CPS510 - A10 Report

By: Isaac Martin, Bowie Chau, Hachi Ndu

Instructor: Dr. S. B. Tajali

TA: George Lopez

Section #: 06

Lab Number: 10

Group #: 12

Nov 27, 2024

Title: Online Movie Store

Content:

ASSIGNMENT 1

Description

The online movie store will be a dynamic web application where users can sign up, log in, and access a comprehensive catalogue of movies. After successfully logging in, users can be presented with the full selection of films available on the site. The application will leverage multiple databases containing critical information contributing to the platform's functionality.

The first database will manage all movie-related data, including a unique ID for each film, a poster image (stored as a JPEG), the movie title, director, cast, description, price and rating. Additionally, users can rate movies they've watched, and their preferences will be used to recommend similar films.

The second database will be the user's information, which will be securely stored, including their name, date of birth, username, hash and salted password (for added security), and email address. This will ensure a personalized user experience while maintaining the highest level of protection for sensitive data.

The third database will include admin information, which consists of the admin's username, email address, and password. This is an import database because only the admin can make changes on the website using an authentication process. The admin will be able to add and delete movies and change the rating of the film and its description.

The fourth database will contain customer payment information, including encrypted credit card details and billing addresses. All transactions will be securely processed using industry-standard encryption protocols, protecting sensitive financial data.

This comprehensive design ensures a scalable, secure, and personalized user experience while providing the admin with full control over the movie catalogue.

Tables

The database will consist of tables with the following attributes and types:

- account Table
    - accountID (NUMBER, Primary Key, Auto-increment)
    - username (VARCHAR2(50), NOT NULL, UNIQUE)
    - password (VARCHAR(255), NOT NULL)
    - email (VARCHAR(50), NOT NULL, UNIQUE)
- admin Table
    - adminID (NUMBER, Primary Key, Foreign Key references account.accountID)
- customer Table
    - customerID (NUMBER, Primary Key, Foreign Key references account.accountID)
    - firstName (VARCHAR(50), NOT NULL)

- lastName (VARCHAR(50), NOT NULL)
- bdayDay (NUMBER, NOT NULL)
- bdayMonth (NUMBER, NOT NULL)
- bdayYear (NUMBER, NOT NULL)

- membership Table
    - membershipID (NUMBER, Primary Key, Auto-increment)
    - customerIDFK (NUMBER, UNIQUE, NOT NULL, Foreign Key references customer.customerID)
    - startDay (NUMBER, NOT NULL)
    - startMonth (NUMBER, NOT NULL)
    - startYear (NUMBER, NOT NULL)
    - endDay (NUMBER, NOT NULL)
    - endMonth (NUMBER, NOT NULL)
    - endYear (NUMBER, NOT NULL)

- payment Table
    - paymentID (NUMBER, Primary Key, Auto-increment)
    - membershipIDFK (NUMBER, NOT NULL, Foreign Key references membership.membershipID)
    - subscriptionCharge (NUMBER, NOT NULL)
    - billingAddress (VARCHAR2(50), NOT NULL)
    - cardHolderName (VARCHAR2(50), NOT NULL)
    - paymentDay (NUMBER, NOT NULL)
    - paymentMonth (NUMBER, NOT NULL)
    - paymentYear (NUMBER, NOT NULL)

- movie Table
    - movieID (NUMBER, Primary Key, Auto-increment)
    - movieName (VARCHAR2(50), NOT NULL)
    - moviePoster (VARCHAR2(255), NOT NULL)
    - movieDescription (VARCHAR2(255), NOT NULL)
    - movieRating (NUMBER, NOT NULL)

- membership_movie Table (Junction Table: Many-to-Many between membership and movie)
    - membershipID (NUMBER, Primary Key, Foreign Key references membership.membershipID)
    - movieID (NUMBER, Primary Key, Foreign Key references movie.movieID)

- genre Table
    - genreID (NUMBER, Primary Key, Auto-increment)

- genreName (VARCHAR2(50), NOT NULL)

- movie_genre Table (Junction Table: Many-to-Many between movie and genre)
  - genreID (NUMBER, Primary Key, Foreign Key references genre.genreID)
  - movieID (NUMBER, Primary Key, Foreign Key references movie.movieID)

- director Table
  - directorID (NUMBER, Primary Key, Auto-increment)
  - directorFirstName (VARCHAR2(30), NOT NULL)
  - directorLastName (VARCHAR2(30), NOT NULL)

- movie_director Table (Junction Table: Many-to-Many between movie and director)
  - directorID (NUMBER, Primary Key, Foreign Key references director.directorID)
  - movieID (NUMBER, Primary Key, Foreign Key references movie.movieID)

Functionality

- Allow customers to manage personal accounts on the site
  - Customers will have personal accounts with details and payment information attached to them.
  - Customers can create and manage watchlists/shopping carts for future purchases.

- Manage movie inventory and advanced keyword searching
  - The site will store data about movies, such as (titles, genres, cast, directors, release dates, and descriptions).
  - Movies will be grouped into various genres and categories for advanced searching and browsing.

- Allow customers to stream movies
  - Customers will be able to order/rent movies on the site.
  - The database will track user permissions and allow verified customers access to content.

- Store payment and billing information securely
  - The site will track user memberships and expiry dates.
  - The site will track customer purchase history.
  - Customers will receive invoices and billing receipts for every purchase and membership renewal.

- Allow staff access to back-end features
  - Admins and staff can update movie entries and prices.

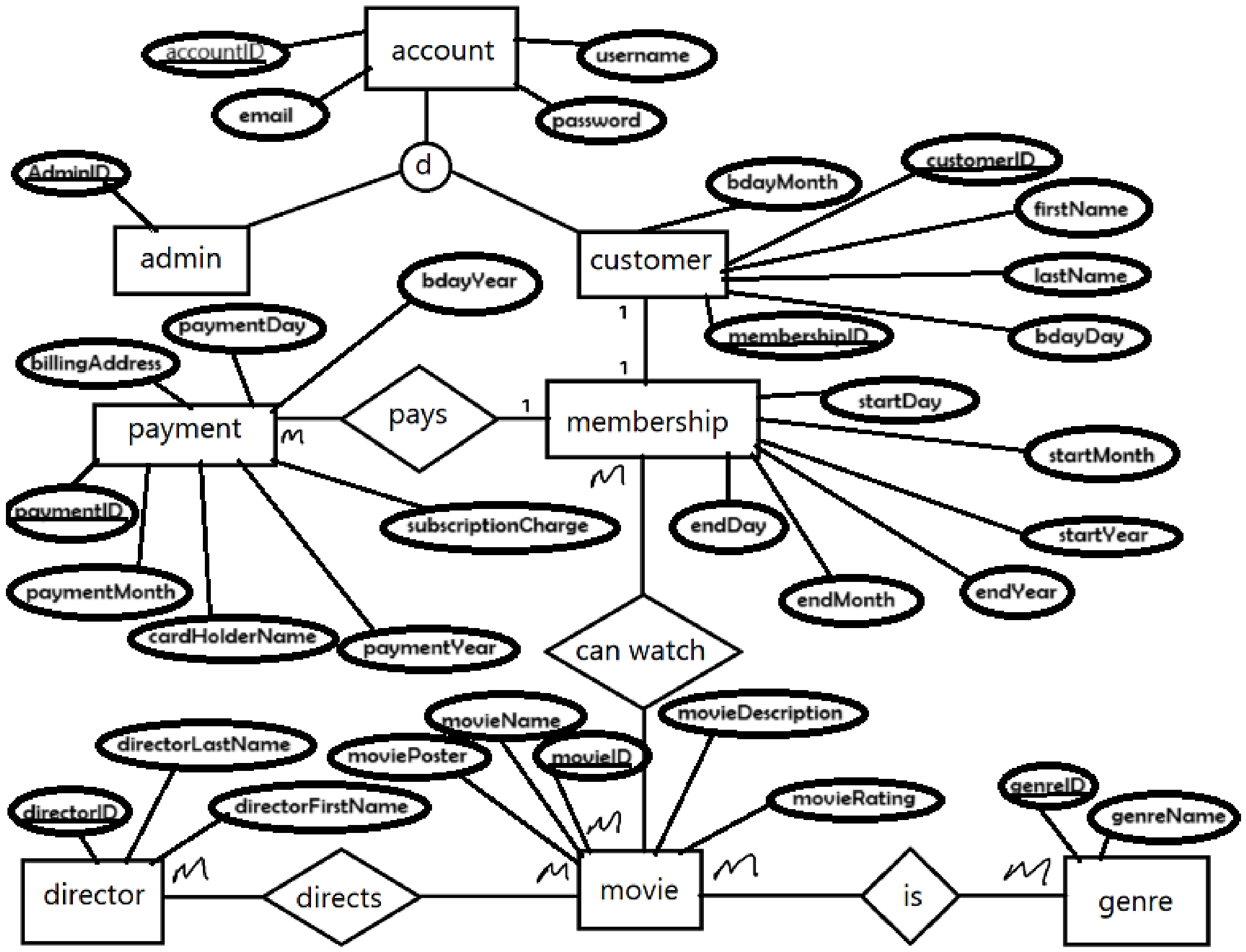
    - Admins will have access to customer support features such as customer accounts and refund management.
    - Admins will have access to the sales index for data-driven decision-making.

<u>Volume of Data</u>

- Movies
    - Number of movie files (varies depending on store offering)
    - The size of high-definition movie files may range from 1GB to 50GB
- Customers
    - Basic customer data (varies)
- Orders
    - Store roughly 5KB of data per order
- Payments
    - Payment records (roughly 5KB of data per payment made)
- Admin
    - Staff data (varies)

ASSIGNMENT 2

Below is the ER diagram used for our database after Normalizing it completely using algorithms:



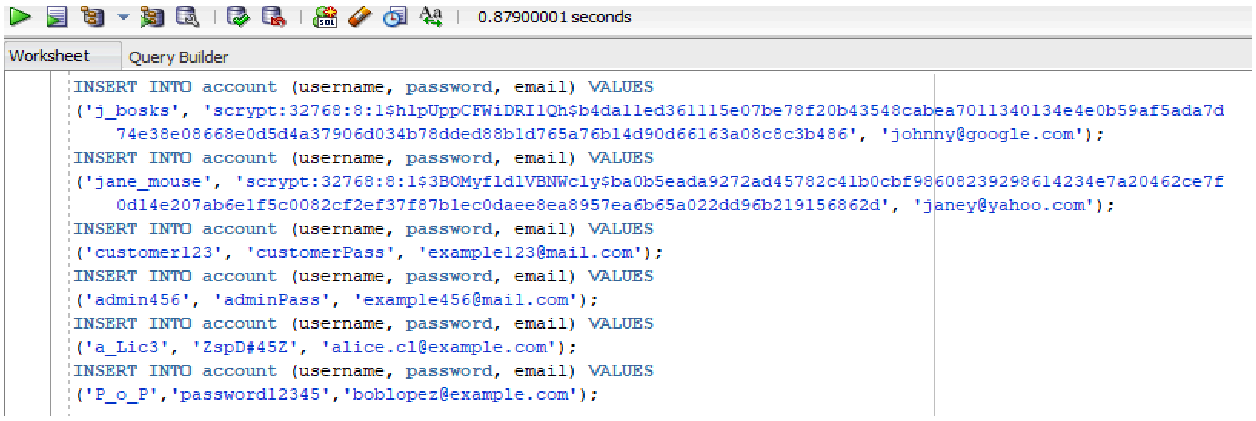
ASSIGNMENT 3

Below is the finalized code used for our database after Normalizing it completely using the Berstein algorithm.

Table Creation:

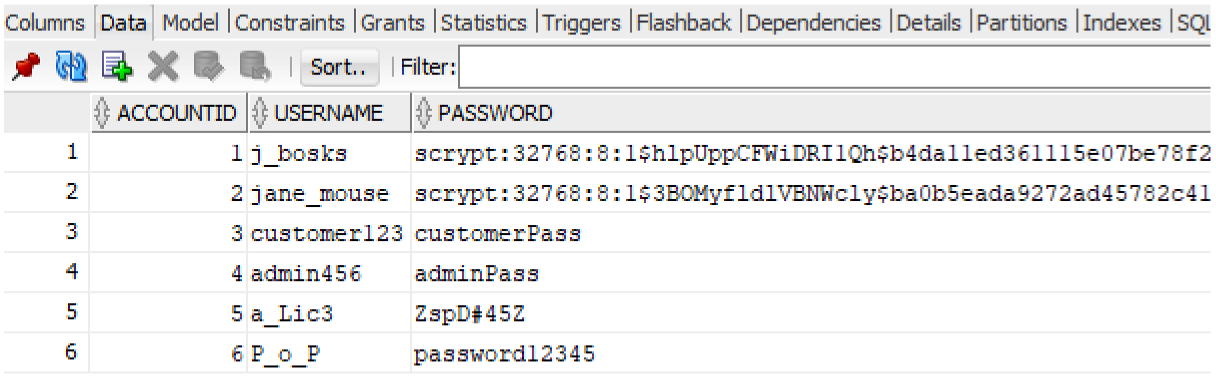
```
CREATE TABLE account(
   accountID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, -- extra stuff basically means auto increment
   username VARCHAR2(50) NOT NULL UNIQUE,
   password VARCHAR(255) NOT NULL, -- 255 space for hash
   email VARCHAR(50) NOT NULL UNIQUE
);

CREATE TABLE admin(
   adminID NUMBER PRIMARY KEY, -- no auto increment to enforce dependency on account
   FOREIGN KEY (adminID) REFERENCES account (accountID)
);

CREATE TABLE customer(
   customerID NUMBER PRIMARY KEY, -- no auto increment to enforce dependency on account
   firstName VARCHAR(50) NOT NULL,
   lastName VARCHAR(50) NOT NULL,
   bdayDay NUMBER NOT NULL,
   bdayMonth NUMBER NOT NULL,
   bdayYear NUMBER NOT NULL,
   FOREIGN KEY (customerID) REFERENCES account (accountID)
);

CREATE TABLE membership(
   membershipID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
   customerIDFK NUMBER UNIQUE NOT NULL, -- unique enforces 1-1 relation from membership to customer
   startDay NUMBER NOT NULL,
   startMonth NUMBER NOT NULL,
   startYear NUMBER NOT NULL,
   endDay NUMBER NOT NULL,
   endMonth NUMBER NOT NULL,
   endYear NUMBER NOT NULL,
   FOREIGN KEY (customerIDFK) REFERENCES customer(customerID)
);

CREATE TABLE payment(
   paymentID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY NOT NULL,
   membershipIDFK NUMBER NOT NULL, -- allow many-1 relation from payment to
membership
   subscriptionCharge NUMBER NOT NULL,
   billingAddress VARCHAR2(50) NOT NULL,
   cardHolderName VARCHAR2(50) NOT NULL,
   paymentDay NUMBER NOT NULL,
   paymentMonth NUMBER NOT NULL,
   paymentYear NUMBER NOT NULL,
   FOREIGN KEY (membershipIDFK) REFERENCES membership (membershipID)
);

CREATE TABLE movie(
   movieID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
   movieName VARCHAR2(50) NOT NULL,
   moviePoster VARCHAR2(255) NOT NULL, -- holds a link to poster
   movieDescription VARCHAR2(255) NOT NULL,
   movieRating NUMBER NOT NULL
);

-- junction table for m-m relation between membership and movie
CREATE TABLE membership_movie(
   membershipID NUMBER,
   movieID NUMBER,
   PRIMARY KEY(membershipID, movieID),
   FOREIGN KEY (membershipID) REFERENCES membership (membershipID),
   FOREIGN KEY (movieID) REFERENCES movie (movieID)
);

CREATE TABLE genre(
   genreID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
   genreName VARCHAR2(50) NOT NULL
);

-- junction table for m-m relation between movie and genre
CREATE TABLE movie_genre(
   genreID NUMBER,
   movieID NUMBER,
   PRIMARY KEY(genreID, movieID),
   FOREIGN KEY (genreID) REFERENCES genre (genreID),
   FOREIGN KEY (movieID) REFERENCES movie (movieID)
);

CREATE TABLE director(
   directorID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    directorFirstName VARCHAR2(30) NOT NULL,
    directorLastName VARCHAR2(30) NOT NULL
);

-- junction table for m-m relation between movie and director
-- has to be m-m for cases where a movie has multiple directors
CREATE TABLE movie_director(
    directorID NUMBER,
    movieID NUMBER,
    PRIMARY KEY(directorID, movieID),
    FOREIGN KEY (directorID) REFERENCES director (directorID),
    FOREIGN KEY (movieID) REFERENCES movie (movieID)
);
```

Demonstration of Simple queries executed on the updated database code above;



```
INSERT INTO account (username, password, email) VALUES
('j_bosks', 'scrypt:32768:8:1$hlpUppCFWiDRI1Qh$b4da11ed361115e07be78f20b43548cabea7011340134e4e0b59af5ada7d
    74e38e08668e0d5d4a37906d034b78dded88b1d765a76b14d90d66163a08c8c3b486', 'johnny@google.com');
INSERT INTO account (username, password, email) VALUES
('jane_mouse', 'scrypt:32768:8:1$3BOMyfldlVBNWcly$ba0b5eada9272ad45782c41b0cbf98608239298614234e7a20462ce7f
    0d14e207ab6e1f5c0082cf2ef37f87b1ec0daee8ea8957ea6b65a022dd96b219156862d', 'janey@yahoo.com');
INSERT INTO account (username, password, email) VALUES
('customer123', 'customerPass', 'example123@mail.com');
INSERT INTO account (username, password, email) VALUES
('admin456', 'adminPass', 'example456@mail.com');
INSERT INTO account (username, password, email) VALUES
('a_Lic3', 'ZspD#45Z', 'alice.c1@example.com');
INSERT INTO account (username, password, email) VALUES
('P_o_P','password12345','boblopez@example.com');
```

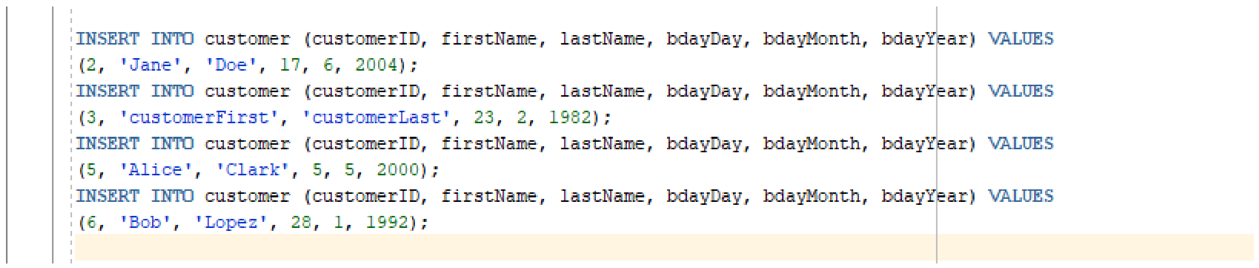
Relational Statements for Account Table



Columns | Data | Model | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Partitions | Indexes | SQL
Sort..
Filter:

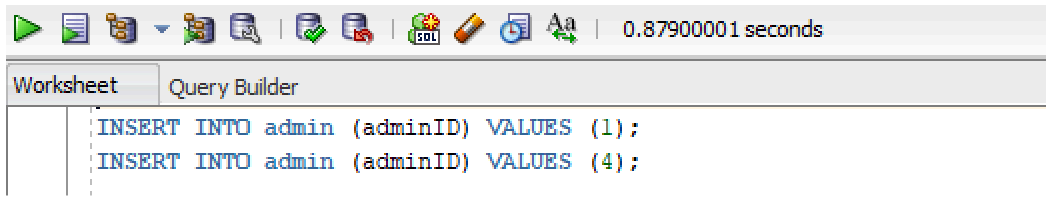
	ACCOUNTID	USERNAME	PASSWORD
1	1	j_bosks	scrypt:32768:8:1$hlpUppCFWiDRI1Qh$b4da11ed361115e07be78f2
2	2	jane_mouse	scrypt:32768:8:1$3BOMyfldlVBNWcly$ba0b5eada9272ad45782c41
3	3	customer123	customerPass
4	4	admin456	adminPass
5	5	a_Lic3	ZspD#45Z
6	6	P_o_P	password12345

Relational Statements for Customer Table



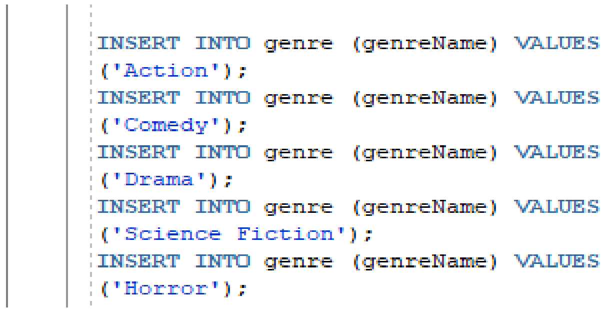
```
INSERT INTO customer (customerID, firstName, lastName, bdayDay, bdayMonth, bdayYear) VALUES
(2, 'Jane', 'Doe', 17, 6, 2004);
INSERT INTO customer (customerID, firstName, lastName, bdayDay, bdayMonth, bdayYear) VALUES
(3, 'customerFirst', 'customerLast', 23, 2, 1982);
INSERT INTO customer (customerID, firstName, lastName, bdayDay, bdayMonth, bdayYear) VALUES
(5, 'Alice', 'Clark', 5, 5, 2000);
INSERT INTO customer (customerID, firstName, lastName, bdayDay, bdayMonth, bdayYear) VALUES
(6, 'Bob', 'Lopez', 28, 1, 1992);
```

Relational Statements for Admin Table



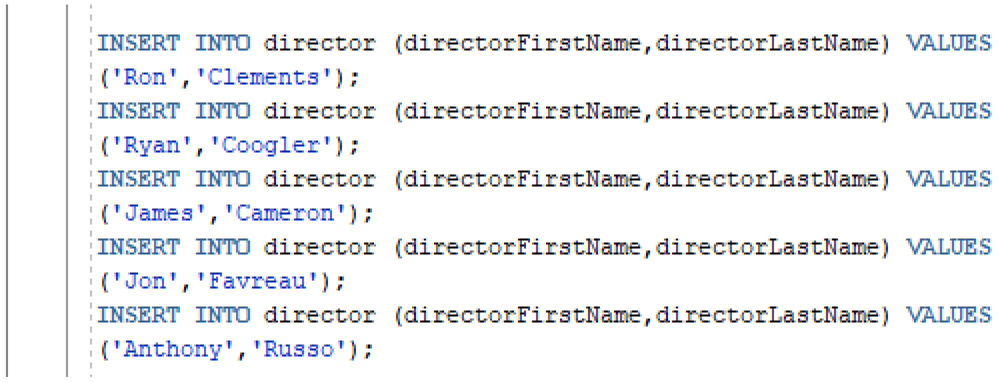
```
INSERT INTO admin (adminID) VALUES (1);
INSERT INTO admin (adminID) VALUES (4);
```

Relational Statements for Genre Table

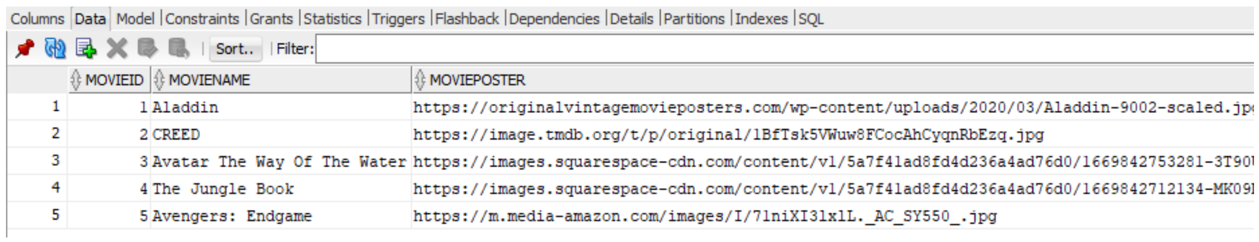


```
INSERT INTO genre (genreName) VALUES
('Action');
INSERT INTO genre (genreName) VALUES
('Comedy');
INSERT INTO genre (genreName) VALUES
('Drama');
INSERT INTO genre (genreName) VALUES
('Science Fiction');
INSERT INTO genre (genreName) VALUES
('Horror');
```

Relational Statements for Director Table



```
INSERT INTO director (directorFirstName,directorLastName) VALUES
('Ron','Clements');
INSERT INTO director (directorFirstName,directorLastName) VALUES
('Ryan','Coogler');
INSERT INTO director (directorFirstName,directorLastName) VALUES
('James','Cameron');
INSERT INTO director (directorFirstName,directorLastName) VALUES
('Jon','Favreau');
INSERT INTO director (directorFirstName,directorLastName) VALUES
('Anthony','Russo');
```

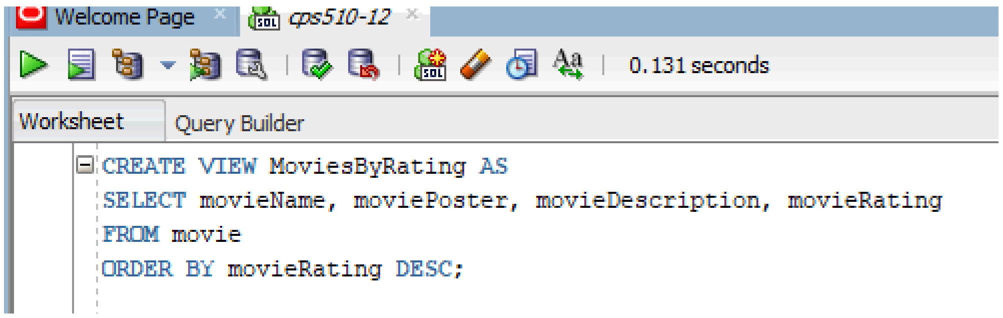

Columns | Data | Model | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Partitions | Indexes | SQL
Sort..
Filter:

	MOVIEID	MOVIENAME	MOVIEPOSTER
1	1	Aladdin	https://originalvintagemovieposters.com/wp-content/uploads/2020/03/Aladdin-9002-scaled.jp
2	2	CREED	https://image.tmdb.org/t/p/original/1BfTsk5VWuw8FCocAhCyqnRbEzq.jpg
3	3	Avatar The Way Of The Water	https://images.squarespace-cdn.com/content/v1/5a7f41ad8fd4d236a4ad76d0/1669842753281-3T90
4	4	The Jungle Book	https://images.squarespace-cdn.com/content/v1/5a7f41ad8fd4d236a4ad76d0/1669842712134-MK09
5	5	Avengers: Endgame	https://m.media-amazon.com/images/I/71niXI31x1L._AC_SY550_.jpg

ASSIGNMENT 4

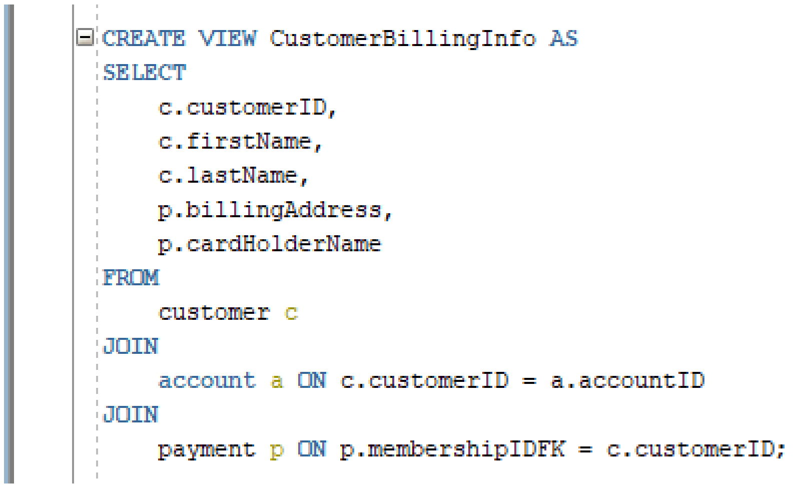
Demonstration of advanced queries executed on the updated database code above:

--create a view that displays movies with a rating



```sql
CREATE VIEW MoviesByRating AS
SELECT movieName, moviePoster, movieDescription, movieRating
FROM movie
ORDER BY movieRating DESC;
```

--create a view that displays customer info from customer table and payment info from payment table



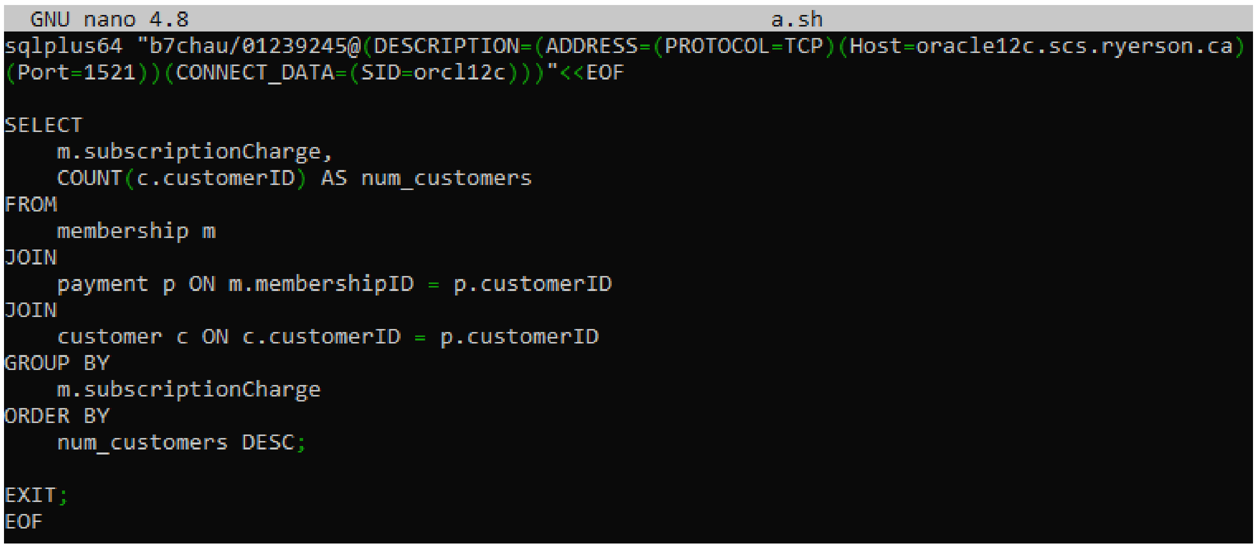
```sql
CREATE VIEW CustomerBillingInfo AS
SELECT
    c.customerID,
    c.firstName,
    c.lastName,
    p.billingAddress,
    p.cardHolderName
FROM
    customer c
JOIN
    account a ON c.customerID = a.accountID
JOIN
    payment p ON p.membershipIDFK = c.customerID;
```

ASSIGNMENT 5

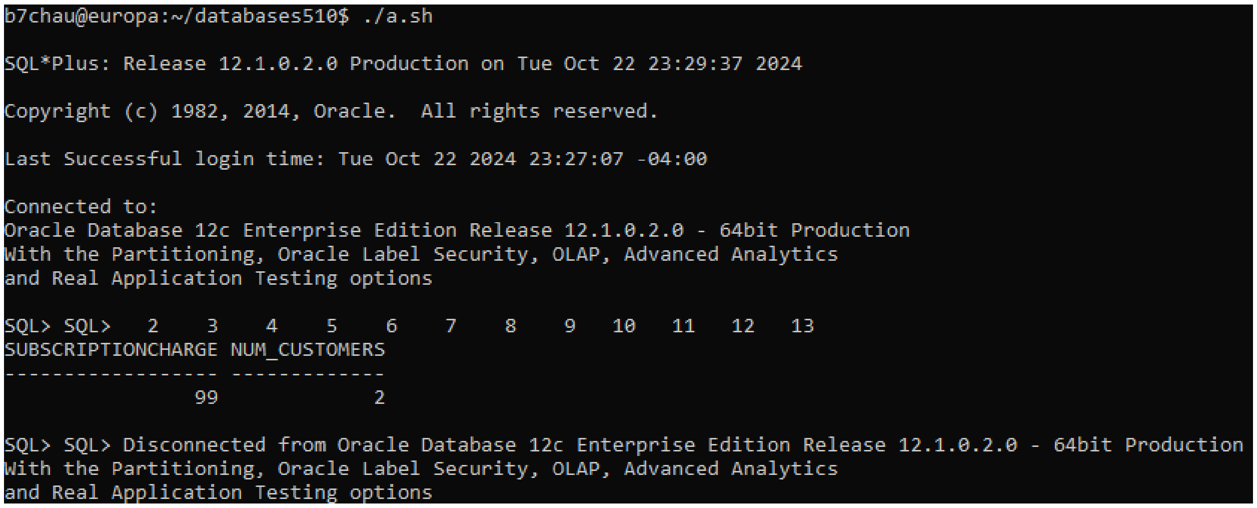
Advanced queries demonstrated using UNIX SHELL.

Shell Script 1:

Gather all customers with membership and group them by the price point of each membership



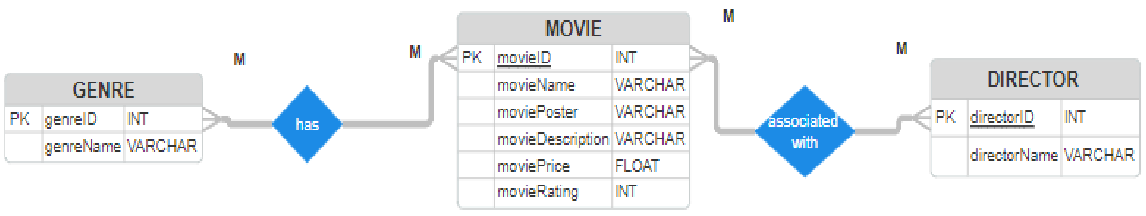
```
  GNU nano 4.8                                                a.sh
sqlplus64 "b7chau/01239245@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(Host=oracle12c.scs.ryerson.ca)
(Port=1521))(CONNECT_DATA=(SID=orcl12c)))"<<EOF

SELECT
    m.subscriptionCharge,
    COUNT(c.customerID) AS num_customers
FROM
    membership m
JOIN
    payment p ON m.membershipID = p.customerID
JOIN
    customer c ON c.customerID = p.customerID
GROUP BY
    m.subscriptionCharge
ORDER BY
    num_customers DESC;

EXIT;
EOF
```



```
b7chau@europa:~/databases510$ ./a.sh

SQL*Plus: Release 12.1.0.2.0 Production on Tue Oct 22 23:29:37 2024

Copyright (c) 1982, 2014, Oracle.  All rights reserved.

Last Successful login time: Tue Oct 22 2024 23:27:07 -04:00

Connected to:
Oracle Database 12c Enterprise Edition Release 12.1.0.2.0 - 64bit Production
With the Partitioning, Oracle Label Security, OLAP, Advanced Analytics
and Real Application Testing options

SQL> SQL>   2    3    4    5    6    7    8    9   10   11   12   13
SUBSCRIPTIONCHARGE NUM_CUSTOMERS
------------------ -------------
                99             2

SQL> SQL> Disconnected from Oracle Database 12c Enterprise Edition Release 12.1.0.2.0 - 64bit Production
With the Partitioning, Oracle Label Security, OLAP, Advanced Analytics
and Real Application Testing options
```

ASSIGNMENT 6

After the completion of our design for our DBMS system, we arrived at the step of normalizing our design. In order to normalize our design, we first outline the functional dependencies in the system. For example, our many-to-many relationship from movie to genre and movie to director.



movieID uniquely identifies each movie and determines all attributes. Each movie can be associated with multiple directors and genres. There the functional dependencies (FD) that would express this relationship would be:

movieID → movieName, moviePoster, movieDescription, moviePrice, movieRating.

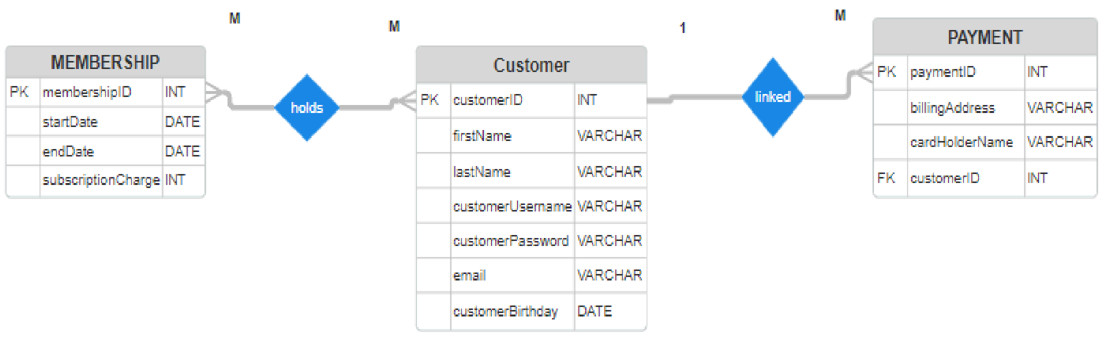
The director table FD is:

directorID → directorName

and is a Many-to-Many relationship with Movie because the directorID uniquely identifies each director and determines directorName. Each director can be linked to multiple movies. The Genres table FD is:

genreID → genreName

and is a Many-to-Many relationship with Movie the genreID uniquely identifies each genre, determining genreName. Each genre can be associated with multiple movies.



The customer's table FD is:

customerID → firstName, lastName, customerUsername, customerPassword, email, customerBirthday

customerUsername → customerID, firstName, lastName, customerPassword, email, customerBirthday email → customerID, firstName, lastName, customerUsername, customerPassword, customerBirthday.

The relationship is a One-to-Many relationship with Payment thus each customer can have multiple payment months. There is also a Many-to-Many relationship with Membership. The customerID, CustomerUsername, and email each uniquely identify a customer and determine all attributes. A customer can have multiple payments and memberships.

The FD to payment table is:

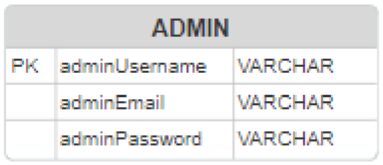
paymentID → billingAddress, cardHolderName,

customerID because payment ID uniquely identifies each payment, determining billingAddress, cardHolderName, and the linked customerID.

The FD to membership is:

membershipID → startDate, endDate, subscriptionCharge,

because membershipID uniquely identifies each membership, determining attributes like startDate, endDate, and subscriptionCharge. A membership can be associated with multiple customers.



ADMIN		
PK	adminUsername	VARCHAR
	adminEmail	VARCHAR
	adminPassword	VARCHAR

The admin table FD is:

adminUsername → adminEmail, adminPassword

adminEmail → adminUsername, adminPassword

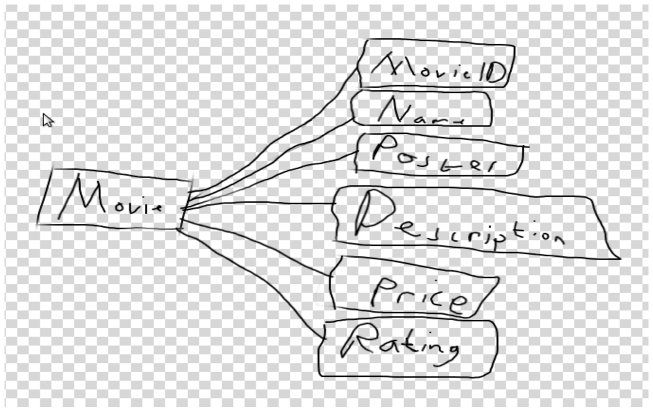
The relationship is independent from other table with mean there is no direct relationship with other tables. The adminUsername and adminEmail uniquely identify each admin, determining the other attributes.

ASSIGNMENT 7

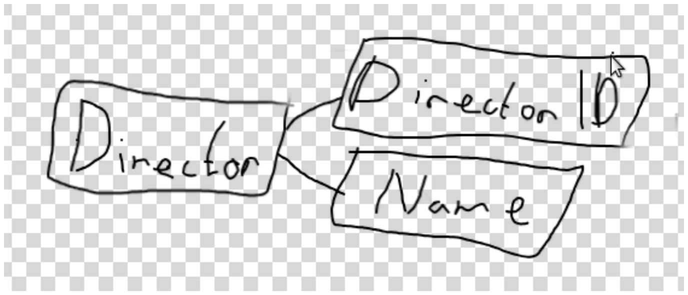
To ensure each table is in the Third Normal Form (3NF), we want to verify that:

1. All the tables are in the Second Normal Form(2NF): This means they have no partial dependencies (a non-key attribute depending only on part of a composite key).
2. No transitive dependencies: In 3NF, non-key attributes must not depend on other non-key attributes

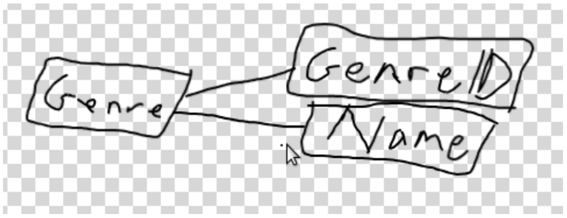
1. Movie Table
   - Functional Dependencies (FDs): movieID → movieName, moviePoster, movieDescription, moviePrice, movieRating
   - Verification: Since movieID is the primary key and determines all other attributes, this table has no partial or transitive dependencies, meaning it should already be in 3NF.



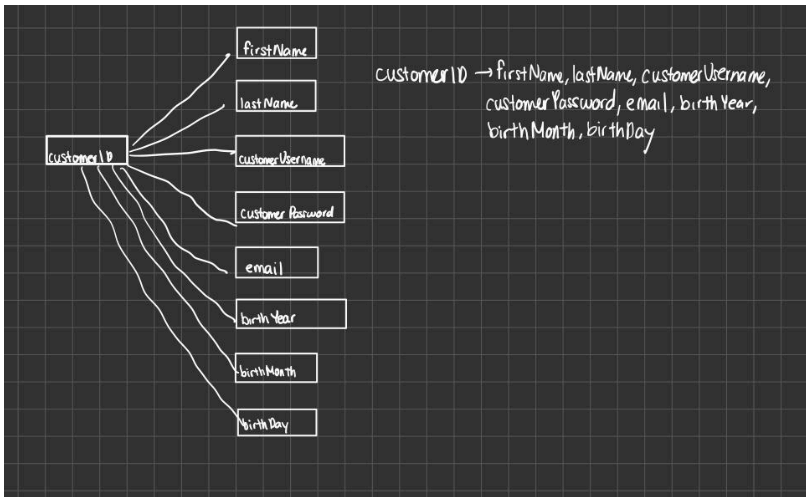
2. Director Table
   - FDs: directorID → directorName
   - Verification: directorID uniquely identifies each director without any non-key attributes depending on other non-key attributes. Therefore it is in 3NF.



3. Genre Table
   - FDs: genreID → genreName
   - Verification: genreID uniquely identifies each genre, and there are no transitive dependencies. Therefore, it is in 3NF.

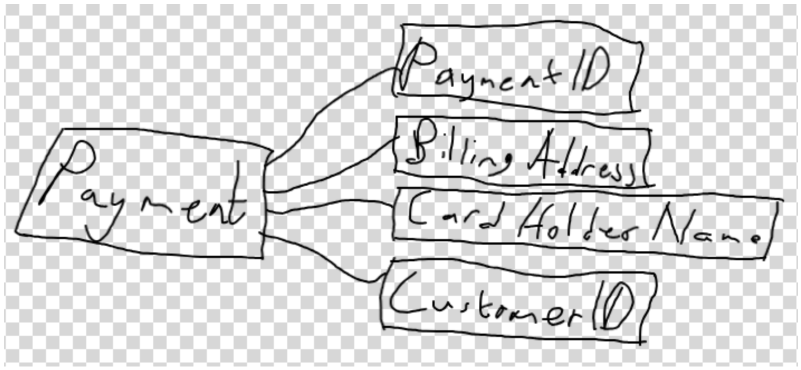


4. Customer Table


customerID
firstName
lastName
customerUsername
customerPassword
email
birthYear
birthMonth
birthDay
customerID → firstName, lastName, customerUsername, customerPassword, email, birthYear, birthMonth, birthDay

To turn Customer into 3NF, we changed the Birthday attribute. Initially we had just customerBirthday however we split it into a separate field for customerYear, customerMonth, customerDay making it more divisible for 1NF. Thus the Customer table is now 3NF.

5. Payment Table



- FDs: paymentID → billingAddress, cardHolderName, customerID
- To turn payment into 3NF, we changed the Address attribute. Initially this contained the street number and street making it divisible (breaching 1NF). So we split up the address into street number and street name
- paymentID → billingStreetNumber, billingStreetName, cardHolderName, customerID

ASSIGNMENT 8

Bernstein's Algorithm Steps:

1. Determine all functional dependencies
    a. Find and remove redundancies
    b. Find and remove partial dependencies
2. Find keys
3. Create tables

STEP 1 (find all functional dependencies):

Movie Table FDs:

- movieID → {movieName, moviePoster, movieDescription, moviePrice, movieRating, movieDirector, movieGenre}

Admin Table FDs:

- adminUsername → {adminEmail, adminPassword}
- adminEmail → {adminUsername, adminPassword} (ASSUMPTION: each email is unique)

Customer Table FDs:

- customerID → {firstName, lastName, customerUsername, customerPassword, email, customerBirthday}
- customerUsername → {customerID, firstName, lastName, customerPassword, email, customerBirthday}
- email → {customerID, firstName, lastName, customerUsername, customerPassword, customerBirthday} (ASSUMPTION: each email is unique)

Payment Table FDs:

- paymentID → {billingAddress, cardHolderName, customerID}
- customerID → paymentID (ASSUMPTION: each customer can make only one payment, or payment records per customer are unique)

Membership Table FDs:

- membershipID → {startDate, endDate, subscriptionCharge, customerID}
- customerID → membershipID (ASSUMPTION: one active membership per customer)

STEP a&b (Decompose by finding and removing redundancies and partial dependencies)

Decomposing the Movie Table

- Functional Dependencies:
    - movieID → movieName, moviePoster, movieDescription, moviePrice, movieRating, movieDirector, movieGenre
- Decomposition:
    - Since movieDirector and movieGenre can introduce redundancy (multiple movies can have the same director or genre), create separate tables:
        - Movie: movieID, movieName, moviePoster, movieDescription, moviePrice, movieRating
        - Director: directorID, directorName
        - Genre: genreID, genreName

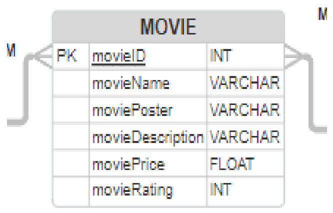
Decomposing the Membership Table

- Functional Dependencies:
  - membershipID → {startDate, endDate, subscriptionCharge}
  - customerID → membershipID (ASSUMPTION: each customer can have only one active membership)
- Decomposition:
  - Since customerID determines membershipID, we need to separate customerID from the main membership attributes.

STEP 2&3 (Find Keys and Create Tables):

- Candidate Key:
  - movieID

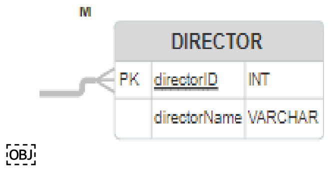
movieID → movieName, moviePoster, movieDescription, moviePrice, movieRating.



MOVIE		
PK	movieID	INT
	movieName	VARCHAR
	moviePoster	VARCHAR
	movieDescription	VARCHAR
	moviePrice	FLOAT
	movieRating	INT

- Candidate Key:
  - directorID

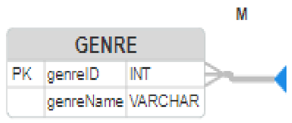
directorID → directorName



DIRECTOR		
PK	directorID	INT
	directorName	VARCHAR

- Candidate Key:
  - genreID

genreID → genreName



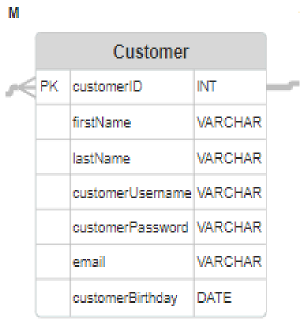
GENRE		
PK	genreID	INT
	genreName	VARCHAR

Candidate Keys:

- customerID
- customerUsername
- email

customerID → firstName, lastName, customerUsername, customerPassword, email, customerBirthday

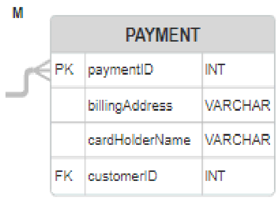
customerUsername → customerID, firstName, lastName, customerPassword, email, customerBirthday email → customerID, firstName, lastName, customerUsername, customerPassword, customerBirthday.



Customer		
PK	customerID	INT
	firstName	VARCHAR
	lastName	VARCHAR
	customerUsername	VARCHAR
	customerPassword	VARCHAR
	email	VARCHAR
	customerBirthday	DATE

- Candidate Key:
    - paymentID

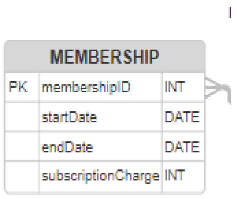
￼ paymentID → billingAddress, cardHolderName,



PAYMENT		
PK	paymentID	INT
	billingAddress	VARCHAR
	cardHolderName	VARCHAR
FK	customerID	INT

- Candidate Key:
    - membershipID

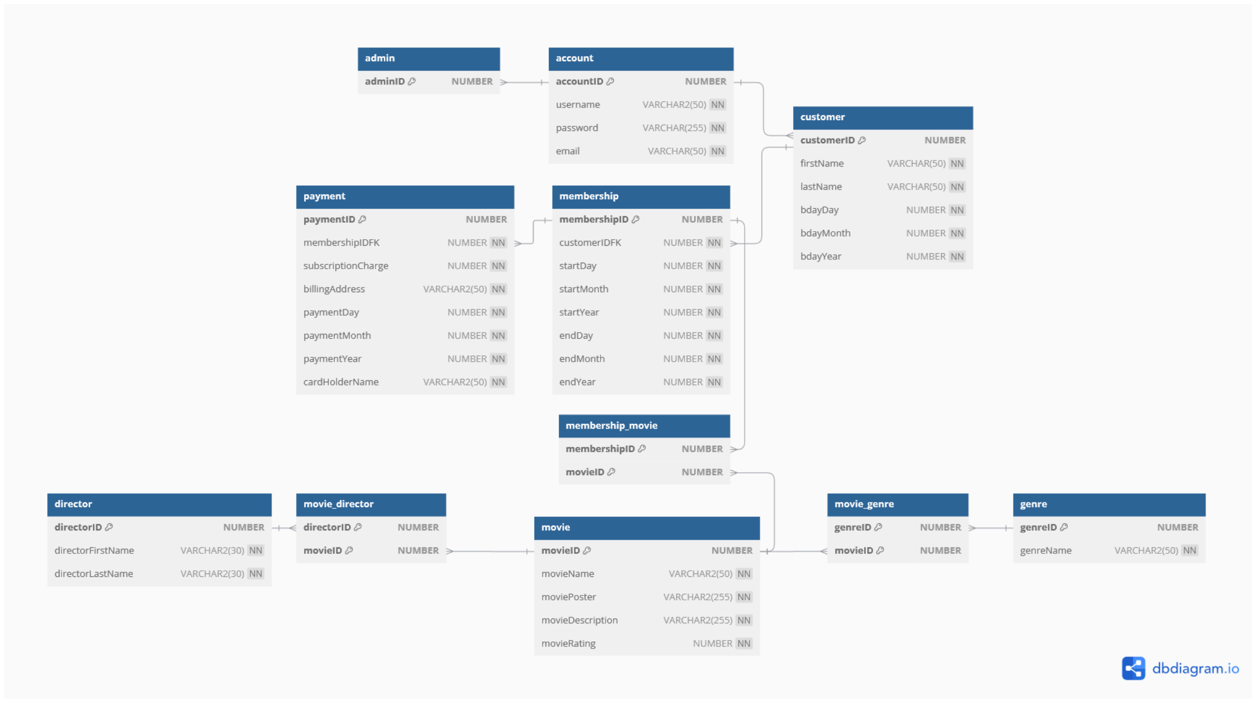
membershipID → startDate, endDate, subscriptionCharge,



MEMBERSHIP		
PK	membershipID	INT
	startDate	DATE
	endDate	DATE
	subscriptionCharge	INT

ASSIGNMENT 9

Below is the Normalized schema of tables and content used for our database



Demonstration of Relational Algebra queries executed on the updated database code above;

1. Find movies directed by a specific director (e.g., "John Doe").
   director ⋈ director.directorID = movie_director.directorID ⋈ movie
   σ directorFirstName = 'John' ∧ directorLastName = 'Doe'
2. Get the genres for a specific movie (e.g., "Inception").
   σ movieName = 'Inception' (movie) ⋈ movie_genre ⋈ genre
3. List all payments made by customers born in 1990.
   σ bdayYear = 1990 (customer) ⋈ payment

Natural Join (⋈)
Selection (σ)

ASSIGNMENT 10

Summary:

Over 10 weeks, a fully functional database and a graphical user interface (GUI) were designed and implemented to support our online movie store. This project aimed to integrate a user-friendly frontend with a robust backend to manage accounts, movies, memberships, and payments efficiently. The project was divided into 10 assignments, each building upon the previous to gradually develop the system.

Key Objectives Included:
Database Development: Design and implement a relational database with normalized tables and optimized queries.
GUI Implementation: Develop a functional and user-friendly interface for end users to interact with the database.
Data Integrity and Scalability: Ensure accuracy, consistency, and scalability through constraints, indexing, and structured design.

Challenges and Solutions:
Challenge: Establishing a secure and efficient connection between the GUI and backend.
- Solution: Used prepared statements and parameterized queries to prevent SQL injection.

Challenge: Normalizing the database while preserving functionality.
- Solution: The Bernstein algorithm was used to decompose tables methodically into BCNF while maintaining dependency preservation and lossless joins. Balancing between eliminating redundancy and retaining ease of data retrieval was complex. Many-to-many relationships and composite keys were added to the database.

Challenge: Managing data dependencies during integration.
- Solution: Ensured proper sequence of table creation and database initialization scripts.

Conclusion:
The project successfully delivered a functional online movie store system. The GUI allows users to browse movies, manage memberships, and make payments, all while interacting seamlessly with the backend. The database is optimized for scalability and maintains data integrity.