Cinema management system

Shani halali – 211523261

Ori katz – 314632449

System Description:

The subject we chose is the "Shani & Ori" Cinema Management System.

Our cinema has a unique ID code and an address.
The cinema employs staff members, including ushers and ticket sellers.
Employee Attributes

ID

Name

Gender

Date of Birth

Usher-Specific Data:
Assisted Customers: Tracks the number of customers the usher has helped during screenings.

Ticket Seller-Specific Data:
Amount of Sales: Records the total number of ticket sales made by the ticket seller.

The cinema includes multiple cinema halls.

Each **cinema hall** has the following attributes:

ID

Number of Rows

Number of Chairs

Each hall has a dedicated usher who is assigned exclusively to that hall.

The cinema screens a variety of movies.

Each **movie** has the following attributes:

ID

Name

Genre

Rating

Duration

Each **screening** has:

ID

Date

Time

Each screening shows a specific movie in a specific hall.

Cinema viewers can attend one or more screenings, and each screening may have multiple viewers.

We have the option to buy tickets.

Each viewer can purchase one or multiple tickets, and each screening has many tickets available.

Each **ticket** has:

ID

Price

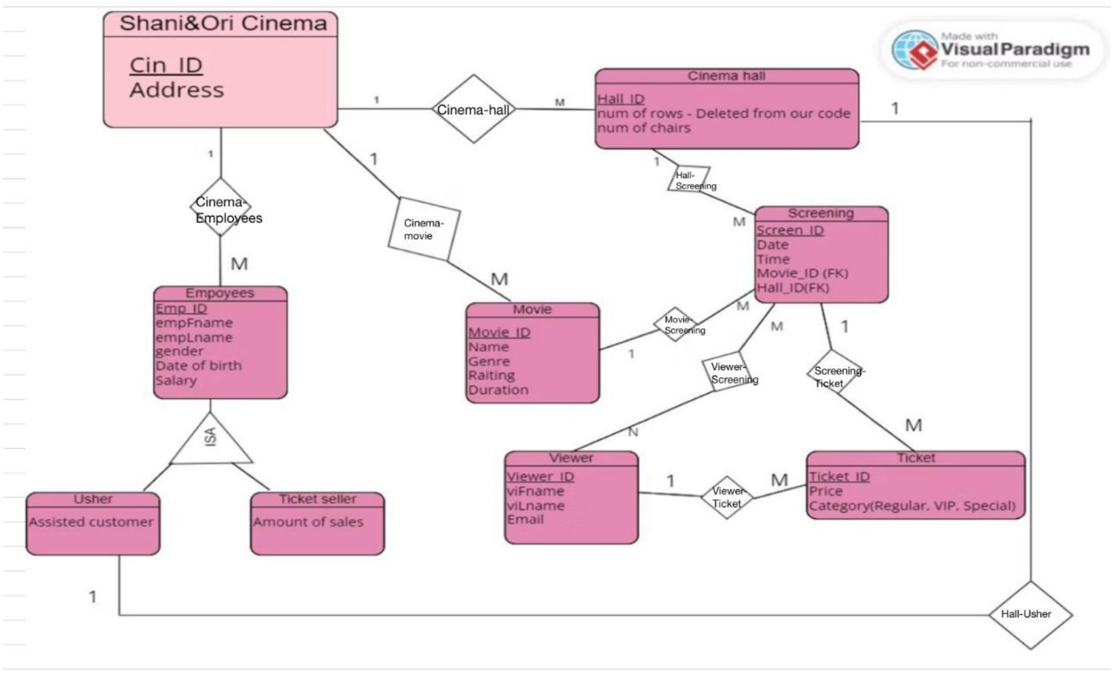
Category (Regular, VIP, Special)

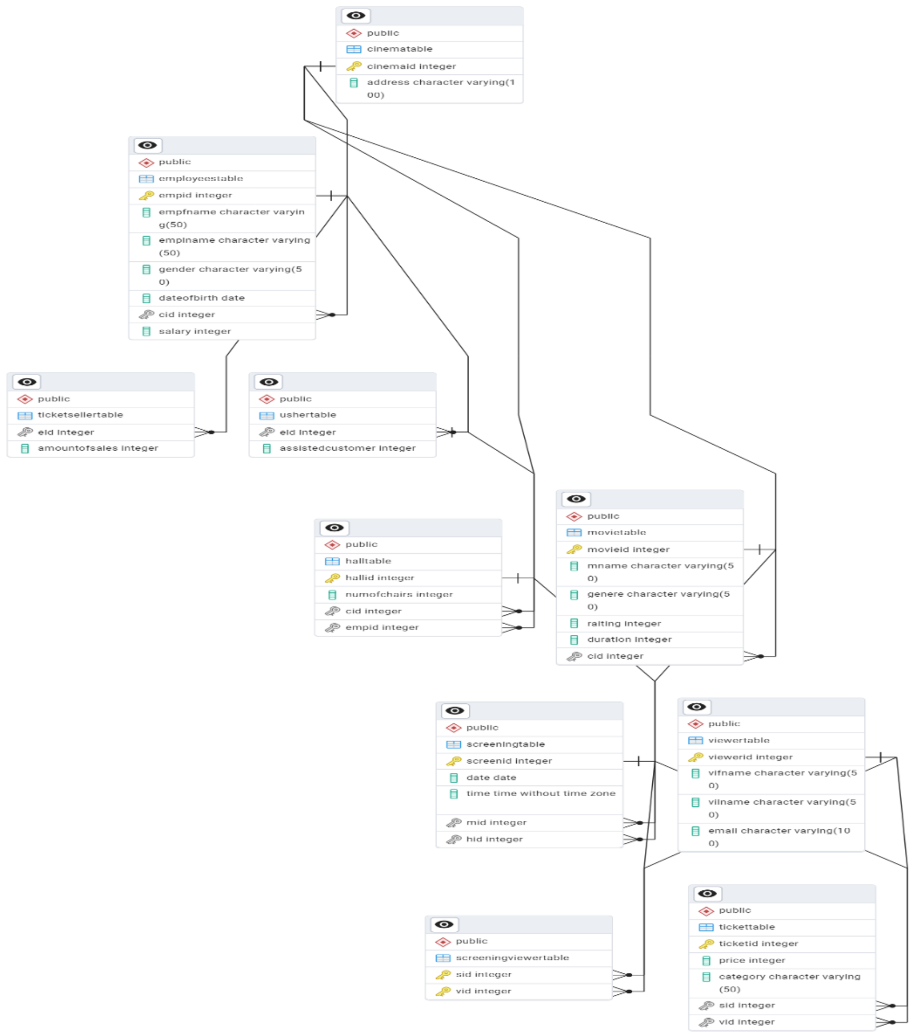
Who are the user of the system? what is the database for?

The primary users of our system are the cinema manager, employees, and viewers.

The data structure is designed to ensure that all information is presented efficiently and organized, allowing the manager to manage the cinema effectively.

Additionally, employees and viewers will benefit from various viewing data we have incorporated into the code, enhancing their experience and access to relevant information about movies and screenings.


Shani&Ori Cinema
Cin_ID
Address
Made with Visual Paradigm For non-commercial use
Cinema-hall
1
M
Cinema hall
Hall_ID
num of rows - Deleted from our code
num of chairs
1
1
Cinema-Employees
M
1
Cinema-movie
M
1
Hall-Screening
M
Screening
Screen_ID
Date
Time
Movie_ID (FK)
Hall_ID(FK)
Empoyees
Emp_ID
empFname
empLname
gender
Date of birth
Salary
Movie
Movie_ID
Name
Genre
Raiting
Duration
Movie-Screening
M
1
M
Viewer-Screening
1
Screening-Ticket
N
M
ISA
Usher
Assisted customer
Ticket seller
Amount of sales
Viewer
Viewer_ID
viFname
viLname
Email
1
Viewer-Ticket
M
Ticket
Ticket_ID
Price
Category(Regular, VIP, Special)
1
Hall-Usher


public
cinematable
cinemaid integer
address character varying(100)
public
employeestable
empid integer
empfname character varying(50)
emplname character varying(50)
gender character varying(50)
dateofbirth date
cid integer
salary integer
public
ticketsellertable
eid integer
amountofsales integer
public
ushertable
eid integer
assistedcustomer integer
public
movietable
movieid integer
mname character varying(50)
genere character varying(50)
raiting integer
duration integer
cid integer
public
halltable
hallid integer
numofchairs integer
cid integer
empid integer
public
screeningtable
screenid integer
date date
time time without time zone
mid integer
hid integer
public
viewertable
viewerid integer
vifname character varying(50)
vilname character varying(50)
email character varying(100)
public
tickettable
ticketid integer
price integer
category character varying(50)
sid integer
vid integer
public
screeningviewertable
sid integer
vid integer

Cinema Tables

Shani&Ori Cinema Table

Cinema_ID (PK)	Address

Employees Table

Emp_ID (PK)	Cinema_ID (FK)	empFname	empLname	Gender	Date of birth	Salary

Usher Table

Emp_ID (FK)	Assisted customer

Ticket seller Table

Emp_ID (FK)	Amount of sales

Hall Table

Hall_ID (PK)	Cinema_ID (FK)	Emp_ID (FK)	Num of rows (Deleted later in the code)	Num of chairs

Movie Table

Movie_ID (PK)	Cinema_ID (FK)	Name	Genere	Raiting	Duration

Screening Table

Screen_ID (PK)	Movie_ID (FK)	Hall_ID (FK)	Date	Time

Viewer Table

Viewer_ID (PK)	viFname	viLname	Email

Screening_Viewer Table

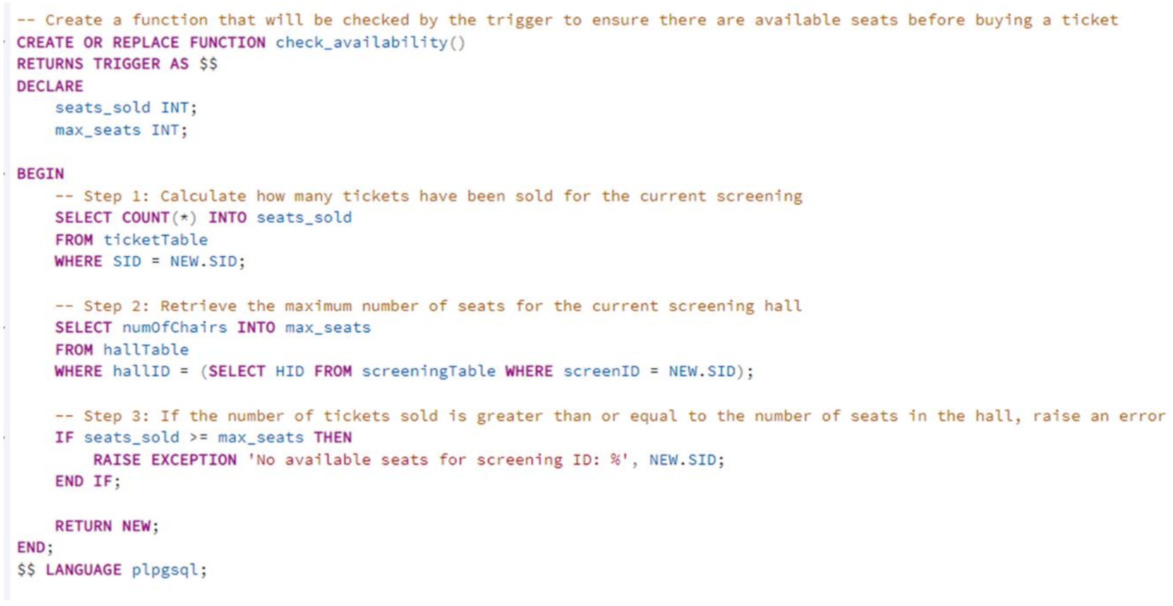
Screen_ID (PK)	Viewer_ID (PK)

Ticket Table

Ticket_ID (PK)	Screen_ID (FK)	Viewer_ID (FK)	Price	Category

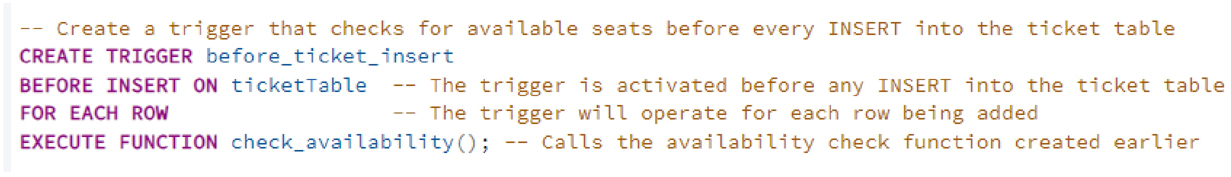
Working with the database:

1. We created the function "check_availability" to ensure that before purchasing a new ticket, there is availability and an open seat for the specific screening the viewer wishes to attend. This is verified based on the number of seats in the hall where the screening occurs.



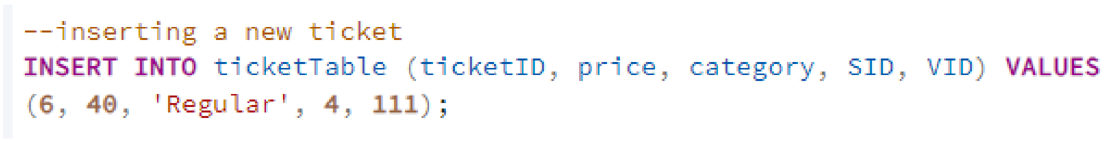
```sql
-- Create a function that will be checked by the trigger to ensure there are available seats before buying a ticket
CREATE OR REPLACE FUNCTION check_availability()
RETURNS TRIGGER AS $$
DECLARE
    seats_sold INT;
    max_seats INT;

BEGIN
    -- Step 1: Calculate how many tickets have been sold for the current screening
    SELECT COUNT(*) INTO seats_sold
    FROM ticketTable
    WHERE SID = NEW.SID;

    -- Step 2: Retrieve the maximum number of seats for the current screening hall
    SELECT numOfChairs INTO max_seats
    FROM hallTable
    WHERE hallID = (SELECT HID FROM screeningTable WHERE screenID = NEW.SID);

    -- Step 3: If the number of tickets sold is greater than or equal to the number of seats in the hall, raise an error
    IF seats_sold >= max_seats THEN
        RAISE EXCEPTION 'No available seats for screening ID: %', NEW.SID;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

2. We established a trigger that activates to check seat availability based on the number of chairs in the screening hall.
This trigger is triggered before each new ticket purchase and checks seat availability using the "check_availability" function we created.



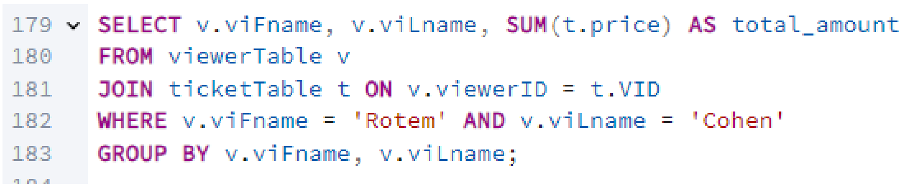
```sql
-- Create a trigger that checks for available seats before every INSERT into the ticket table
CREATE TRIGGER before_ticket_insert
BEFORE INSERT ON ticketTable  -- The trigger is activated before any INSERT into the ticket table
FOR EACH ROW                  -- The trigger will operate for each row being added
EXECUTE FUNCTION check_availability(); -- Calls the availability check function created earlier
```

3. We added an INSERT statement aimed at purchasing a new ticket for a screening.



```sql
--inserting a new ticket
INSERT INTO ticketTable (ticketID, price, category, SID, VID) VALUES
(6, 40, 'Regular', 4, 111);
```

4. The query checks the total expenditure on ticket purchases by the viewer "Rotem Cohen" in the cinema using GROUP BY and the SUM



```sql
SELECT v.viFname, v.viLname, SUM(t.price) AS total_amount
FROM viewerTable v
JOIN ticketTable t ON v.viewerID = t.VID
WHERE v.viFname = 'Rotem' AND v.viLname = 'Cohen'
GROUP BY v.viFname, v.viLname;
```

5. We executed a query that deletes the column for the number of rows in the hall, as we decided to remove it because the number of seats includes the total chairs in the hall. This change reflects a revision of our initial design since we no longer needed this data.

```
ALTER TABLE hallTable
DROP COLUMN numOfRows;
```

6. We wrote an UPDATE statement to modify the screening dates for screenings with the identifiers 2 and 3 from the date 2024-09-25 to 2024-09-27, effectively postponing these screenings by two days.

```
UPDATE screeningTable
SET date = '2024-09-27'
WHERE screenID = 2 OR screenID = 3;
```

7. We created a query that sorts all cinema screenings by date and time. The resulting table displays the movie title for each screening, the screening date, time, and the duration of the movie, aiming to present the screenings in an accessible and organized manner.

```
SELECT
    MT.Mname AS movie_name,
    ST.date AS screening_date,
    ST.time AS screening_time,
    MT.duration AS movie_duration
FROM
    screeningTable ST
JOIN
    movieTable MT ON MT.movieID = ST.MID
ORDER BY
    ST.date,
    ST.time;
```

8. We executed a query to add a new movie to the cinema – the movie SUPERMAN.

```
--inserting a new Movie
INSERT INTO movieTable (movieID, Mname, Genere, Raiting, Duration, CID) VALUES
(50, 'Superman', 'Action',4, 143,1);
```

9. We implemented a query that deletes movies with a rating of 5 or below, as the cinema does not wish to retain less popular films. However, if a movie has already scheduled screenings, it will not be deleted to avoid impacting viewers who have purchased tickets.

```
DELETE FROM movieTable
WHERE raiting <= 5 AND NOT EXISTS (
    SELECT 1
    FROM screeningTable
    WHERE MID = movieTable.movieID
);
```

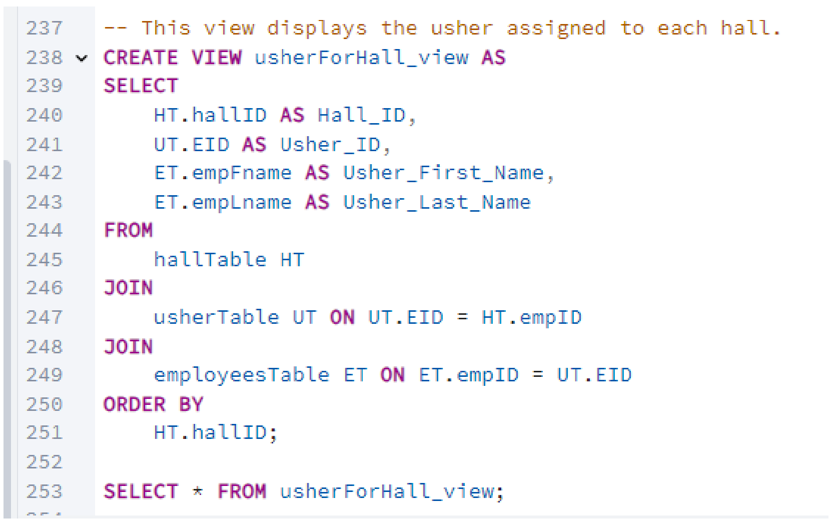
10. We formulated a query that finds the movie with the maximum number of screenings using GROUP BY and ORDER BY.

```sql
--Finding the Movie name with the max number of Screening
SELECT MT.Mname, COUNT(ST.screenID) AS screening_count
FROM movieTable MT
JOIN screeningTable ST ON MT.movieID = ST.MID
GROUP BY MT.Mname
ORDER BY screening_count DESC
LIMIT 1;
```

11. We added a column to the employees table containing the salary per hour for each employee.
We decided to include this field after the initial design of the cinema.

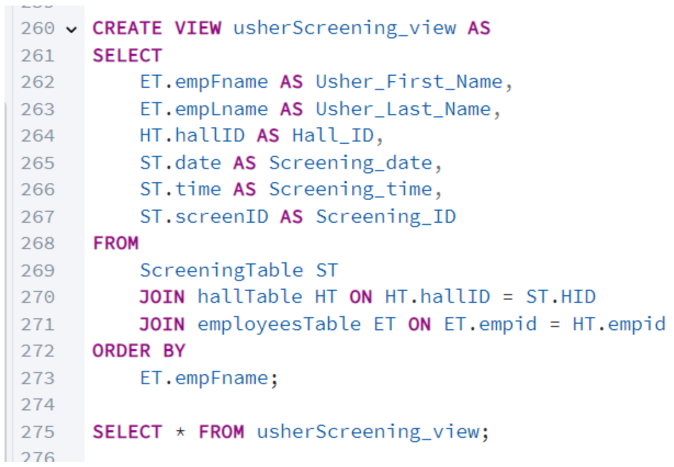
```sql
ALTER TABLE employeesTable
ADD COLUMN salary INT DEFAULT 30;
```

12. We created a VIEW that presents the list of halls along with the name of the usher assigned to each hall.



```sql
-- This view displays the usher assigned to each hall.
CREATE VIEW usherForHall_view AS
SELECT
    HT.hallID AS Hall_ID,
    UT.EID AS Usher_ID,
    ET.empFname AS Usher_First_Name,
    ET.empLname AS Usher_Last_Name
FROM
    hallTable HT
JOIN
    usherTable UT ON UT.EID = HT.empID
JOIN
    employeesTable ET ON ET.empID = UT.EID
ORDER BY
    HT.hallID;

SELECT * FROM usherForHall_view;
```

13. The usherScreening_view displays ushers and their assigned screening shifts, showing each usher's first and last name, hall ID, screening date, time, and screening ID.
This view organizes the data by the ushers' first names for easier reference.



```sql
CREATE VIEW usherScreening_view AS
SELECT
    ET.empFname AS Usher_First_Name,
    ET.empLname AS Usher_Last_Name,
    HT.hallID AS Hall_ID,
    ST.date AS Screening_date,
    ST.time AS Screening_time,
    ST.screenID AS Screening_ID
FROM
    ScreeningTable ST
    JOIN hallTable HT ON HT.hallID = ST.HID
    JOIN employeesTable ET ON ET.empid = HT.empid
ORDER BY
    ET.empFname;

SELECT * FROM usherScreening_view;
```

14. We added two new employees via INSERT, designating one as an usher and the other as a counter worker.

```
--inserting a new employees and give them job
INSERT INTO employeesTable VALUES
(4,'Noa','Kirel','Female','2001-04-10',1),
(5,'Omer','Adam','Male','1993-10-22',1);

INSERT INTO ticketSellerTable VALUES
(4,20);

INSERT INTO usherTable VALUES
(5,1);
```

15. We implemented a query that awards a bonus to outstanding employees, giving a bonus of 10 shekels to the hourly salary of each employee. An outstanding employee is defined as a counter worker who sold more than 30 tickets or an usher who assisted at least ten people.

```
--Bonus for the best workers
UPDATE employeesTable
SET salary = salary + 10
WHERE empID IN (
    SELECT EID
    FROM ticketSellerTable
    WHERE amountOfSales > 30

    UNION

    SELECT EID
    FROM usherTable
    WHERE assistedCustomer >= 10
);
```

16. We added "employees_view," which displays the employee table sorted alphabetically by first names, including their roles but excluding their salaries so that this information is not accessible to everyone.

```
CREATE VIEW employees_view AS
SELECT ET.empID, ET.empFname, ET.empLname, ET.Gender, ET.DateOfBirth,
       CASE
           WHEN TS.EID IS NOT NULL THEN 'Ticket Seller'
           WHEN UT.EID IS NOT NULL THEN 'Usher'
           ELSE 'Unknown'
       END AS jobTitle
FROM employeesTable ET
LEFT JOIN ticketSellerTable TS ON ET.empID = TS.EID
LEFT JOIN usherTable UT ON ET.empID = UT.EID
ORDER BY ET.empFname;

SELECT * FROM employees_view;
```

17. We added a query that calculates the cinema's revenue from ticket sales, summing up the total sales.

```
SELECT SUM(price) AS total_income
FROM ticketTable;
```

18. We introduced a query that includes an aggregate and nested function to calculate the average number of viewers and revenue for each movie. It will only display movies where the average number of viewers per screening exceeds 20, enabling the manager to monitor which films successfully attract a larger audience and the average revenue generated by each.

```sql
-- This query calculates the average number of viewers per movie and the average income per movie,
-- based on the number of viewers per screening as recorded in the screeningViewer table.
SELECT MT.mname,
       ROUND(AVG(viewers_count), 3) AS avg_viewers,
       ROUND(AVG(total_income), 3) AS avg_income
FROM movieTable MT
JOIN screeningTable ST ON MT.movieID = ST.MID
JOIN (
    -- calculate total number of viewers per screening
    SELECT SVT.SID, COUNT(SVT.VID) AS viewers_count
    FROM screeningViewerTable SVT
    GROUP BY SVT.SID
) VC ON ST.screenID = VC.SID
JOIN (
    -- calculate total income per screening
    SELECT TT.SID, SUM(TT.price) AS total_income
    FROM ticketTable TT
    GROUP BY TT.SID
) TI ON ST.screenID = TI.SID
GROUP BY MT.mname
HAVING AVG(viewers_count) >= 20;
```

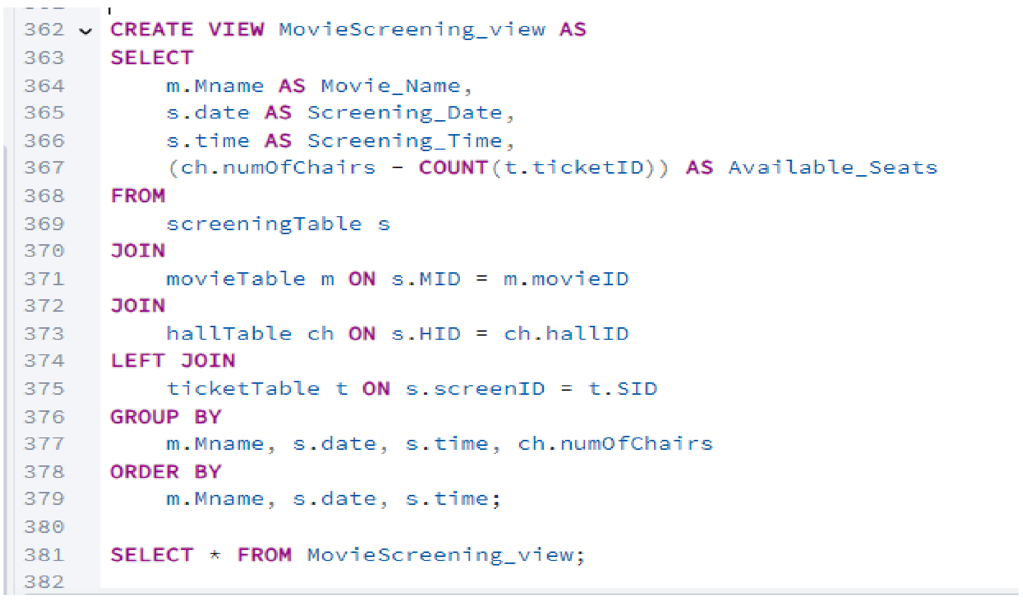
19. We created a function named "check_movie_screenings" designed to verify if a specific movie has already had two screenings on the same date; if so, it raises an exception.

```sql
-- This function checks if a movie is already scheduled 2 times on the same date
CREATE OR REPLACE FUNCTION check_movie_screenings()
RETURNS TRIGGER AS $$
DECLARE
    screenings_count INT;
BEGIN
    SELECT COUNT(*) INTO screenings_count
    FROM screeningTable
    WHERE MID = NEW.MID
    AND date = NEW.date;

    IF screenings_count = 2 THEN
        RAISE EXCEPTION 'The movie has already been scheduled 2 times on this date: %', NEW.date;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

20. We defined a trigger named "before_screening_insert" that activates this function before any new screening is added to the screeningTable.

```sql
CREATE TRIGGER before_screening_insert
BEFORE INSERT ON screeningTable
FOR EACH ROW
EXECUTE FUNCTION check_movie_screenings();
```

21. We established a VIEW that allows for a convenient display of the screened movies, organized by movie title and including the screening date, time, and the number of remaining seats for each screening (calculated as the number of chairs in the hall minus the number of tickets already sold).



```sql
CREATE VIEW MovieScreening_view AS
SELECT
    m.Mname AS Movie_Name,
    s.date AS Screening_Date,
    s.time AS Screening_Time,
    (ch.numOfChairs - COUNT(t.ticketID)) AS Available_Seats
FROM
    screeningTable s
JOIN
    movieTable m ON s.MID = m.movieID
JOIN
    hallTable ch ON s.HID = ch.hallID
LEFT JOIN
    ticketTable t ON s.screenID = t.SID
GROUP BY
    m.Mname, s.date, s.time, ch.numOfChairs
ORDER BY
    m.Mname, s.date, s.time;

SELECT * FROM MovieScreening_view;
```