DBMS ASSIGNMENT – 4

UE19CS301

PROJECT TITLE: ONLINE MOVIE TICKET BOOKING MANAGEMENT SYSTEM

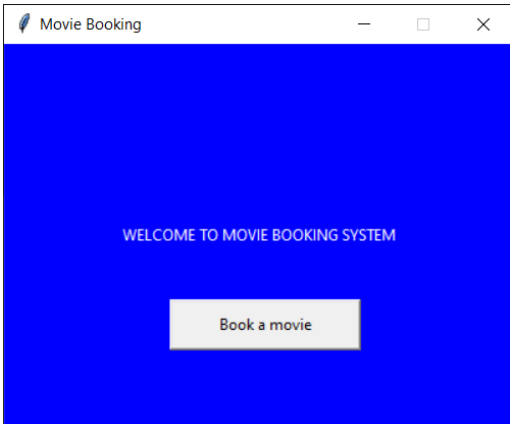
SEMESTER: 5

SECTION: B

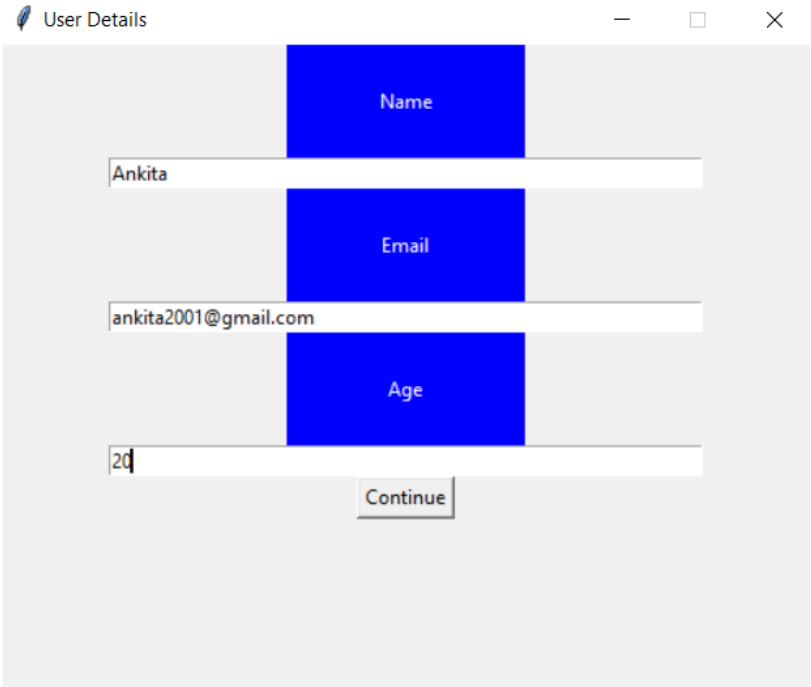
SRN	Team Members
PES1UG19CS080	Apoorva BS
PES1UG19CS068	Ankita V
PES1UG19CS079	Anvika D Shriyan

SCREENSHOTS OF FRONTEND:

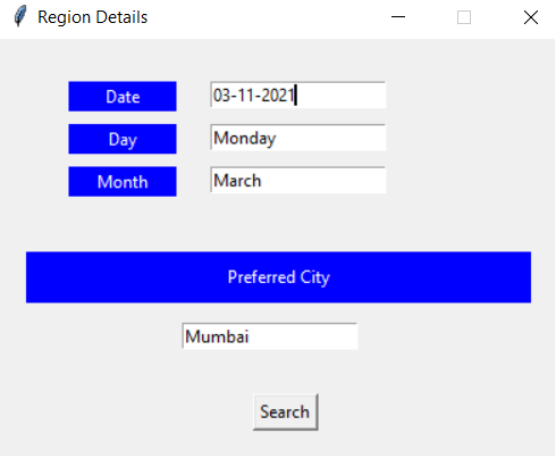
Welcome Screen:



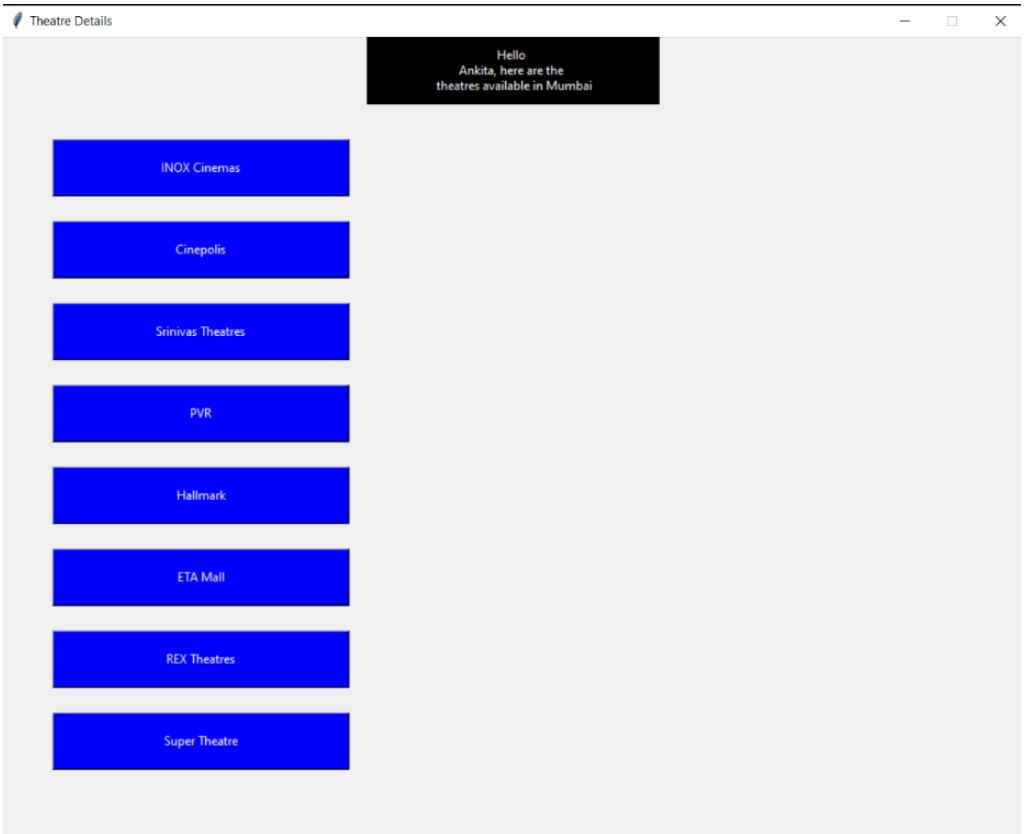
The site asks for the user details:



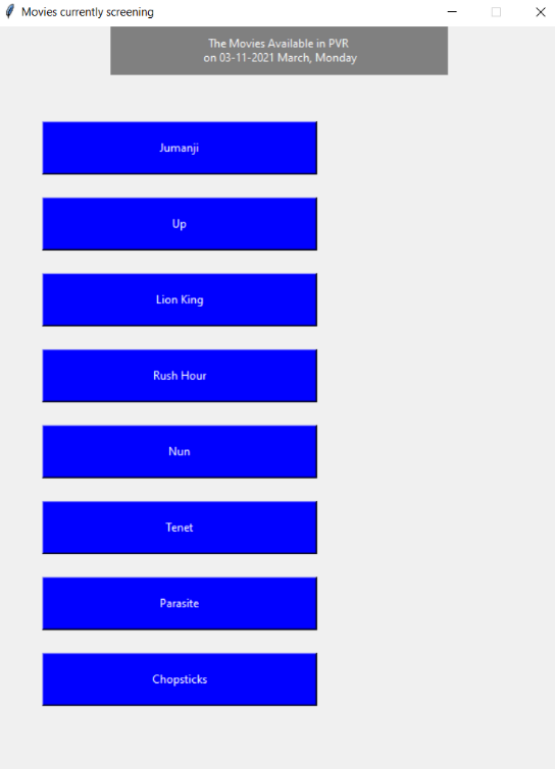
The details of the region the user currently resides in:



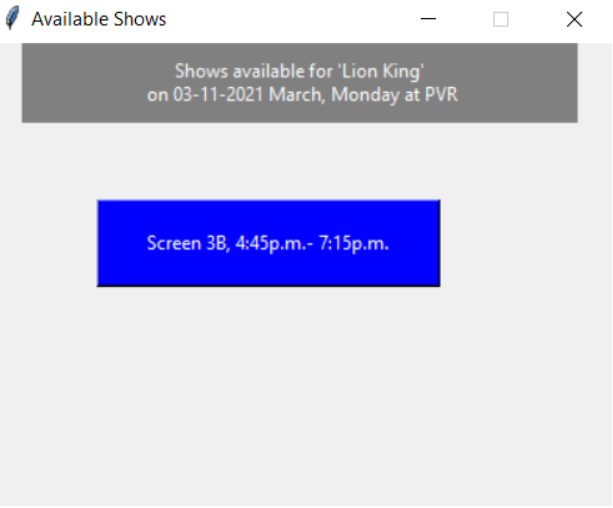
The user gets asked to choose the Theatre to book a movie in:



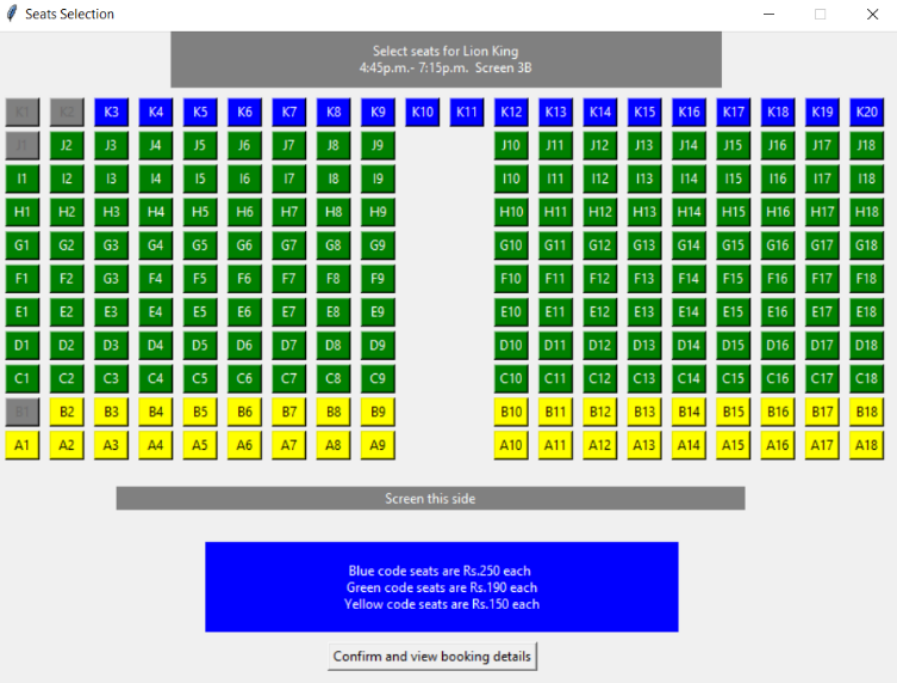
The currently screening movies are displayed:



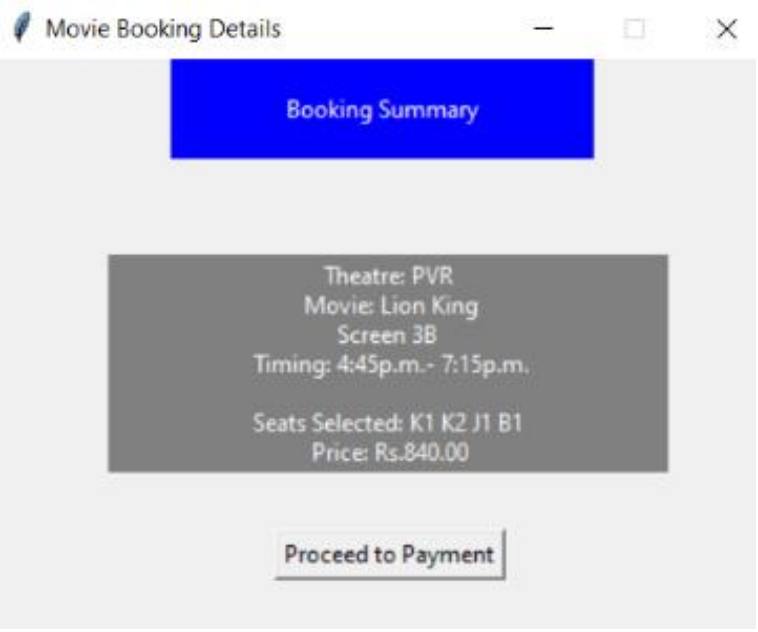
The shows available for the chosen movie are displayed:



The seats available in the movie booking system are shown:



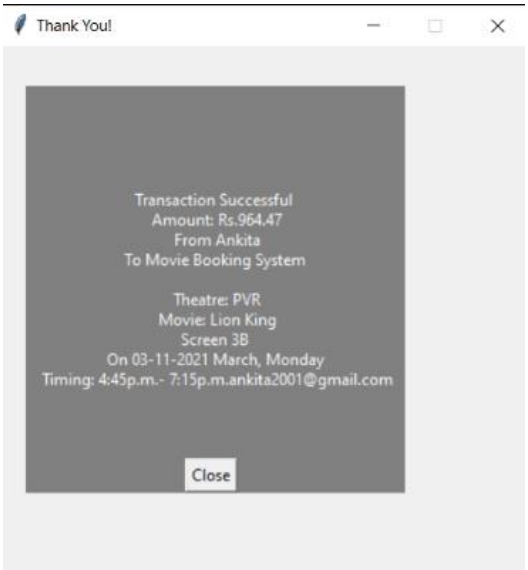
The booking summary is shown:



The Payment summary along with the method to chosen is displayed:

There are two payment methods – Credit/Debit Card or Net banking

The final booking summary along with the total amount paid is displayed:



DEPENDENCIES INSTALLED FOR DATABASE CONNECTIVITY:

To connect the frontend with the backend, these were the following steps done:

- Installing the psycopg2 package
  `pip3 install psycopg2`
- Connection is done using `connect()` function
  `Connection = psycopg2.connect("dbname=cs068_079_080 user=postgres password=postgres")`
- Use `config()` with `connect()` to obtain the Postgresql version of database
  `from config import config`
- For succesful connection, execute `connect.py` file
  `python3 connect.py`

VIEWS:

There are 2 views for the database.

```
cs068_079_080=# create view viewname as select;
CREATE VIEW
cs068_079_080=# create view user1 as select * from users where age>30;
CREATE VIEW
cs068_079_080=# create view movie_genre as select * from movies where genre = 'HORROR';
CREATE VIEW
cs068_079_080=# select * from user1;
 user_id  |   name   | age |        email_id
----------+----------+-----+-------------------------
 1FGH6539 | ANVIKA   |  31 | anvikas@gmail.com
 8GRF4377 | RAHUL    |  54 | rahulmehta@gmail.com
 3RFP5564 | BHARATH  |  69 | bharathpan@gmail.com
 7DXK8945 | KALYANAM |  81 | kalyanamseshu@gmail.com
(4 rows)

cs068_079_080=# select * from movie_genre;
 movie_id | movie_name | release_date | genre
----------+------------+--------------+--------
 NUN444   | NUN        | 2021-09-14   | HORROR
(1 row)

cs068_079_080=#
```

PERFORMANCE ANALYSIS:

Simple query using EXPLAIN:

```
postgres=# \c cs068_079_080
You are now connected to database "cs068_079_080" as user "postgres".
cs068_079_080=#
cs068_079_080=#
cs068_079_080=# EXPLAIN select movie_name from movies where genre = 'ANIMATED';
                       QUERY PLAN
---------------------------------------------------------
 Seq Scan on movies  (cost=0.00..15.38 rows=2 width=68)
   Filter: ((genre)::text = 'ANIMATED'::text)
(2 rows)

cs068_079_080=#
```

Simple query using EXPLAIN ANALYZE:

```
cs068_079_080=# EXPLAIN(ANALYZE) select theatre_name, no_of_screens from theatre where theatre_name = 'PVR';
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on theatre  (cost=0.00..16.12 rows=2 width=62) (actual time=0.039..0.043 rows=1 loops=1)
   Filter: ((theatre_name)::text = 'PVR'::text)
   Rows Removed by Filter: 7
 Planning Time: 0.394 ms
 Execution Time: 0.078 ms
(5 rows)

cs068_079_080=#
```

Complex query using EXPLAIN ANALYZE:

```
cs068_079_080=# EXPLAIN(ANALYZE) select B.booking_id, B.no_of_tickets, M.payment from booking as B, make_booking as M where B.booking_id = M.bid and payment = 'DEBITCARD'
ORDER BY no_of_tickets;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=32.62..32.63 rows=2 width=100) (actual time=0.142..0.145 rows=2 loops=1)
   Sort Key: b.no_of_tickets
   Sort Method: quicksort  Memory: 25kB
   ->  Nested Loop  (cost=0.15..32.61 rows=2 width=100) (actual time=0.057..0.069 rows=2 loops=1)
         ->  Seq Scan on make_booking m  (cost=0.00..16.25 rows=2 width=96) (actual time=0.021..0.024 rows=2 loops=1)
               Filter: ((payment)::text = 'DEBITCARD'::text)
               Rows Removed by Filter: 6
         ->  Index Scan using booking_pkey on booking b  (cost=0.15..8.17 rows=1 width=42) (actual time=0.016..0.016 rows=1 loops=2)
               Index Cond: ((booking_id)::text = (m.bid)::text)
 Planning Time: 0.891 ms
 Execution Time: 0.229 ms
(11 rows)
```

Nested query using EXPLAIN ANALYZE:

```
cs068_079_080=# EXPLAIN(ANALYZE) select city, pincode from region where no_of_theatres = (select MAX(no_of_theatres) from region);
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on region  (cost=18.26..36.51 rows=3 width=52) (actual time=0.042..0.045 rows=1 loops=1)
   Filter: (no_of_theatres = $0)
   Rows Removed by Filter: 7
   InitPlan 1 (returns $0)
     ->  Aggregate  (cost=18.25..18.26 rows=1 width=4) (actual time=0.017..0.019 rows=1 loops=1)
           ->  Seq Scan on region region_1  (cost=0.00..16.60 rows=660 width=4) (actual time=0.004..0.007 rows=8 loops=1)
 Planning Time: 0.532 ms
 Execution Time: 0.113 ms
(8 rows)
```

EXISTS query using EXPLAIN ANALYZE:

```
cs068_079_080=# EXPLAIN ANALYZE select user_id, name
cs068_079_080-# from users
cs068_079_080-# where EXISTS (select no_of_tickets, cost from booking where users.user_id = booking.user_id and no_of_tickets>3 and cost<1200)
cs068_079_080-# ORDER BY user_id;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Sort  (cost=37.09..37.24 rows=60 width=86) (actual time=0.170..0.173 rows=1 loops=1)
   Sort Key: users.user_id
   Sort Method: quicksort  Memory: 25kB
   ->  Hash Semi Join  (cost=18.85..35.32 rows=60 width=86) (actual time=0.100..0.109 rows=1 loops=1)
         Hash Cond: ((users.user_id)::text = (booking.user_id)::text)
         ->  Seq Scan on users  (cost=0.00..14.30 rows=430 width=86) (actual time=0.009..0.012 rows=8 loops=1)
         ->  Hash  (cost=18.10..18.10 rows=60 width=38) (actual time=0.044..0.044 rows=1 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 9kB
               ->  Seq Scan on booking  (cost=0.00..18.10 rows=60 width=38) (actual time=0.021..0.025 rows=1 loops=1)
                     Filter: ((no_of_tickets > 3) AND (cost < 1200))
                     Rows Removed by Filter: 7
 Planning Time: 1.315 ms
 Execution Time: 0.328 ms
(13 rows)
```

NOT EXISTS query using EXPLAIN ANALYZE:

```
cs068_079_080=# EXPLAIN ANALYZE select user_id, name
from users
where NOT EXISTS (select no_of_tickets, cost from booking where users.user_id = booking.user_id and no_of_tickets>3 and cost<1200)
ORDER BY user_id;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Sort  (cost=54.20..55.13 rows=370 width=86) (actual time=0.091..0.095 rows=7 loops=1)
   Sort Key: users.user_id
   Sort Method: quicksort  Memory: 25kB
   ->  Hash Anti Join  (cost=18.85..38.42 rows=370 width=86) (actual time=0.062..0.070 rows=7 loops=1)
         Hash Cond: ((users.user_id)::text = (booking.user_id)::text)
         ->  Seq Scan on users  (cost=0.00..14.30 rows=430 width=86) (actual time=0.020..0.023 rows=8 loops=1)
         ->  Hash  (cost=18.10..18.10 rows=60 width=38) (actual time=0.021..0.022 rows=1 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 9kB
               ->  Seq Scan on booking  (cost=0.00..18.10 rows=60 width=38) (actual time=0.012..0.016 rows=1 loops=1)
                     Filter: ((no_of_tickets > 3) AND (cost < 1200))
                     Rows Removed by Filter: 7
 Planning Time: 0.254 ms
 Execution Time: 0.161 ms
(13 rows)
```

IN query using EXPLAIN ANALYZE:

```
cs068_079_080=# EXPLAIN(ANALYZE) select name from users where user_id IN (select user_id from booking where no_of_tickets > 3);
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=20.19..37.62 rows=180 width=48) (actual time=0.131..0.141 rows=2 loops=1)
   Hash Cond: ((users.user_id)::text = (booking.user_id)::text)
   ->  Seq Scan on users  (cost=0.00..14.30 rows=430 width=86) (actual time=0.028..0.031 rows=8 loops=1)
   ->  Hash  (cost=18.53..18.53 rows=133 width=38) (actual time=0.045..0.046 rows=2 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  HashAggregate  (cost=17.20..18.53 rows=133 width=38) (actual time=0.027..0.031 rows=2 loops=1)
               Group Key: (booking.user_id)::text
               ->  Seq Scan on booking  (cost=0.00..16.75 rows=180 width=38) (actual time=0.013..0.016 rows=2 loops=1)
                     Filter: (no_of_tickets > 3)
                     Rows Removed by Filter: 6
 Planning Time: 0.469 ms
 Execution Time: 0.296 ms
(12 rows)
```

TRIGGER FUNCTION:

Triggers have been used to avoid booking of the same seats again by multiple users.

```
cs068_079_080=# CREATE TABLE MOVIE (MOVIE_ID INT NOT NULL, NO_OF_SEATS INT, PRIMARY KEY(MOVIE_ID));
CREATE TABLE
cs068_079_080=# CREATE TABLE SEAT (SEAT_ID INT NOT NULL, MOVIE_ID INT NOT NULL, FOREIGN KEY(MOVIE_ID) REFERENCES MOVIE(MOVIE_ID), PRIMARY KEY(SEAT_ID));
CREATE TABLE
cs068_079_080=#
cs068_079_080=#
cs068_079_080=# CREATE FUNCTION INCREASE_SEATS()
 RETURNS TRIGGER AS $$
 BEGIN
  UPDATE MOVIE
  SET NO_OF_SEATS = NO_OF_SEATS + 1
  WHERE MOVIE_ID = NEW.MOVIE_ID;
  RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';
CREATE FUNCTION
cs068_079_080=# CREATE FUNCTION DECREASE_SEATS()
 RETURNS TRIGGER AS $$
 BEGIN
  UPDATE MOVIE
  SET NO_OF_SEATS = NO_OF_SEATS - 1
  WHERE MOVIE_ID = OLD.MOVIE_ID;
  RETURN OLD;
END;
$$
LANGUAGE 'plpgsql';
CREATE FUNCTION
cs068_079_080=#
cs068_079_080=# ^M
CREATE TRIGGER INCREASE
BEFORE DELETE ON SEAT
FOR EACH ROW
EXECUTE PROCEDURE INCREASE_SEATS();
CREATE TRIGGER
cs068_079_080=#
cs068_079_080=# ^M
CREATE TRIGGER DECREASE
AFTER INSERT ON SEAT
FOR EACH ROW
EXECUTE PROCEDURE DECREASE_SEATS();
CREATE TRIGGER
```

The trigger functions are INCREASE_SEATS() and DECREASE_SEATS().
The triggers are INCREASE and DECREASE.
INCREASE trigger is called BEFORE DELETE of records from the MOVIE table.
DECREASE trigger is called AFTER INSERT of records in the MOVIE table.

REVOKING ACCESS FROM USERS:

5 users with access to different parts of the database have been created (presented in Assignment 3). The below code was used to revoke access on some parts of the database from all 5 users.

```
cs068_079_080=# revoke select on Movies from USER1;
REVOKE
cs068_079_080=# revoke insert on Users from USER2;
REVOKE
cs068_079_080=# revoke all on Booking from USER3;
REVOKE
cs068_079_080=# revoke delete on Languages from USER4;
REVOKE
cs068_079_080=# revoke all on Screen from USER5;
REVOKE
```

IMPLEMENTING CONCURRENCY CONTROL:

A transaction was started in one terminal and database access was tried from another terminal. Accessing database from another terminal during transaction resulted in an error.

Transaction in terminal 1:

```
postgres=# \c cs068_079_080
You are now connected to database "cs068_079_080" as user "postgres".
cs068_079_080=# create table food_counters (price int, item varchar(15));
CREATE TABLE
cs068_079_080=# begin;
BEGIN
cs068_079_080=#
```

Error in terminal 2:

```
dell@dell-Inspiron-7559:~/Desktop/DBMS_Assignment$ sudo -i -u postgres
[sudo] password for dell:
postgres@dell-Inspiron-7559:~$ psql -U postgres -f /home/dell/Desktop/DBMS_Assignment/cs068_079_080.sql
psql:/home/dell/Desktop/DBMS_Assignment/cs068_079_080.sql:1: ERROR:  database "cs068_079_080" is being accessed by other users
DETAIL:  There is 1 other session using the database.
psql:/home/dell/Desktop/DBMS_Assignment/cs068_079_080.sql:2: ERROR:  database "cs068_079_080" already exists
You are now connected to database "cs068_079_080" as user "postgres".
psql:/home/dell/Desktop/DBMS_Assignment/cs068_079_080.sql:6: ERROR:  relation "users" already exists
psql:/home/dell/Desktop/DBMS_Assignment/cs068_079_080.sql:8: ERROR:  relation "screen" already exists
psql:/home/dell/Desktop/DBMS_Assignment/cs068_079_080.sql:10: ERROR:  relation "movies" already exists
psql:/home/dell/Desktop/DBMS_Assignment/cs068_079_080.sql:13: ERROR:  relation "show" already exists
psql:/home/dell/Desktop/DBMS_Assignment/cs068_079_080.sql:15: ERROR:  relation "booking" already exists
psql:/home/dell/Desktop/DBMS_Assignment/cs068_079_080.sql:17: ERROR:  relation "ticket" already exists
psql:/home/dell/Desktop/DBMS_Assignment/cs068_079_080.sql:19: ERROR:  relation "languages" already exists
psql:/home/dell/Desktop/DBMS_Assignment/cs068_079_080.sql:21: ERROR:  relation "theatre" already exists
psql:/home/dell/Desktop/DBMS_Assignment/cs068_079_080.sql:23: ERROR:  relation "region" already exists
psql:/home/dell/Desktop/DBMS_Assignment/cs068_079_080.sql:25: ERROR:  relation "make_booking" already exists
psql:/home/dell/Desktop/DBMS_Assignment/cs068_079_080.sql:31: ERROR:  constraint "usid" for relation "make_booking" already exists
psql:/home/dell/Desktop/DBMS_Assignment/cs068_079_080.sql:32: ERROR:  constraint "bid" for relation "make_booking" already exists
```

SCHEMA AND CONSTRAINTS CHANGE:

The database schema was changed by adding another table to the existing 10 tables with 8 values. The newly added table has 8 values and 2 foreign key references to two different tables in the previous schema. There are 11 tables in total in the new schema.

```
DROP DATABASE cs068_079_080;
CREATE DATABASE cs068_079_080;

\c cs068_079_080

CREATE TABLE USERS (USER_ID VARCHAR (10) NOT NULL, NAME VARCHAR (15), AGE INT, EMAIL_ID VARCHAR (25),UNIQUE (USER_ID), PRIMARY KEY(USER_ID, EMAIL_ID));

CREATE TABLE SCREEN (SCREEN_ID VARCHAR (10) NOT NULL, SEATS_IN_EACH_CLASS VARCHAR (15), PRIMARY KEY(SCREEN_ID));

CREATE TABLE MOVIES (MOVIE_ID VARCHAR (10) NOT NULL, MOVIE_NAME VARCHAR (25), RELEASE_DATE DATE, GENRE VARCHAR (15), UNIQUE (MOVIE_ID), PRIMARY KEY(MOVIE_ID));

CREATE TABLE SHOW (SHOW_ID VARCHAR (10)NOT NULL, TIME INT NOT NULL, SHOW_DATE DATE, UNIQUE (SHOW_ID), SCREEN_ID VARCHAR (10), MOVIE_ID VARCHAR (10), FOREIGN KEY (SCREEN_ID) REFERENCES SCREEN(SCREEN_ID)
FOREIGN KEY (MOVIE_ID) REFERENCES MOVIES(MOVIE_ID), PRIMARY KEY(SHOW_ID));

CREATE TABLE BOOKING (BOOKING_ID VARCHAR (10) NOT NULL, NO_OF_TICKETS INT NOT NULL, COST INT NOT NULL, UNIQUE (BOOKING_ID), PRIMARY KEY(BOOKING_ID), USER_ID VARCHAR (10), SHOW_ID VARCHAR (10), FOREIGN
KEY (USER_ID) REFERENCES USERS(USER_ID),FOREIGN KEY (SHOW_ID) REFERENCES SHOW(SHOW_ID));

CREATE TABLE TICKET (TICKET_ID VARCHAR (10) NOT NULL, PRICE INT NOT NULL, CLASS VARCHAR (10), BOOKING_ID VARCHAR (10), FOREIGN KEY (BOOKING_ID) REFERENCES BOOKING(BOOKING_ID), PRIMARY KEY(TICKET_ID));

CREATE TABLE LANGUAGES (LANGUAGE_ID VARCHAR (10) NOT NULL, LANGUAGE_NAME VARCHAR (10) DEFAULT 'ENGLISH' NOT NULL, MOVIE_ID VARCHAR (10), FOREIGN KEY (MOVIE_ID) REFERENCES MOVIES(MOVIE_ID), PRIMARY
KEY(LANGUAGE_ID));

CREATE TABLE THEATRE (THEATRE_ID VARCHAR (10) NOT NULL, THEATRE_NAME VARCHAR (20) NOT NULL, NO_OF_SCREENS INT, MOVIE_ID VARCHAR (10), FOREIGN KEY (MOVIE_ID) REFERENCES MOVIES(MOVIE_ID), UNIQUE
(THEATRE_ID), PRIMARY KEY(THEATRE_ID));

CREATE TABLE REGION (CITY VARCHAR (15), PINCODE INT NOT NULL, NO_OF_THEATRES INT, THEATRE_ID VARCHAR (10), FOREIGN KEY (THEATRE_ID) REFERENCES THEATRE(THEATRE_ID), PRIMARY KEY(PINCODE));

CREATE TABLE MAKE_BOOKING (PAYMENT VARCHAR (20) NOT NULL, USID VARCHAR (10), BID VARCHAR (10));

CREATE TABLE FOOD_COUNTER (FOOD_ID VARCHAR (10) NOT NULL, MOVIE_ID VARCHAR (10), BOOKING_ID VARCHAR (10), FOREIGN KEY (MOVIE_ID) REFERENCES MOVIES(MOVIE_ID), FOREIGN KEY (BOOKING_ID) REFERENCES
BOOKING(BOOKING_ID), PRIMARY KEY (FOOD_ID));


--Add the following constraint after the data values have been entered into the tables.

ALTER TABLE MAKE_BOOKING ADD CONSTRAINT USID FOREIGN KEY (USID) REFERENCES USERS(USER_ID);
ALTER TABLE MAKE_BOOKING ADD CONSTRAINT BID FOREIGN KEY (BID) REFERENCES BOOKING(BOOKING_ID);
```

Values for the newly inserted table:

```
INSERT into FOOD_COUNTER values ('POU67', 'JUM345', '1QWE2675');
INSERT into FOOD_COUNTER values ('JFJN0', 'UP2134', 'JPN50080');
INSERT into FOOD_COUNTER values ('SIUB3', 'LIO897', 'BSK60106');
INSERT into FOOD_COUNTER values ('IUEFB', 'RUS332', 'KRP15060');
INSERT into FOOD_COUNTER values ('WOUE2', 'NUN444', 'BTM36080');
INSERT into FOOD_COUNTER values ('POI88', 'TEN976', 'JPN50304');
INSERT into FOOD_COUNTER values ('PKO00', 'PAR866', 'RRN87426');
INSERT into FOOD_COUNTER values ('JNNW1', 'CHO117', 'MGR48769');
```

Displaying the table FOOD_COUNTER:

```
cs068_079_080=# select * from food_counter;
 food_id | movie_id | booking_id
---------+----------+------------
 POU67   | JUM345   | 1QWE2675
 JFJN0   | UP2134   | JPN50080
 SIUB3   | LIO897   | BSK60106
 IUEFB   | RUS332   | KRP15060
 WOUE2   | NUN444   | BTM36080
 POI88   | TEN976   | JPN50304
 PKO00   | PAR866   | RRN87426
 JNNW1   | CHO117   | MGR48769
(8 rows)
```

QUERIES USING NEW SCHEMA:

Query 1: Find the movie_id of the movie with the food_id = ‘POU67’

```
cs068_079_080=# select movie_id from food_counter where food_id = 'POU67';
 movie_id
----------
 JUM345
(1 row)
```

Query 2: Obtain the show_id of all food_id

```
cs068_079_080=# select food_id, show_id from food_counter, show where food_counter.movie_id = show.movie_id;
 food_id | show_id
---------+---------
 POU67   | AKL123I
 JFJN0   | GHJ5OOC
 SIUB3   | IKLD23Q
 IUEFB   | HJK2217
 WOUE2   | CVNB09Z
 POI88   | SD67FRT
 PKO00   | UI77PLM
 JNNW1   | WTTYH89
(8 rows)
```

Query 3: Obtain the names and ages of users along with the corresponding food_id

```
cs068_079_080=# select name, age, food_id from food_counter, users, booking where food_counter.booking_id = booking.booking_id and booking.user_id = users.user_id;
   name   | age | food_id
----------+-----+---------
 ANIRUDH  |  18 | POU67
 ANKITA   |  20 | JFJN0
 APOORVA  |  23 | SIUB3
 ANVIKA   |  31 | IUEFB
 RAHUL    |  54 | WOUE2
 MAYA     |  17 | POI88
 BHARATH  |  69 | PKO00
 KALYANAM |  81 | JNNW1
(8 rows)
```

Query 4: Get all movie names using food_id

```
cs068_079_080=# select movie_name from movies,food_counter where food_counter.movie_id = movies.movie_id;
 movie_name
------------
 JUMANJI
 UP
 LIONKING
 RUSHHOUR
 NUN
 TENET
 PARASITE
 CHOPSTICKS
(8 rows)
```

Query 5: Get the cost of tickets and their corresponding booking_id for all food_id values

```
cs068_079_080=# select cost, booking.booking_id, food_id from booking, food_counter where food_counter.booking_id = booking.booking_id
 cost | booking_id | food_id
------+------------+---------
 1000 | 1QWE2675   | POU67
  900 | JPN50080   | JFJN0
  800 | BSK60106   | SIUB3
  600 | KRP15060   | IUEFB
  400 | BTM36080   | WOUE2
 1600 | JPN50304   | POI88
  900 | RRN87426   | PKO00
  400 | MGR48769   | JNNW1
(8 rows)
```

MIGRATING THE DATABSE TO AN ALTERNATIVE ONE DUE TO PERFORMANCE ISSUES:

The process of transferring data from one or more source databases to one or more target databases is known as database migration. When a migration is complete, the dataset in the source databases is fully replicated in the target databases, however it may be reorganized.

Any data migration will include at least the transform and load steps in the extract/transform/load (ETL) process. This means that extracted data must be processed through a sequence of functions before being fed into a destination place.

- Before the execution, a backup of the data must be maintained so that the data won't be lost even if something goes wrong during the installation.

- The Data Migration Strategy chosen must be followed.

- Testing the data migration during the planning and design phases, as well as during implementation and maintenance, ensures that the desired outcome will be achieved eventually.

Steps in a Data Migration Strategy:

Exploring and Assessing the Source

There may be a large amount of data with many fields, some of which will not need to be mapped to the target system. There may also be missing data fields within a source, necessitating the use of data from another source to fill in the gaps. It is necessary to analyze what should be migrated and what should not. Performing a data audit on the information included therein will be helpful. If there are poorly populated fields, a large number of partial data pieces, inaccuracies, or other issues, examining whether the data needs to be migrated in the first place might be necessary.

Defining and Designing the Migration

Organizations define the type of migration they want to do — big bang or trickle — during the design phase. This also entails laying out the solution's technical architecture and describing the migration procedures.

We can begin to identify timescales and any project problems by considering the design, the data to be pulled over, and the target system. The entire project should be documented by the end of this step.

It's critical to think about data security plans when planning. Protection should be threaded throughout the strategy for any data that needs to be protected.

Building the Migration Solution

This involves breaking the data down into subsets and creating one category at a time, then testing it. It might make sense to develop and test in parallel if an organization is working on a very large migration.

Conducting a Live Test

The testing process does not end when the code has been tested during the build step. To confirm the quality of the implementation and completeness of the application, it's critical to test the data migration design with real data.

Flipping the Switch

Following final testing, implementation can begin in the manner specified in the plan.

Auditing

Setting up a way to audit the data once the implementation is live to ensure that the migration is accurate.

Improvements that can be made to the data before migrating the database:

- Ensure scripts are idempotent
- Ensure scripts are immutable
- Guard against database drift
- Perform early and frequent testing
- Fail Gracefully
- Introduce governance checks early in the cycle

BUSINESS/APPLICATION CHANGES THAT MIGHT LEAD TO SCHEMA CHANGE, CONSTRAINT CHANGES OR MIGRATION OF DATABASE:

- The introduction of branches in a particular type of cinema company and a larger number of screens to choose from will require constant updating in the movie booking database.
- Multiple booking from the same user will need accommodation.
- Database migration projects usually include refactoring of the application and database code, and also the schema, which is a time-consuming, iterative process. The refactoring process can

take anywhere from a few weeks to several months, depending on the complexity of your application and database.

- Once the database is extended to a larger system if can be shifted to a cloud system so that multiple changes can be made to the system at once.
- Database Migration also requires extensive network knowledge which has to be taken under consideration during data migration.

MIGRATING FROM POSTGRESQL TO A NO – SQL VARIETY:

The same database system can be implemented easily without glitches using the graph based database Neo4j. Neo4j would be an apt choice considering the amount of dependencies in our database between all tables. The node creation and display provides a better visualization of the physical schema/internal schema of the entire database allowing easy and efficient querying.

An alternative choice to Neo4j would be to use MongoDB (document based database). Using MongoDB allows us to use the very popular MERN stack for both frontend and backend development.