Databases

Final Project Report

Course: Databases

Group ID: 11

Name: Robert van Timmeren

Student ID: S4535553

Date: 30-10-2020

Table of Contents

1. User requirements

Netflix is going to open its first physical location, a so-called pop-up store. On this new location they are also planning to sell films and series and keep track of their customer subscriptions. The problem is, they do not have a database yet for this new concept. As such, they wish to create a new database to keep track of their sales in this new store. To correctly do this, they wish to keep track what they are selling, who they are selling to, and what has been sold.

All general information of customers should be tracked, such as their name, address, date of birth and email. The subscription that a customer owns should include their plan, the timeframe and when their subscriptions starts and ends. Customers can buy multiple series and films, but a subscription is something only one person can own! The films that are sold by Netflix have a list of information, ranging from their title and genre to their price and release date. The ratings of films are not needed, as Netflix is currently negotiating with IMDB on this. However, it would be interesting to know what director directed which movie. Series are also sold as a whole, and information on them should also be tracked, such as the title, genre, rating and of course the price Netflix will sell them for. Finally, Netflix wishes to give customers suggestions in the future on what actors are playing in which movies. Therefore, also keep track of the actors that play in series and films. Include their names, date of birth and what agency they work for.

1.1. Entity & attributes

With the initial narrative description of the problem and information needed, a list of entities and attributes is created (see Table 1: 'Entity and attributes').

Entity	**Attributes**
Subscription	SubscriptionID, Type/Plan, Timeframe (1mnd, 1yr, 2yr), Start_date, End_date
Film	FilmID, Title, Genre, Director (first_name & last_name), Length, Release_date, Price
Series	SerieID, Title, Genre, Rating, Price
Customer	CustomerID, First_name, Last_name, Address (house_number, postal_code, street, city), Date_of_birth, E_mail
Sales	SalesID, Name, Quantity, Total_price, Date
Actor	ActorID, First_name, Last_name, Age, Agency

Table 1: 'Entity and attributes'

2. E-R diagram

With the list of entities and attributes an E-R diagram was created. In Figure 1: 'E-R Diagram' the final version of the diagram can be found. The diagram offers a good visualization of the conceptual data model. In chapter 3, this diagram will be converted into relational schemas to take the next step in the design of a database.

Before the creation of the E-R diagram, some extensions and restrictions were noted:

1. Normally, an email would be a multi-valued attribute, since a person can have multiple emails. However, since a Netflix account is registered to a specific email, the email a single-valued attribute.
2. Since the end date of a subscription is dependent on the subscription plan, this is a derived attribute.

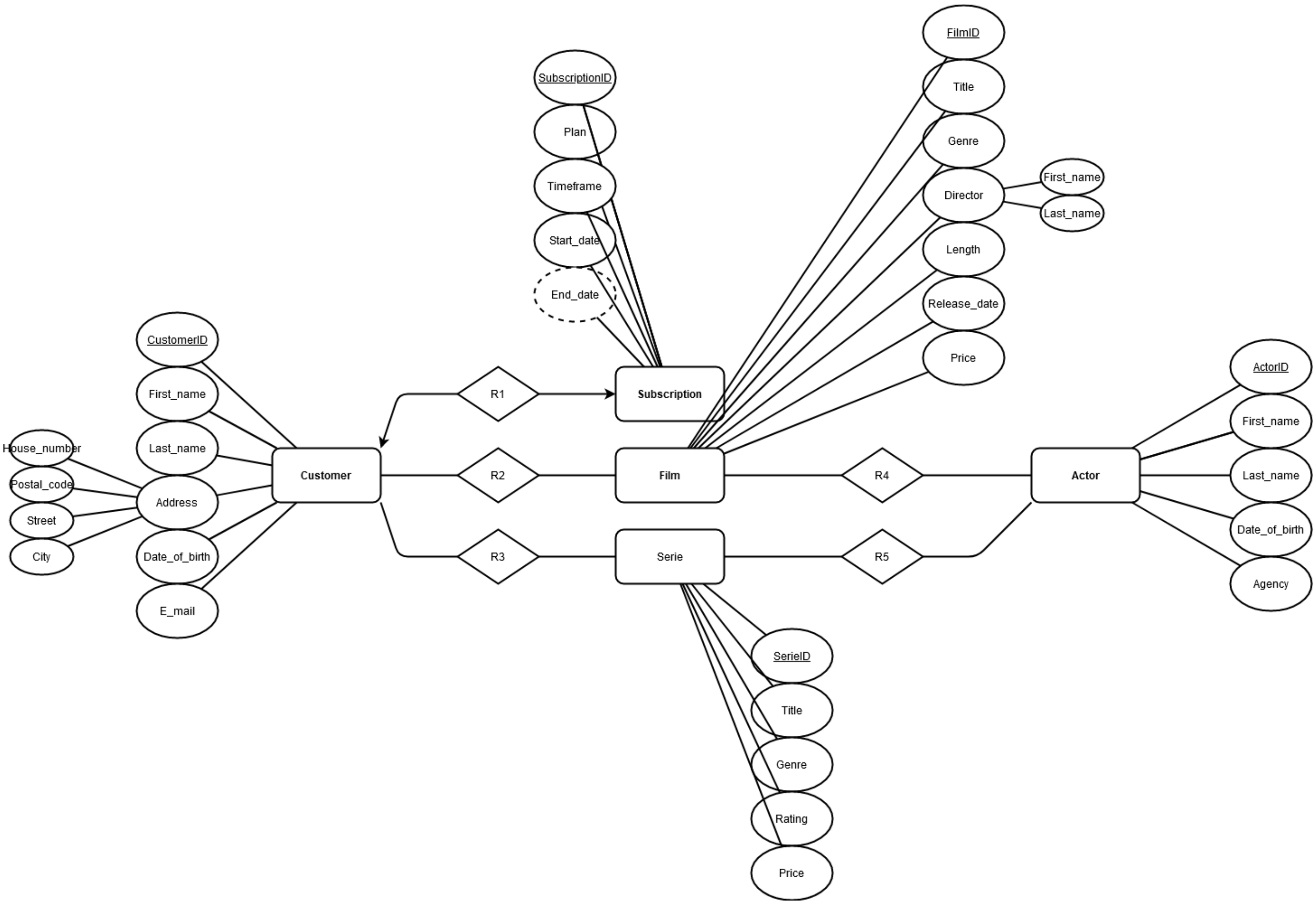

Figure 1:'E-R Diagram'

3. Relational schemas

The E-R diagram of chapter 2 is now converted into relational schemas. In these schemas we specify the names of the tables, attributes as they will appear in the database and the primary key for the table. This is done by converting all entities, attributes and relationships.

CUSTOMER(<u>customerID (pk)</u>, first_name, last_name, postal_code, street, house_number, city, date_of_birth, e_mail)

SUBSCRIPTION(<u>subscriptionID (pk)</u>, <u>customerID (fk)</u>, plan, timeframe, start_date)

Foreign key customerID references CUSTOMER(customerID)

FILM(<u>filmID (pk)</u>, title, genre, director_firstname, director_lastname, length, release_date, price)

SERIE(<u>serieID (pk)</u>, title, genre, rating, price)

ACTOR(<u>actorID (pk)</u>, first_name, last_name, date_of_birth, agency)

R2(<u>customerID (fk)</u>, <u>filmID (fk)</u>)

Foreign key customerID references CUSTOMER(customerID)

Foreign key filmID references FILM(filmID)

R3(<u>customerID (fk)</u>, <u>serieID (fk)</u>)

Foreign key customerID references CUSTOMER(customerID)

Foreign key serieID references SERIE(serieID)

R4(<u>filmID (fk)</u>, <u>actorID (fk)</u>)

Foreign key filmID references FILM(filmID)

Foreign key actorID references ACTOR(actorID)

R5(<u>serieID (fk)</u>, <u>actorID (fk)</u>)

Foreign key serieID references SERIE(serieID)

Foreign key actorID references ACTOR(actorID)

With this relational schema the process of normalization will begin.

4. Normalization

The relational schemas of the previous chapter are now normalized to ensure that only related data is stored in each table, and that redundancy is minimized. To normalize the relation schemas the functional dependencies will first have to be discovered.

4.1. Functional dependencies

In order to normalize all relational schemas, the functional dependencies must be determined. We differentiate 'hard' and 'soft' functional dependencies. Meaning that hard functional dependencies can be said without a doubt, and soft functional dependencies have a possibility of not holding (especially with large data sets).

CUSTOMER (<u>customerID</u>, first_name, last_name, postal_code, street, house_number, city, date_of_birth, e_mail)

customerID → {first_name, last_name, postal_code, street, house_number, city, date_of_birth, e_mail}

postal_code → {street, city} (W*e assumed that a postal code never has more than one street on it*.)

~~e_mail → customerID~~ (*Emails can be re-registered and/or fake emails can be given. As such, this functional dependency does not hold.*)

SUBSCRIPTION (<u>subscriptionID</u>, <u>customerID</u>, plan, timeframe, start_date)

subscriptionID → {customerID, plan, timeframe, start_date}

plan → timeframe

FILM (<u>filmID</u>, title, genre, director_firstname, director_lastname, length, release_date, price)

filmID → {title, genre, director_firstname, director_lastname, length, release_date, price}

SERIE (<u>serieID</u>, title, genre, rating, price)

serieID → {title, genre, rating, price}

ACTOR (<u>actorID</u>, first_name, last_name, date_of_birth, agency)

actorID → {first_name, last_name, date_of_birth, agency}

R2(<u>customerID</u>, <u>filmID</u>)

customerID → filmID

FilmID → customerID

R3(<u>customerID</u>, <u>serieID</u>)

customerID → serieID

serieID → customerID

R4(<u>filmID</u>, <u>actorID</u>)

filmID → actorID

actorID → filmID

R5(<u>serieID</u>, <u>actorID</u>)

serieID → actorID

actorID → serieID

4.2. Soft functional dependencies

The following functional dependencies are functional dependencies with a high probability, but not with certainty.

FILM(<u>filmID</u>, title, genre, director_firstname, director_lastname, length, release_date, price)

{title, director_firstname, director_lastname} → {filmID, genre, length, release_date, price, film}

{director_firstname, director_lastname, release_date} → {filmID, genre, length, price, film}

SERIE(<u>serieID</u>, title, genre, rating, price)

{title, genre, rating} → {serieID, price}

{title, genre, price} → {serieID, rating}

ACTOR(<u>actorID</u>, first_name, last_name, date_of_birth, agency)

actorID → {first_name, last_name, date_of_birth, agency}

{first_name, last_name, date_of_birth} → {actorID, agency}

{first_name, last_name, agency} → {actorID, date_of_birth}

4.3. Normal forms

In this sub-paragraph we will analyze the current normal form of all the schemas. We will first define how all normal forms are met:

- 1NF: requires that all tables entries are atomic, so no groups of elements or repeating elements.
- 2NF: is in 1NF and does not have any non-prime (an attribute that is not part of any candidate key) attribute that is functionally dependent on any proper subset of any candidate key.
- 3NF: is in 2NF and requires that there are no dependencies on non-prime attributes (no transitive dependencies). Attributes that are not in the primary key cannot have independent functional dependencies.
- BCNF: is in 3NF and requires that every non-trivial functional dependency, the left-hand side is a superkey.

The tables that were not in 3NF/BCNF have been normalized and an in-depth description is given. The schemas that were easier to normalize and required no decomposition are still described, but the description is less in-depth.

CUSTOMER (<u>customerID</u>, first_name, last_name, postal_code, street, house_number, city, date_of_birth, e_mail)

- 1NF: ✓ Subscription meets the definitions of a relation.
- 2NF: ✓ There are no partial key dependencies.
- 3NF: ✕ A transitive dependency exists, namely {customerID} → {postal_code}and {postal_code} → {city, street}. The functional dependency {postal_code} → {city, street} violates 3NF, as the left-hand side is not a superkey and right-hand side is a set of non-key attributes. In-depth explanation below.

Given:

customerID → {first_name, last_name, postal_code, street, house_number, city, date_of_birth, e_mail}

postal_code → {street, city}

Closures:

customerID+ → {customerID, first_name, last_name, postal_code, street, house_number, city, date_of_birth, e_mail}

postal_code+ → {postal_code, street, city}

Prime attributes: customerID

Non-prime attributes: first_name, last_name, postal_code, street, house_number, city, date_of_birth, e_mail

Canonical cover:

Regular form:

customerID → first_name

customerID → last_name

customerID → postal_code

customerID → street

customerID → house_number

customerID → city

customerID → date_of_birth

customerID → e_mail

postal_code → street

postal_code → city

Closures of left-hand side:

CustomerID+ = {customerID, first_name, last_name, postal_code, street, house_number, city, date_of_birth, e_mail}

Postal_code+ = {postal_code, street, city}

Redundant functional dependencies:

customerID → street

customerID → city

Because, postal_code → {street, city} and customerID → postal_code.

Canonical cover:

customerID → {first_name, last_name, postal_code, house_number, date_of_birth, e_mail}

postal_code → {street, city}

Superkeys check

customerID → {first_name, last_name, postal_code, street, house_number, city, date_of_birth, e_mail} ✓

postal_code → {street, city} ✕

Since postal_code is not a superkey, the table is not in 3NF.

Decomposing

The solution to get the customer table in 3NF is to split customer into two new tables: CustomerData and PostalCodes:

- CustomerData (<u>customerID</u> (pk), first_name, last_name, <u>postal_code</u> (fk), house_number, date_of_birth, e_mail)
- PostalCodes (<u>postal_code</u> (pk), street, city)

customerID → {first_name, last_name, postal_code, city, date_of_birth, e_mail}

postal_code → {street, city}

We will now recheck both tables on all normal forms:

CustomerData

- 1NF: ✓ Customer meets the definition of a relation.
- 2NF: ✓ There are no partial key dependencies.
- 3NF: ✓ There are no transitive dependencies.
- BCNF: ✓ All determinants are candidate keys.

PostalCodes

- 1NF: ✓ PostalCodes meets the definition of a relation.
- 2NF: ✓ There are no partial key dependencies.
- 3NF: ✓ There are no transitive dependencies.
- BCNF: ✓ All determinants are candidate keys.

SUBSCRIPTION (<u>subscriptionID</u>, <u>customerID</u>, plan, timeframe, start_date)

- 1NF: ✓ Subscription meets the definitions of a relation.
- 2NF: ✓ There are no partial key dependencies.
- 3NF: ✕ A transitive dependency exists, namely subscriptionID → plan and plan → timeframe. The functional dependency plan → timeframe violates 3NF, as the left-hand side is not a superkey and right-hand side is a set of non-key attributes.

Given:

subscriptionID → {customerID, plan, timeframe, start_date}

plan → timeframe

Closures:

subscriptionID+ → {customerID, plan, timeframe, start_date}

plan+ → timeframe

Prime attributes: subscriptionID

Non-prime attributes: customerID, plan, timeframe, start_date

Canonical cover:

Regular form:

subscriptionID → customerID

subscriptionID → plan

subscriptionID → timeframe

subscriptionID → start_date

plan → timeframe

Closures of left-hand side:

subscriptionID+ = subscriptionID, customerID, plan, timeframe, start_date

plan+ = plan, timeframe

Redundant functional dependencies:

subscriptionID → timeframe

Because, subscriptionID → plan and plan → timeframe.

Canonical cover:

subscriptionID → {customerID, plan, start_date}

plan → timeframe

Superkeys check

subscriptionID → {customerID, plan, start_date, timeframe} ✓

plan → timeframe ✕

Since plan is not a superkey, the table is not in 3NF.

Decomposing

The solution to get the subscription table in 3NF is to split subscription into two new tables: SubscriptionData and Plans:

- SubscriptionData (<u>subscriptionID</u> (pk), <u>customerID</u> (fk), <u>planID</u> (fk), start_date)
- Plans (<u>plan</u> (pk), timeframe)

subscriptionID → {customerID, planID, start_date}

plan → {timeframe}

We will now recheck both tables on all normal forms:

SubscriptionData

- 1NF: ✓ SubscriptionData meets the definition of a relation.
- 2NF: ✓ There are no partial key dependencies.
- 3NF: ✓ There are no transitive dependencies.
- BCNF: ✓ All determinants are candidate keys.

Plans

- 1NF: ✓ Plans meets the definition of a relation.
- 2NF: ✓ There are no partial key dependencies.
- 3NF: ✓ There are no transitive dependencies.
- BCNF: ✓ All determinants are candidate keys.

FILM

- 1NF: ✓ Film meets the definitions of a relation.
- 2NF: ✓ There are no partial key dependencies.
- 3NF: ✓ There are no transitive dependencies.
- BCNF: ✓ All determinants are candidate keys.

SERIE

- 1NF: ✓ Serie meets the definitions of a relation.
- 2NF: ✓ There are no partial key dependencies.
- 3NF: ✓ There are no transitive dependencies.
- BCNF: ✓ All determinants are candidate keys.

ACTOR

- 1NF: ✓ Actor meets the definitions of a relation.
- 2NF: ✓ There are no partial key dependencies.
- 3NF: ✓ There are no transitive dependencies.
- BCNF: ✓ All determinants are candidate keys.

R2

- 1NF: ✓ R2 meets the definitions of a relation.
- 2NF: ✓ There are no partial key dependencies.
- 3NF: ✓ There are no transitive dependencies.
- BCNF: ✓ All determinants are candidate keys.

R3

- 1NF: ✓ R3 meets the definitions of a relation.
- 2NF: ✓ There are no partial key dependencies.
- 3NF: ✓ There are no transitive dependencies.
- BCNF: ✓ All determinants are candidate keys.

R4

- 1NF: ✓ R4 meets the definitions of a relation.
- 2NF: ✓ There are no partial key dependencies.
- 3NF: ✓ There are no transitive dependencies.
- BCNF: ✓ All determinants are candidate keys.

R5

- 1NF: ✓ R5 meets the definitions of a relation.
- 2NF: ✓ There are no partial key dependencies.
- 3NF: ✓ There are no transitive dependencies.
- BCNF: ✓ All determinants are candidate keys.

5. SQL

The SQL section can be divided between a DDL and a DML section. The DDL section goes over the creation of the database and filling it with data. The DML section goes over querying the database to retrieve relevant information.

5.1. DDL

All the SQL DLL queries can be found in Appendix I as well as the digital enclosure.

5.2. DML

Multiple SQL queries were created to query the database. The queries can be found below, with a description of what they retrieve in the query.

Query 1

```
-- Selects all distinct actors who played in movies with crime genre.
SELECT DISTINCT actor.first_name, actor.last_name
FROM actor
NATURAL JOIN r4
WHERE filmID IN(
    SELECT filmID
    FROM film
    NATURAL JOIN r4
    NATURAL JOIN actor
    WHERE genre='crime'
);
```

Query 2

```
-- Selects all actors who play in more than one movie.
SELECT first_name, last_name, COUNT(first_name) AS Movie_count
FROM actor
NATURAL JOIN r4
GROUP BY actorID
HAVING COUNT(actorID) > 1;
```

Query 3

```
-- Selects all distinct actors who played in both a serie and film.
SELECT DISTINCT first_name, last_name
FROM actor
NATURAL JOIN r4
WHERE filmID IN (
        SELECT filmID
        FROM film
        NATURAL JOIN r4
        NATURAL JOIN r5
        WHERE serieID IN(
                SELECT serieID
                FROM serie
                NATURAL JOIN r5
        )
);
```

Query 4

```
-- Select the average ratings of all genres
SELECT AVG(rating), genre
FROM serie
GROUP BY genre;
```

Query 5

```
-- Select persons and their total amount of money spent on movies
SELECT first_name, last_name, sum(price) AS Paid_on_movies
FROM film
NATURAL JOIN r2
NATURAL JOIN customerdata
GROUP BY first_name;
```

Query 6

```
-- Select actors who played in the movie The Shining
SELECT first_name, last_name
FROM actor
NATURAL JOIN film
NATURAL JOIN r4
WHERE title = 'the Shining';
```

Query 7

```
-- Select distinct plans that have a timeframe of longer than a month
SELECT DISTINCT p.plan
FROM CustomerData cd
JOIN SubscriptionData sd
ON cd.customerID = sd.customerID
JOIN Plans p
ON sd.plan = p.plan
WHERE timeframe > 31;
```

Query 8

```
-- Select all persons that live in Groningen and who's streets start with
an H
SELECT c.first_name, c.last_name
FROM Customerdata c
WHERE EXISTS(
    SELECT *
    FROM PostalCodes po
    WHERE po.city = 'Groningen'
    AND po.street LIKE 'H%'
    AND po.postal_code = c.postal_code
);
```

Query 9

```
-- Select customers who have a plan with the highest timeframe possible
SELECT c.first_name, c.last_name
FROM customerdata c
WHERE customerid IN (
    SELECT customerid
    FROM subscriptiondata s
    WHERE plan IN(
        SELECT plan
        FROM plans p
        WHERE timeframe = (SELECT MAX(timeframe) FROM plans)
    )
);
```

Query 10

```
-- Select the series and films with the highest price
(SELECT title, price
FROM serie
WHERE price = (SELECT MAX(price) from serie))
UNION
(SELECT title, price
FROM film
WHERE price = (SELECT MAX(price) from film));
```

Appendix

I. SQL DDL – Database creation

```
-- Creation of CustomerData table.
CREATE TABLE CustomerData (
    customerID      INT NOT NULL AUTO_INCREMENT,
    first_name      VARCHAR(40),
    last_name       VARCHAR(40),
    postal_code     varchar(12) NOT NULL,
    house_number    VARCHAR(5), #there is no suffix column, so we also allow
characters
    date_of_birth   DATE,
    e_mail          VARCHAR(255),
    PRIMARY KEY (customerID)
);


-- Creation of PostalCodes table.
CREATE TABLE PostalCodes (
    postal_code     VARCHAR(12) NOT NULL,
    city            VARCHAR(40),
    street          VARCHAR(40),
    PRIMARY KEY (postal_code)
);


-- Creating the foreign key postal_code in CustomerData.
ALTER TABLE CustomerData
ADD FOREIGN KEY (postal_code) REFERENCES PostalCodes(postal_code);


-- Creation of SubscriptionData table and adding the foreign key reference
to customerID.
CREATE TABLE SubscriptionData (
    subscriptionID INT NOT NULL AUTO_INCREMENT,
    customerID      INT NOT NULL,
    plan            VARCHAR(40),
    start_date      DATE,
    PRIMARY KEY (subscriptionID),
    FOREIGN KEY (customerID) REFERENCES CustomerData(customerID),
    CONSTRAINT uq_customerID UNIQUE (customerID)
);


-- Creation of SubscriptionData table.
CREATE TABLE Plans (
    plan            VARCHAR(40) NOT NULL,
    timeframe       INT NOT NULL,
    PRIMARY KEY (plan)
);


-- Creating the foreign key reference of Plan.
ALTER TABLE SubscriptionData
ADD FOREIGN KEY (plan) REFERENCES Plans(plan);
```

```sql
-- Creating the table Film
CREATE TABLE Film (
    filmID                  INT NOT NULL AUTO_INCREMENT,
    title                   VARCHAR(40),
    genre                   VARCHAR(40),
    director_firstname      VARCHAR(40),
    director_lastname       VARCHAR(40),
    length                  TIME,
    release_date            DATE,
    price                   DECIMAL(13,2),
    PRIMARY KEY (filmID)
);

-- Creating the table Serie
CREATE TABLE Serie (
    serieID  INT NOT NULL AUTO_INCREMENT,
    title    VARCHAR(40),
    genre    VARCHAR(40),
    rating   DECIMAL(3,1) CHECK(rating BETWEEN 0 and 10.0),
    price    DECIMAL(13,2),
    PRIMARY KEY (serieID)
);

-- Creating the table Actor
CREATE TABLE Actor (
    actorID          INT NOT NULL AUTO_INCREMENT,
    first_name       VARCHAR(40),
    last_name        VARCHAR(40),
    date_of_birth    DATE,
    agency           VARCHAR(40),
    PRIMARY KEY (actorID)
);

-- Creating table R2
CREATE TABLE R2 (
    customerID  INT NOT NULL,
    filmID      INT NOT NULL,
    FOREIGN KEY (customerID) REFERENCES CustomerData(customerID),
    FOREIGN KEY (filmID) REFERENCES Film(filmID),
    CONSTRAINT uq_fk_customeridfilmid UNIQUE (customerID, filmID)
);

-- Creating table R3
CREATE TABLE R3 (
    customerID  INT NOT NULL,
    serieID     INT NOT NULL,
    FOREIGN KEY (customerID) REFERENCES CustomerData(customerID),
    FOREIGN KEY (serieID) REFERENCES Serie(serieID),
    CONSTRAINT uq_fk_customeridserieid UNIQUE (customerID, serieID)
);

-- Creating table R4
CREATE TABLE R4 (
    filmID      INT NOT NULL,
    actorID     INT NOT NULL,
    FOREIGN KEY (filmID) REFERENCES Film(filmID),
    FOREIGN KEY (actorID) REFERENCES Actor(actorID),
    CONSTRAINT uq_fk_filmidactorid UNIQUE (filmID, actorID)
);
```

```sql
-- Creating table R5
CREATE TABLE R5 (
    serieID      INT NOT NULL,
    actorID      INT NOT NULL,
    FOREIGN KEY (serieID) REFERENCES Serie(serieID),
    FOREIGN KEY (actorID) REFERENCES Actor(actorID),
    CONSTRAINT uq_fk_serieidactorid UNIQUE (serieID, actorID)
);
```

II. SQL DDL – Database data insertion

```
-- PostalCodes data
INSERT INTO PostalCodes (postal_code, city, street)
VALUES ('1837EA', 'Hoorn', 'Lepelweg'),
       ('9723GB', 'Groningen', 'Schoenstraat'),
       ('3173PA', 'Amsterdam', 'Kerkstraat'),
       ('8779AE', 'Groningen', 'Woortmanslaan'),
       ('9761WQ', 'Assen', 'Reitdiep'),
       ('6554OL', 'Wageningen', 'Van Royenlaan'),
       ('2232WO', 'Rotterdam', 'Emmastraat'),
       ('9218PE', 'Groningen', 'Helpeerplein'),
       ('9871GB', 'Groningen', 'Van Ketchwich Verschuurlaan'),
       ('7661EE', 'Nieuwegein', 'Olsterveste')
;

-- customerdata
INSERT INTO CustomerData (first_name, last_name, postal_code, house_number,
date_of_birth, e_mail)
VALUES ('William', 'Shakespear', '1837EA', '101', '1951-11-28',
'wshakespear@yahoo.com'),
       ('Andrew', 'Gower', '1837EA', '32', '1961-11-28',
'gower61@gmail.com'),
       ('Laura', 'Nguyen', '9723GB', '1789', '1940-11-28',
'lauranguyen@gmail.com'),
       ('Casper', 'Spook', '9761WQ', '36C', '1975-11-28',
'caspertspookje@gmail.com'),
       ('Dennis', 'Zwart', '2232WO', '398', '1999-11-28',
'dennis9991@live.nl'),
       ('Tom', 'Riddle', '6554OL', '8E', '1992-11-28',
'lordvoldemort@live.nl'),
       ('Mark', 'Zuckerberg', '9218PE', '33', '1980-11-28',
'mriseeyou@fb.com'),
       ('Barrack', 'Obama', '7661EE', '2A', '1955-11-28',
'mrstealyogirl@gmail.com'),
       ('Harry', 'Kuiper', '9723GB', '876', '1940-11-28',
'tonnenmaker@yahoo.nl'),
       ('Lisa', 'Kuiper', '8779AE', '178', '1990-11-28',
'lisa1990@hotmail.com')
;


-- plans
INSERT INTO Plans (plan, timeframe)
VALUES ('Platinum', 730),
       ('Gold', 365),
       ('Silver', 182),
       ('Bronze', 31),
       ('Trial', 7)
;

-- subscriptiondata
#subscriptiondata
INSERT INTO SubscriptionData (customerID, plan, start_date)
VALUES (1, 'Trial', '2020-01-15'),
       (2, 'Gold', '2019-12-19'),
       (3, 'Gold', '2018-09-01'),
       (4, 'Platinum', '2020-01-20'),
       (5, 'Trial', '2019-01-02'),
       (6, 'Bronze', '2018-11-09'),
       (7, 'Silver', '2015-06-21'),
```

```sql
        (8, 'Platinum', '2019-05-29'),
        (9, 'Platinum', '2019-10-31'),
        (10, 'Bronze', '2020-12-20')
;

-- serie
INSERT INTO Serie (title, genre, rating, price)
VALUES  ('Breaking Bad', 'Drama', 9.5, 15.50),
        ('Game of Thrones', 'Drama', 9.3, 20),
        ('Friends', 'Comedy', 9.0, 10),
        ('House of Cards', 'Drama', 8.8, 20),
        ('Stranger Things', 'Adventure', 8.8, 13.50),
        ('Family Guy', 'Comedy', 8.5, 15.50),
        ('Vikings', 'History', 8.0, 7.50),
        ('South Park', 'Comedy', 9.0, 9.50),
        ('Modern Family', 'Comedy', 8.2, 5),
        ('Arrow', 'Action', 8.0, 10),
        ('La Casa De Papel', 'Mystery', 8.5, 4.50)
;

-- actor
INSERT INTO Actor (first_name, last_name, date_of_birth, agency)
VALUES  ('Morgan', 'Freeman', '1937-06-01', 'CAA'),
        ('Tim', 'Robbins', '1958-10-16', 'CAA'),
        ('Bradley', 'Cooper', '1975-01-05', 'BWA'),
        ('Leonardo', 'DiCaprio', '1974-11-11', 'CAA'),
        ('Matthew', 'McConaughey', '1969-11-04', 'BWA'),
        ('Bradd', 'Pitt', '1963-12-18', 'CAA'),
        ('Robert', 'De Niro', '1943-08-17', 'BWA'),
        ('Joaquin', 'Phoenix', '1974-10-28', 'CAA'),
        ('Jack', 'Nicholson', '1937-04-22', 'GERSH'),
        ('Shelley', 'Duvall', '1949-07-07', 'GERSH'),
        ('Scatman', 'Crothers', '1910-05-23', 'GERSH'),
        ('Travis', 'Fimmel', '1979-07-15', 'BWA'),
        ('Sofia', 'Vergara', '1972-07-10', 'BWA'),
        ('Stephen', 'Amell', '1981-05-08', 'BWA'),
        ('Ursula', 'Corbero', '1989-08-11', 'CAA'),
        ('Alvaro', 'Morte', '1975-02-23', 'GERSH'),
        ('Bryan', 'Cranston', '1956-03-07', 'CAA'),
        ('Emilia', 'Clarke', '1986-10-23', 'GERSH'),
        ('Jennifer', 'Aniston', '1969-02-11', 'BWA'),
        ('Kevin', 'Spacey', '1959-07-26', 'GERSH'),
        ('Millie', 'Bobby Brown', '2004-02-19', 'CAA')
;


-- film
INSERT INTO film (title, genre, director_firstname, director_lastname,
length, release_date, price)
VALUES  ('Shawshank Redemption', 'Drama', 'Frank', 'Darabont', '2:22:00',
'1995-03-02', 10.50),
        ('The Place Beyond The Pines', 'Crime', 'Derek', 'Cianfrance',
'2:20:00', '2013-04-04', 14.50),
        ('Shutter Island', 'Mystery', 'Martin', 'Scorsese', '2:19:00',
'2010-02-16', 16.30),
        ('Interstellar', 'Adventure', 'Christopher', 'Nolan', '2:22:00',
'2014-03-02', 18.80),
        ('Se7en', 'Crime', 'David', 'Fincher', '2:08:00', '1995-11-09',
4.60),
        ('Joker', 'Crime', 'Todd', 'Phillips', '2:02:00', '2019-10-02',
7.50),
```

```sql
        ('Gladiator', 'Action', 'Ridley', 'Scott', '2:51:00', '2000-05-18', 
7.50),
        ('Django Unchained', 'Drama', 'Quentin', 'Tarantino', '2:45:00', 
'2013-01-17', 7.95),
        ('Lion King', 'Animation', 'Jon', 'Favreau', '1:58:00', '2019-07-
17', 6.75),
        ('The Shining', 'Drama', 'Stanley', 'Kubrick', '2:26:00', '1980-10-
30', 6.50),
        ('Needle', 'horror',  'John', 'Soto',  '1:30:00', '2010-10-09', 
4.60)
;


-- r2
INSERT INTO r2 (customerID, filmID)
VALUES (1, 1),
        (1, 4),
        (1, 5),
        (2, 3),
        (3, 9),
        (4, 5),
        (5, 3),
        (5, 9),
        (5, 8),
        (6, 8),
        (6, 9),
        (6, 2),
        (6, 1),
        (8, 8),
        (8, 9),
        (9, 1),
        (9, 8),
        (9, 9)
;

-- r3
INSERT INTO r3 (customerID, serieID)
VALUES (2, 4),
        (2, 8),
        (2, 9),
        (3, 9),
        (3, 3),
        (4, 5),
        (5, 5),
        (6, 5),
        (8, 5),
        (9, 8),
        (9, 9),
        (9, 2),
        (1, 1),
        (1, 4),
        (1, 5),
        (1, 6),
        (1, 7),
        (1, 9)
;

-- r4
INSERT INTO r4 (filmID, actorID)
VALUES (1, 1),
    (1, 2),
```

```sql
    (2, 3),
    (3, 4),
    (4, 5),
    (5, 1),
    (5, 6),
    (6, 7),
    (6, 8),
    (7, 8),
    (8, 4),
    (10, 9),
    (10, 10),
    (10, 11),
    (11, 12)

;

-- r5
INSERT INTO r5 (serieID, actorID)
VALUES (1, 17),
    (2, 18),
    (3, 19),
    (4, 20),
    (5, 21),
    (7, 12),
    (9, 13),
    (10, 14),
    (11, 15),
    (11, 16)
;
```

III. E-R Diagram