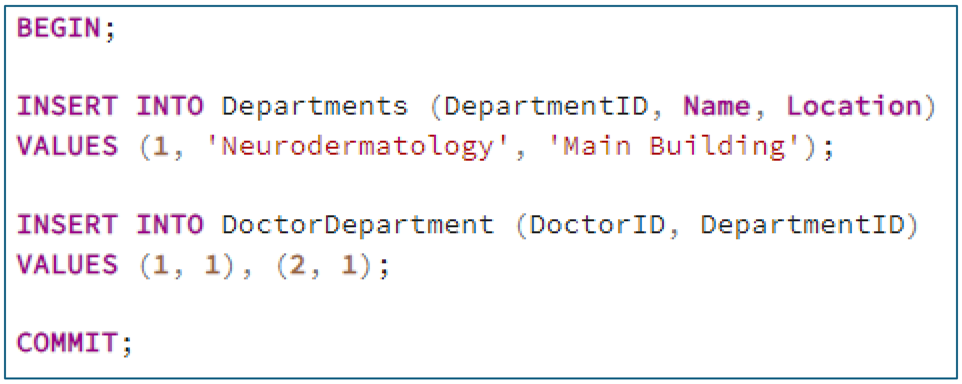
Appointments table after transaction:



| appointmentid [PK] integer | patientid integer | doctorid integer | appointmentdate date | appointmenttime time without time zone | reason character varying (100) | status character varying (20) |
|---|---|---|---|---|---|---|
| 2 | 1 | 2 | 2024-06-30 | 10:00:00 | Check-up | Scheduled |
| 3 | 2 | 1 | 2024-06-30 | 11:00:00 | Check-up | Scheduled |
| 4 | 2 | 3 | 2024-06-30 | 12:00:00 | Check-up | Scheduled |
| 1 | 1 | 3 | 2024-06-30 | 09:00:00 | Check-up | Completed |

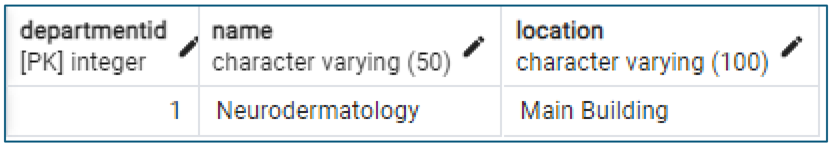
## Second Atomic Transaction

In this transaction, I am aiming to adding a new department and assigning doctors to this department. These two operations should be succeed together or fail together.



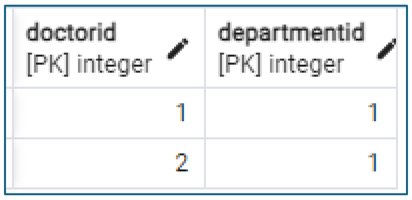
```
BEGIN;

INSERT INTO Departments (DepartmentID, Name, Location)
VALUES (1, 'Neurodermatology', 'Main Building');

INSERT INTO DoctorDepartment (DoctorID, DepartmentID)
VALUES (1, 1), (2, 1);

COMMIT;
```

Departments table after transaction:



| departmentid [PK] integer | name character varying (50) | location character varying (100) |
|---|---|---|
| 1 | Neurodermatology | Main Building |

DoctorDepartment table after transaction:



| doctorid [PK] integer | departmentid [PK] integer |
|---|---|
| 1 | 1 |
| 2 | 1 |