



2024-09-23

DATABASES 2

LAB 1 – RELATIONAL DATABASE DESIGN



Paulina Czarnota C21365726

INTRODUCTION

This report examines relational database modeling through a practical approach using PostgreSQL. It covers the design and implementation of entity-relationship (ER) diagrams, the translation to relational models, and the deployment of the physical database structure. The exercises included in the report illustrate the practical application of these concepts through real-world scenarios, enhancing the understanding of efficient and scalable database design principles.

EXERCISE 1 – DB MODELLING OF TOYS

PROBLEM DESCRIPTION

The database contains a list of toys, each identified by a unique ID, name, and price. Different toys possess specific attributes that are unique to each type, such as:

- **Toy Car:**
 - engine_size
 - petrol_or_diesel
- **Teddy Bear:**
 - material
 - age

To store this information efficiently, I propose using Single Table Inheritance (STI). In this model, a parent table (toy) contains common attributes, while separate child tables (car and teddy) store unique attributes.

TABLE STRUCTURES

Toy Table:

| | A-Z uid | A-Z name | 123 price | 123 engine_size | A-Z petrol_or_diesel | 123 age | A-Z material |
|---|---------|------------|-----------|-----------------|----------------------|---------|--------------|
| 1 | 860 | Teddy Bear | 14 | [NULL] | [NULL] | 3 | Felt |
| 2 | 310 | Toy Car | 10 | 1.8 | Petrol | [NULL] | [NULL] |

POSTGRESQL IMPLEMENTATION

-- Dropping tables if they exist to avoid conflicts

DROP TABLE IF EXISTS car CASCADE;

DROP TABLE IF EXISTS teddy CASCADE;

DROP TABLE IF EXISTS toy CASCADE;

-- Creating the 'toy' table with common attributes

```
CREATE TABLE toy (  
    uid VARCHAR(20) PRIMARY KEY UNIQUE NOT NULL,  
    name VARCHAR(20),  
    price DECIMAL(10, 2) -- Using DECIMAL for precise price representation  
);
```

-- Creating the 'teddy' table for specific attributes of Teddy Bears

```
CREATE TABLE teddy (  
    uid VARCHAR(20) PRIMARY KEY UNIQUE NOT NULL,  
    age INT,  
    material VARCHAR(20),  
    FOREIGN KEY (uid) REFERENCES toy(uid) ON DELETE CASCADE  
);
```

-- Creating the 'car' table for specific attributes of Toy Cars

```
CREATE TABLE car (  
    uid VARCHAR(20) PRIMARY KEY UNIQUE NOT NULL,  
    engine_size DECIMAL(3, 1),  
    petrol_or_diesel VARCHAR(20),  
    FOREIGN KEY (uid) REFERENCES toy(uid) ON DELETE CASCADE
```

```
);
```

```
-- Inserting data into the 'toy' table
```

```
INSERT INTO toy (uid, name, price) VALUES
```

```
('860', 'Teddy Bear', 14.00),
```

```
('310', 'Toy Car', 10.00);
```

```
-- Inserting data into the 'teddy' table
```

```
INSERT INTO teddy (uid, age, material) VALUES
```

```
('860', 3, 'Felt');
```

```
-- Inserting data into the 'car' table
```

```
INSERT INTO car (uid, engine_size, petrol_or_diesel) VALUES
```

```
('310', 1.8, 'Petrol');
```

```
-- Querying data to retrieve all toy details along with specific attributes
```

```
SELECT
```

```
    t.uid,
```

```
    t.name,
```

```
    t.price,
```

```
    c.engine_size,
```

```
    c.petrol_or_diesel,
```

```
    te.age,
```

```
    te.material
```

```
FROM
```

```
    toy t
```

```
LEFT JOIN
```

```
    car c ON t.uid = c.uid
```

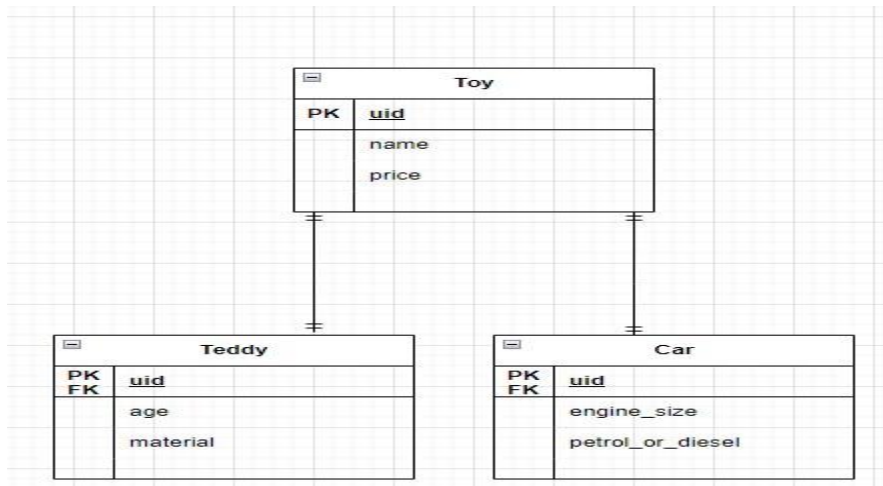
LEFT JOIN

```
teddy te ON t.uid = te.uid;
```

EXPLANATION

- **Flexibility:** Using separate tables allows for easy additions of new toy types or attributes without modifying existing structures.
- **Performance:** This design minimizes unnecessary NULL fields by storing attributes only when relevant. However, it's important to note that while Single Table Inheritance (STI) can streamline data retrieval for simple queries, it might complicate more complex queries, especially as the dataset grows larger. This could lead to performance issues in terms of query execution time, as the database may need to process multiple joins or handle increased data volumes.
- **Storage Efficiency:** Each toy type stores only the pertinent attributes, reducing overall storage requirements. Yet, if a significant number of toys share a large number of unique attributes, this approach might lead to a proliferation of tables, which can complicate the schema and increase maintenance overhead.
- **Maintainability:** Changes to specific toy types can be made independently, simplifying the maintenance process. However, in scenarios where many toy types share similar attributes, STI could create redundancy in the database schema. This redundancy might necessitate additional consideration for maintaining data consistency and integrity across multiple tables.

ER DIAGRAM



ER DIAGRAM EXPLANATION

1. Toy Entity

- **Description:** The Toy entity acts as the parent table in this database model, encapsulating all common attributes shared by different types of toys. It includes fields that provide essential information about each toy.
- **Key Attributes:**
 - **uid:** Serves as the primary key for this table, uniquely identifying each toy in the database. This ensures that each toy record is distinct and can be referenced by other entities.

2. Teddy Entity

- **Description:** The Teddy entity is a child table that holds attributes specific to teddy bears. This structure allows for the extension of the Toy entity by adding unique characteristics relevant to teddy bears.
- **Key Attributes:**
 - **uid:** Functions as both the primary key for the Teddy table and a foreign key referencing the uid in the Toy table. This dual role maintains referential integrity, ensuring that each teddy bear corresponds to a valid toy record.

3. Car Entity

- **Description:** Similar to the Teddy entity, the Car entity is a child table that contains attributes particular to toy cars. This design allows for the incorporation of specific details about toy cars while maintaining the shared structure with the Toy entity.
- **Key Attributes:**
 - **uid:** Serves as the primary key for the Car table and as a foreign key referencing the Toy table. This linkage ensures that each toy car is accurately associated with its corresponding toy record.

Relationships

- **Toy to Teddy:**
 - **Type:** One-to-One (1:1) relationship

- **Description:** Each instance of the Teddy entity is uniquely associated with a specific instance of the Toy entity. The foreign key uid in the Teddy table directly references the primary key uid in the Toy table, establishing a direct link between the two entities.
- **Toy to Car:**
 - **Type:** One-to-One (1:1) relationship
 - **Description:** Each instance of the Car entity corresponds uniquely to a specific instance of the Toy entity. Similar to the Teddy relationship, the uid in the Car table serves as a foreign key, referencing the primary key uid in the Toy table, ensuring a clear association between toy cars and their parent toy records.

CONCLUSION

In summary, the proposed database design using Single Table Inheritance (STI) effectively addresses the diverse requirements of toy attributes while ensuring flexibility, performance, storage efficiency, and maintainability. However, as with any design approach, it is crucial to remain aware of potential limitations, especially concerning complex queries and scalability. Future iterations of this database could explore alternative inheritance strategies and indexing improvements to enhance performance further.

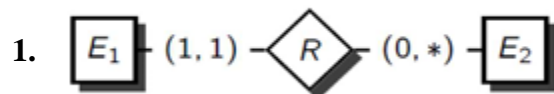
FUTURE CONSIDERATIONS

As the database grows or new toy types are added, reassessing the schema may be necessary. Investigating Class Table Inheritance (CTI) or Concrete Table Inheritance (CIT) could optimize performance and simplify maintenance. Additionally, refining indexing strategies will be crucial for enhancing data retrieval efficiency.

EXERCISE 2 – FROM ER DIAGRAM TO RELATIONAL MODEL

PART A: SELECTED SITUATIONS

SITUATION 1



Description: E1 must exist and can relate to multiple instances of E2.

Relational Model:

- **E1:** K1 as Primary Key
- **E2:** K2 as Primary Key, K1 as Foreign Key (references E1)

TABLE STRUCTURES

E1_relation Table:

| Grid | Text |
|--------|------|
| 123 k1 | 1 |
| | 2 |

POSTGRESQL IMPLEMENTATION

```
DROP TABLE IF EXISTS E2_relation CASCADE;
```

```
DROP TABLE IF EXISTS E1_relation CASCADE;
```

```
CREATE TABLE E1_relation (
```

```
    K1 SERIAL PRIMARY KEY
```


);

CREATE TABLE E2_relation (

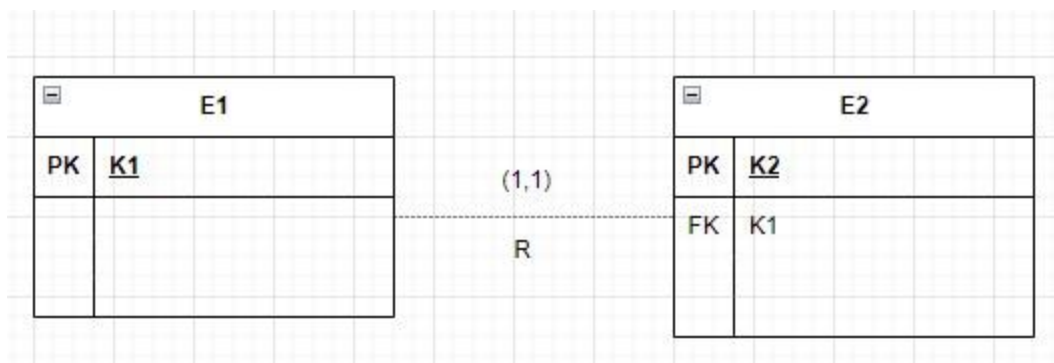
K2 SERIAL PRIMARY KEY,

K1 INT NOT NULL,

FOREIGN KEY (K1) REFERENCES E1_relation(K1) ON DELETE CASCADE

);

ER DIAGRAM



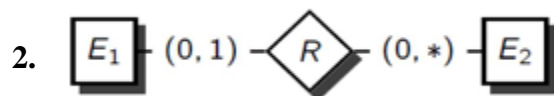
ER DIAGRAM EXPLANATION

- E1 has a primary key K1.
- E2 has a primary key K2 and a foreign key K1 referencing E1(K1).

Relationship R:

- (1,1) cardinality on both E1 and E2, meaning each E1 instance must have exactly one associated E2, and each E2 instance must have exactly one associated E1.

SITUATION 2



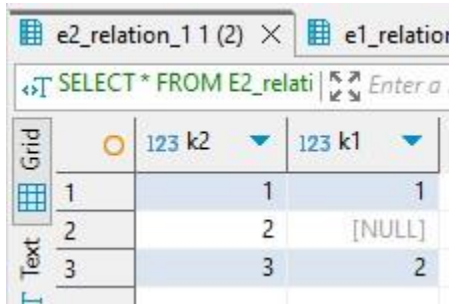
Description: E1 can exist optionally, while E2 can have multiple instances.

Relational Model:

- **E1:** K1 as Primary Key, nullable
- **E2:** K2 as Primary Key, K1 as Foreign Key, nullable

TABLE STRUCTURES

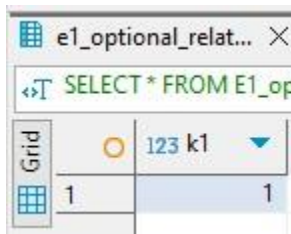
E2_relation Table:



The screenshot shows a database query window with the title 'e2_relation_1 1 (2)'. The query bar contains 'SELECT * FROM E2_relati'. Below the query bar is a table with three columns: an index column, '123 k2', and '123 k1'. The table has three rows of data.

| | 123 k2 | 123 k1 |
|---|--------|--------|
| 1 | 1 | 1 |
| 2 | 2 | [NULL] |
| 3 | 3 | 2 |

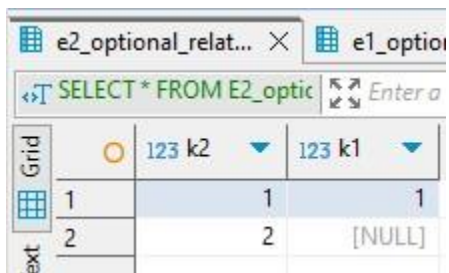
E1_optional Table:



The screenshot shows a database query window with the title 'e1_optional_rel...'. The query bar contains 'SELECT * FROM E1_op'. Below the query bar is a table with two columns: an index column and '123 k1'. The table has one row of data.

| | 123 k1 |
|---|--------|
| 1 | 1 |

E2_optional Table:



The screenshot shows a database query window with the title 'e2_optional_rel...'. The query bar contains 'SELECT * FROM E2_optic'. Below the query bar is a table with three columns: an index column, '123 k2', and '123 k1'. The table has two rows of data.

| | 123 k2 | 123 k1 |
|---|--------|--------|
| 1 | 1 | 1 |
| 2 | 2 | [NULL] |

POSTGRESQL IMPLEMENTATION

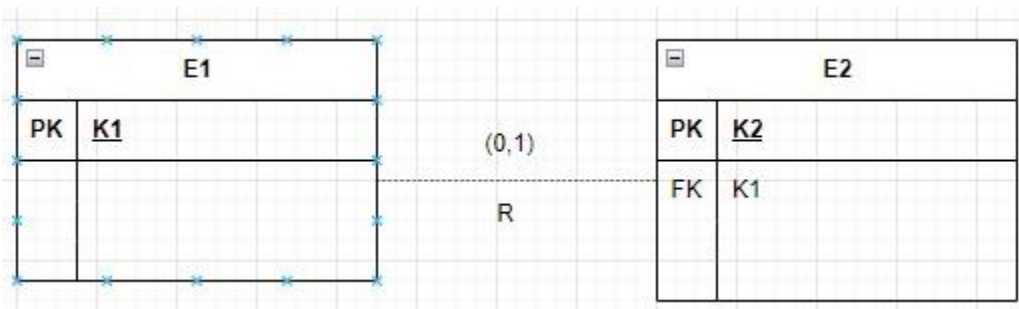
DROP TABLE IF EXISTS E2_optional CASCADE;

DROP TABLE IF EXISTS E1_optional CASCADE;

```
CREATE TABLE E1_optional (
    K1 SERIAL PRIMARY KEY
);
```

```
CREATE TABLE E2_optional (
    K2 SERIAL PRIMARY KEY,
    K1 INT,
    FOREIGN KEY (K1) REFERENCES E1_optional(K1) ON DELETE SET NULL
);
```

ER DIAGRAM



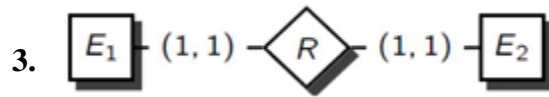
ER DIAGRAM EXPLANATION

- E1 has a primary key K1.
- E2 has a primary key K2 and a foreign key K1 referencing E1(K1).

Relationship R:

- (0,1) cardinality on the E1 side, meaning E1 can exist optionally and each instance of E1 can relate to E2 at most once.
- (0,*) cardinality on the E2 side, meaning each E1 can be related to zero or more instances of E2.

SITUATION 3



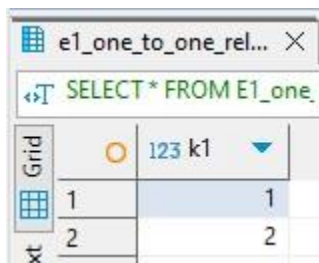
Description: Both E1 and E2 must exist, with each having one instance.

Relational Model:

- **E1:** K1 as Primary Key
- **E2:** K2 as Primary Key, K1 as Foreign Key, unique

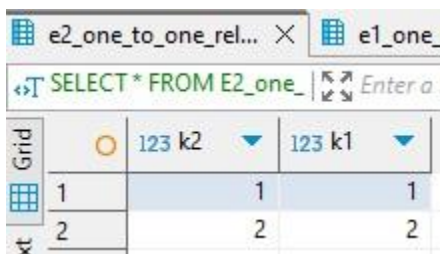
TABLE STRUCTURES

E1_one_to_one Table:



| Grid | 123 k1 |
|------|--------|
| 1 | 1 |
| 2 | 2 |

E2_one_to_one Table:



| Grid | 123 k2 | 123 k1 |
|------|--------|--------|
| 1 | 1 | 1 |
| 2 | 2 | 2 |

POSTGRESQL IMPLEMENTATION

```
DROP TABLE IF EXISTS E2_one_to_one CASCADE;
```

```
DROP TABLE IF EXISTS E1_one_to_one CASCADE;
```

```
CREATE TABLE E1_one_to_one (
```

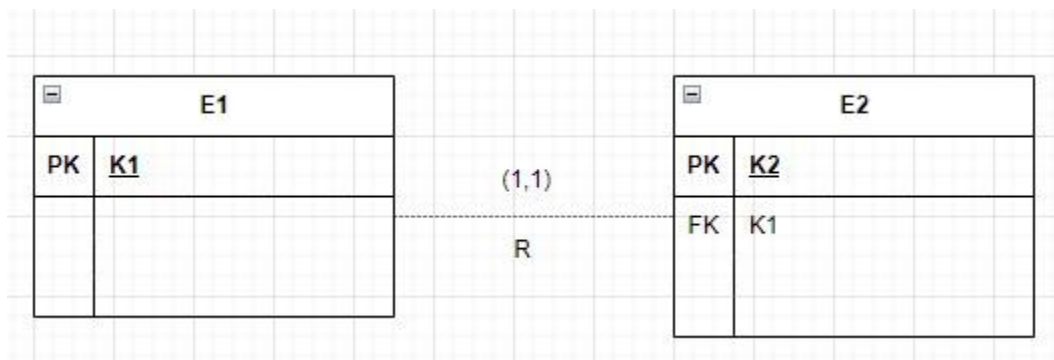
```

    K1 SERIAL PRIMARY KEY
);

CREATE TABLE E2_one_to_one (
    K2 SERIAL PRIMARY KEY,
    K1 INT UNIQUE NOT NULL,
    FOREIGN KEY (K1) REFERENCES E1_one_to_one(K1) ON DELETE CASCADE
);

```

ER DIAGRAM



ER DIAGRAM EXPLANATION

- E1 has a primary key K1.
- E2 has a primary key K2 and a foreign key K1 referencing E1(K1).

Relationship R:

- (1,1) cardinality on both E1 and E2, meaning each E1 instance must have exactly one associated E2, and each E2 instance must have exactly one associated E1.

PART B: CONSIDERATION OF RELATION

Relation Description:



In this relationship, each instance of E1 must exist and can be associated with one or more instances of E2. Conversely, each instance of E2 must have one associated instance of E1.

Modeling with Relational Model:

This relationship can indeed be modeled using a relational model, where:

- E1 will have a primary key K1.
- E2 will have a primary key K2 and a non-null foreign key K1 that references E1. This ensures that every E2 instance is tied to a corresponding E1 instance.

Relational Model Structure:

- **E1:** K1 as Primary Key
- **E2:** K2 as Primary Key, K1 as Foreign Key (NOT NULL)

TABLE STRUCTURES

E1_relation Table:

| Grid | 123 k1 |
|------|--------|
| 1 | 1 |

E2_relation Table:

| Grid | 123 k2 | 123 k1 |
|------|--------|--------|
| 1 | 1 | 1 |
| 2 | 2 | 1 |

ORACLE IMPLEMENTATION:

```
DROP TABLE E2_relation CASCADE CONSTRAINTS;
```

```
DROP TABLE E1_relation CASCADE CONSTRAINTS;
```

```
CREATE TABLE E1_relation (
```

```
    K1 NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY
```

```
);
```

```
CREATE TABLE E2_relation (
```

```
    K2 NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
```

```
    K1 NUMBER NOT NULL,
```

```
    FOREIGN KEY (K1) REFERENCES E1_relation(K1) ON DELETE CASCADE
```

```
);
```

Justification:

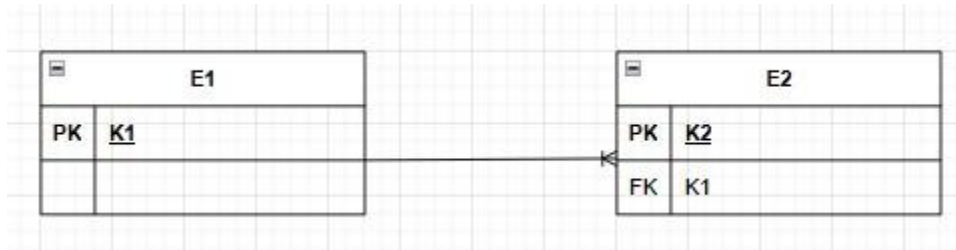
- The foreign key K1 in E2 ensures that every instance of E2 is associated with one and only one instance of E1, which adheres to the (1, *) cardinality.
- The use of ON DELETE CASCADE maintains referential integrity, automatically removing all related E2 instances when an E1 instance is deleted.

FEASIBILITY OF MODELING WITH ORACLE

Is it possible to model it with a relational model and implement it with Oracle “create table” statements? Why?

Yes, it is possible to model this relationship using a relational model and implement it with Oracle "create table" statements. The implementation allows for the use of primary keys and foreign keys to maintain relationships between the tables, ensuring that referential integrity is preserved. The structure defined adheres to the requirements of the relational database model, where the foreign key in E2 is marked as NOT NULL to enforce the requirement that each E2 instance must be associated with an E1 instance.

ER DIAGRAM



ER DIAGRAM EXPLANATION

- E1_relation has a primary key K1.
- E2_relation has a primary key K2 and a foreign key K1 referencing E1_relation(K1).

Relationship R:

- This relationship is a (1,*) cardinality on the E1_relation side, meaning each E1 instance can be associated with zero or more E2 instances.
- On the E2_relation side, the cardinality is (1,1), meaning each E2 instance must be associated with exactly one E1 instance.

EXERCISE 3 – DESIGN A RELATIONAL MODEL FOR A CINEMA DATABASE

RELATIONAL MODEL DESIGN

To design a relational database for a cinema management system, I will identify the main entities and their relationships:

1. Cinemas

- **Attributes:**

- cinema_id (Primary Key)
- location
- contact_number
- name
- number_of_screens

2. Movies

- **Attributes:**

- movie_id (Primary Key)
- title
- duration
- rating

3. Screens

- **Attributes:**

- screen_id (Primary Key)
- cinema_id (Foreign Key)
- number_of_seats

4. Showings

- **Attributes:**

- show_id (Primary Key)
- movie_id (Foreign Key)
- screen_id (Foreign Key)
- show_time

5. Customers

- **Attributes:**

- customer_id (Primary Key)

- username
- password
- dob

6. Bookings

Attributes:

- booking_id (Primary Key)
- customer_id (Foreign Key)
- show_id (Foreign Key)
- adult_tickets
- child_tickets
- total_price
- seat_numbers (Can be stored as a comma-separated string)

TABLE STRUCTURES

Cinemas Table:

| cinemas 1 × movies 1 (2) screens 1 (3) showings 1 (4) customers 1 (5) bo | | | | | | |
|---|---------------|--------------|-----------------|------------|-----------------|--|
| SELECT * FROM cinemas Enter a SQL expression to filter results (use Ctrl+Space) | | | | | | |
| Grid | 123 cinema_id | A-Z location | A-Z contact_num | A-Z name | 123 number_of_s | |
| 1 | 1 | New York | 123456789 | Cinema One | 5 | |
| 2 | 2 | Los Angeles | 987654321 | Cinema Two | 8 | |

Movies Table:

| cinemas 1 movies 1 (2) × screens 1 (3) showings 1 (4) | | | | |
|--|--------------|-----------|--------------|------------|
| SELECT * FROM movies Enter a SQL expression to filter results (use Ctrl+Space) | | | | |
| Grid | 123 movie_id | A-Z title | 123 duration | A-Z rating |
| 1 | 1 | Inception | 148 | PG-13 |
| 2 | 2 | Toy Story | 100 | G |

Screens Table:

| cinemas 1 movies 1 (2) screens 1 (3) × showings 1 (4) | | | |
|---|---------------|---------------|---------------------|
| SELECT * FROM screens Enter a SQL expression to filter results (use Ctrl+Space) | | | |
| Grid | 123 screen_id | 123 cinema_id | 123 number_of_seats |
| 1 | 1 | 1 | 100 |
| 2 | 2 | 2 | 150 |

Showings Table:

cinemas 1

movies 1 (2)

screens 1 (3)

showings 1 (4)

×

custom

⌕

SELECT * FROM showings

⌵⌵ Enter a SQL expression to filter results (use Ctrl+Space)

Grid

123 show_id

▼

123 movie_id

▼

123 screen_id

▼

⌚ show_time

▼

1

1

1

24-10-26 14:00:00.000

2

2

2

24-10-26 16:00:00.000

Customers Table:

cinemas 1

movies 1 (2)

screens 1 (3)

showings 1 (4)

customers 1 (5)

SELECT * FROM customers

Enter a SQL expression to filter results (use Ctrl+Space)

Grid

123 customer_id

A-Z username

A-Z password

🕒 dob

1

1

john_doe

password123

1985-05-12

2

2

jane_smith

securepass456

1990-08-23

Bookings Table:

cinemas 1

movies 1 (2)

screens 1 (3)

showings 1 (4)

customers 1 (5)

bookings 1 (6) X

SELECT * FROM bookings

Enter a SQL expression to filter results (use Ctrl+Space)

| Grid | 123 booking_id | 123 customer_id | 123 show_id | 123 adult_tickets | 123 child_tickets | 123 total_price | A-Z seat_number |
|------|----------------|-----------------|-------------|-------------------|-------------------|-----------------|-----------------|
| 1 | 1 | 1 | 1 | 2 | 1 | 30 | 3,4,5 |
| 2 | 2 | 2 | 2 | 1 | 2 | 45 | 10,11,12 |

POSTGRESQL IMPLEMENTATION

-- Dropping existing tables to avoid conflicts

DROP TABLE IF EXISTS bookings CASCADE;

DROP TABLE IF EXISTS customers CASCADE;

DROP TABLE IF EXISTS showings CASCADE;

DROP TABLE IF EXISTS screens CASCADE;

DROP TABLE IF EXISTS movies CASCADE;

DROP TABLE IF EXISTS cinemas CASCADE;

-- Creating the 'cinemas' table

```
CREATE TABLE cinemas (  
    cinema_id SERIAL PRIMARY KEY,  
    location VARCHAR(100),  
    contact_number VARCHAR(15),  
    name VARCHAR(100),  
    number_of_screens INT  
);
```

-- Creating the 'movies' table

```
CREATE TABLE movies (  
    movie_id SERIAL PRIMARY KEY,  
    title VARCHAR(100),  
    duration INT,  
    rating VARCHAR(5)  
);
```

-- Creating the 'screens' table

```
CREATE TABLE screens (  
    screen_id SERIAL PRIMARY KEY,  
    cinema_id INT,  
    number_of_seats INT,  
    FOREIGN KEY (cinema_id) REFERENCES cinemas(cinema_id) ON DELETE CASCADE  
);
```

-- Creating the 'showings' table

```
CREATE TABLE showings (  
    show_id SERIAL PRIMARY KEY,
```

```
movie_id INT,  
screen_id INT,  
show_time TIMESTAMP,  
FOREIGN KEY (movie_id) REFERENCES movies(movie_id) ON DELETE CASCADE,  
FOREIGN KEY (screen_id) REFERENCES screens(screen_id) ON DELETE CASCADE  
);
```

-- Creating the 'customers' table

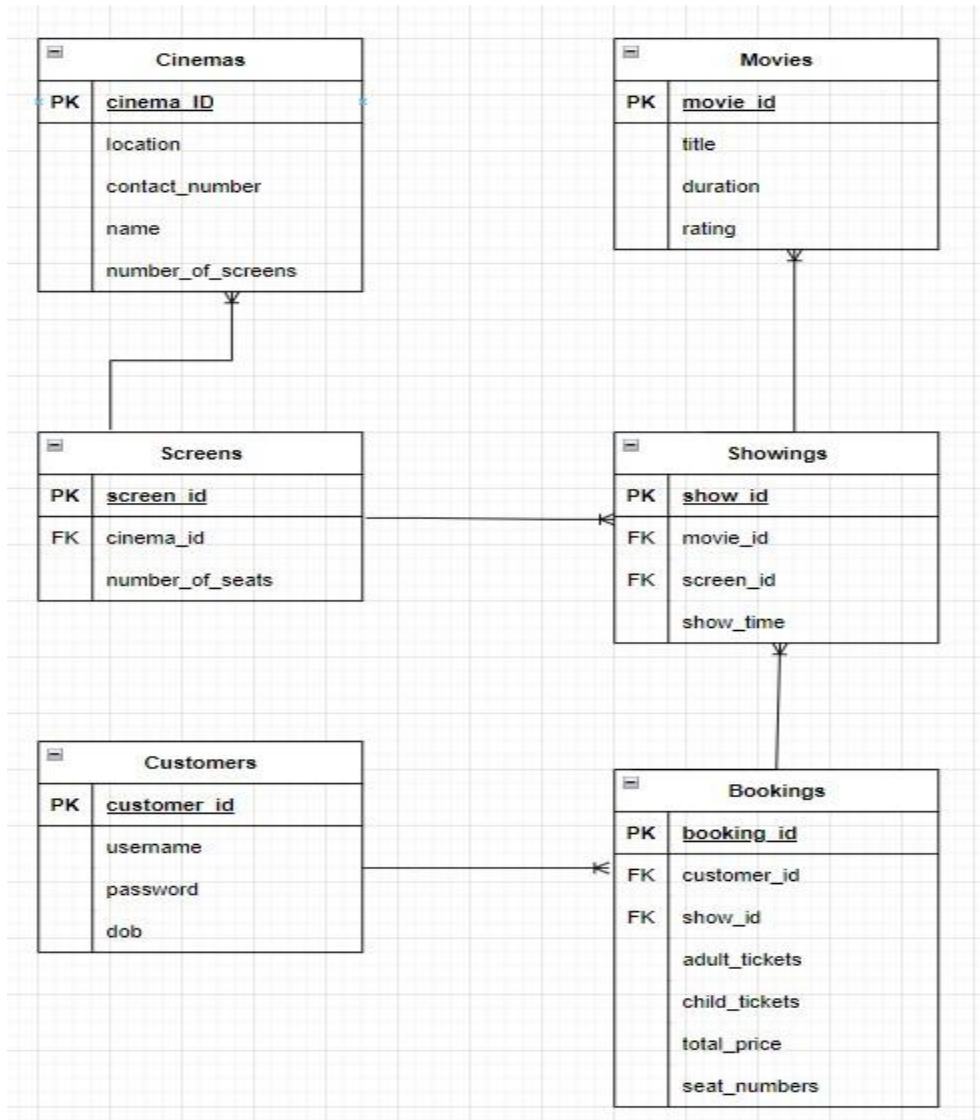
```
CREATE TABLE customers (  
    customer_id SERIAL PRIMARY KEY,  
    username VARCHAR(50) UNIQUE NOT NULL,  
    password VARCHAR(50) NOT NULL,  
    dob DATE NOT NULL  
);
```

-- Creating the 'bookings' table

```
CREATE TABLE bookings (  
    booking_id SERIAL PRIMARY KEY,  
    customer_id INT,  
    show_id INT,  
    adult_tickets INT,  
    child_tickets INT,  
    total_price DECIMAL(10, 2),  
    seat_numbers VARCHAR(255),  
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id) ON DELETE  
CASCADE,  
    FOREIGN KEY (show_id) REFERENCES showings(show_id) ON DELETE CASCADE
```

);

ER DIAGRAM



ER DIAGRAM EXPLANATION

1. Cinemas Entity

- **Description:** This is the parent table that contains attributes for cinema locations.
- **Key Attribute:**
 - **cinema_id:** Primary key that uniquely identifies each cinema.

- **Relationship:**
 - **Connected to Screens:** One-to-Many (A cinema has multiple screens).

2. Movies Entity

- **Description:** This table contains attributes specific to films, such as title, duration, and rating.
- **Key Attribute:**
 - **movie_id:** Primary key that uniquely identifies each movie.
- **Relationship:**
 - **Connected to Showings:** One-to-Many (A movie can have multiple showings).

3. Screens Entity

- **Description:** This child table represents individual screens within cinemas.
- **Key Attribute:**
 - **screen_id:** Primary key.
- **Foreign Key:**
 - **cinema_id:** Foreign key referencing the cinema_id in the Cinemas table.
- **Relationship:**
 - **Connected to Cinemas:** Many-to-One (Multiple screens belong to one cinema).
 - **Connected to Showings:** One-to-Many (A screen can host multiple showings).

4. Showings Entity

- **Description:** This table records scheduled showings of movies.
- **Key Attribute:**
 - **show_id:** Primary key.
- **Foreign Keys:**
 - **movie_id:** Foreign key referencing the Movies table.
 - **screen_id:** Foreign key referencing the Screens table.

- **Relationship:**
 - **Connected to Movies:** Many-to-One (Multiple showings can belong to one movie).
 - **Connected to Screens:** Many-to-One (Multiple showings can occur on one screen).
 - **Connected to Bookings:** One-to-Many (A showing can have multiple bookings).

5. Customers Entity

- **Description:** This table captures customer information, including username and date of birth.
- **Key Attribute:**
 - **customer_id:** Primary key that uniquely identifies each customer.
- **Relationship:**
 - **Connected to Bookings:** One-to-Many (A customer can make multiple bookings).

6. Bookings Entity

- **Description:** This child table tracks customer bookings for showings.
- **Key Attribute:**
 - **booking_id:** Primary key.
- **Foreign Keys:**
 - **customer_id:** Foreign key referencing the Customers table.
 - **show_id:** Foreign key referencing the Showings table.
- **Relationship:**
 - **Connected to Customers:** Many-to-One (Multiple bookings can belong to one customer).
 - **Connected to Showings:** Many-to-One (Multiple bookings can be made for one showing).