GMSFlix

Group number : 3

Surname: Al Mallouhi Name: Satea Student number :22011503

Surname : Dayioglu Name :Gürgün Student number :22024990

Surname : El Gargouri Name :Mariem Student number :22015070

Specification for GMSFlix

The following is a specification for the creation of a streaming service, GMSFlix.
The implementation uses MySQL.

GMSFlix (like Netflix) allows people to watch high-quality Content (movies, TV shows, etc.) from the comfort of their homes. Here are the entity types:

- Person is a supertype entity that will be inherited by Member, Director, Actor. A Person is identified by a unique ID, and has first name and last name attributes.
- Account is identified by an email address and requires a password to be created. An Account can be of two types: free or premium. A Free Account has a preferred ad category to help us serve relevant advertisements, and for Premium Accounts, we store their billing intervals and their subscription end dates so that we can inform them when their subscription is over.
- Member is a subtype entity with a last login date attribute (TIMESTAMP) for us to track their login activity.
- Director is also a subtype entity. Same for Actor, but it has one extra attribute, ethnicity.
- Producer is any company that creates Content for our users. It has an ambiguous name attribute, which means it doesn't have to be unique.
- Content represents any type of consumable media and is made up of a type, category, title, and description.

Moving on to relationships:

- An account can have up to 4 Members. A Member has to be tied to an Account.
- Any number of Producers can Produce a Content.
- Any number of Directors can Direct a Content.
- Any number of Actors can Play in a Content.
- Any number of Members can Watch a Content. The Content can be Watched multiple times a day by a Member.
- A Member can leave a Review containing a rating between 0 and 10 and an optional comment for a Content **after** Watching it.

Data Dictionary :

Table account :

Nom	NULL ?	Type	semantics
EMAIL	NOT NULL	VARCHAR2(255)	The email address of the creator of the account and the primary key of the Table account.
PASSWORD	NOT NULL	VARCHAR2(255)	The password of the account.

Table premium_account :

Nom	NULL ?	Type	semantics
EMAIL	NOT NULL	VARCHAR2(255)	The email address of the acccount and also a foreign key referencing the Table account.
PREMIUM_END_DATE	NOT NULL	DATE	The date of the end of premium privileges of the account.
BILLING_INTERVAL	NOT NULL	VARCHAR2(255)	The interval the user chose to pay for the subscription (monthly or yearly).

Table free_account :

Nom	NULL ?	Type	semantics
EMAIL	NOT NULL	VARCHAR2(255)	The email address of the acccount and also a foreign key referencing the Table account.
PREFERED_AD_CATEGORY		VARCHAR2(255)	The ad category that the account user prefers.

Table person :

Nom	NULL ?	Type	semantics
ID	NOT NULL	NUMBER(38)	The identification number of the person and the primary key of the Table person.
LAST_NAME	NOT NULL	VARCHAR2(255)	Last name of the person.
FIRST_NAME	NOT NULL	VARCHAR2(255)	First name of the person.

Table member :

Nom	NULL ?	Type	semantics
ID	NOT NULL	NUMBER(38)	The identification number of the member and a foreign key referencing the Table person.
EMAIL	NOT NULL	VARCHAR2(255)	The email address of the member and also a foreign key referencing the Table account.
LAST_LOGIN		TIMESTAMP(6)	The date and time of the most recent time the member has logged into the account.

Table actor :

Nom	NULL ?	Type	semantics
ID	NOT NULL	NUMBER(38)	The identification number of the actor and a foreign key referencing the Table person.

ETHNICITY	VARCHAR2(255)	The ethnicity of the actor.

Table director :

Nom	NULL ?	Type	semantics
ID	NOT NULL	NUMBER(38)	The identification number of the director and a foreign key referencing the Table person.

Table producer :

Nom	NULL ?	Type	semantics
ID	NOT NULL	NUMBER(38)	The identification number of the producer and the primary key of the Table producer.
NAME	NOT NULL	VARCHAR2(255)	Name of the producer or the production company.

Table content :

Nom	NULL ?	Type	semantics
ID	NOT NULL	NUMBER(38)	The identification number of the content and the primary key of the Table content.
TYPE	NOT NULL	VARCHAR2(255)	Type of content (movie, series etc).
CATEGORY	NOT NULL	VARCHAR2(255)	Category of content (Romance, Comedy etc).

TITLE	NOT NULL	VARCHAR2(255)	Title of the content.
DESCRIPTION		VARCHAR2(255)	Description for the content to give the user an idea about it.

Table produce :

Nom	NULL ?	Type	semantics
IDP	NOT NULL	NUMBER(38)	The identification number of the producer producing/has produced the content. Also a foreign key referencing the Table producer and a part of the primary key of the Table produce.
IDC	NOT NULL	NUMBER(38)	The identification number of the content being/has been produced. Also a foreign key referencing the Table content and a part of the primary key of the Table produce.

Table direct :

Nom	NULL ?	Type	semantics
IDD	NOT NULL	NUMBER(38)	The identification number of the director directing/has directed the content. Also a foreign key referencing the Table director and a part of the primary key of the Table direct.

IDC	NOT NULL NUMBER(38)	The identification number of the content being/has been directed. Also a foreign key referencing the Table content and a part of the primary key of the Table direct.

Table play_in :

Nom	NULL ?	Type	semantics
IDA	NOT NULL	NUMBER(38)	The identification number of the actor playing in the content. Also a foreign key referencing the Table actor and a part of the primary key of the Table play_in.
IDC	NOT NULL	NUMBER(38)	The identification number of the content being played in. Also a foreign key referencing the Table content and a part of the primary key of the Table play_in.

Table watch :

Nom	NULL ?	Type	semantics
IDM	NOT NULL	NUMBER(38)	The identification number of the member watching the content. Also a foreign key referencing the Table member and a part of the primary key of the Table watch.
IDC	NOT NULL	NUMBER(38)	The identification number of the content being/ has been watched. Also a foreign key referencing the Table content and a part of the primary key of the Table watch.
DATEW	NOT NULL	TIMESTAMP(6)	The date and time the member watched the concerned content. Also a part of the primary key of the Table watch.

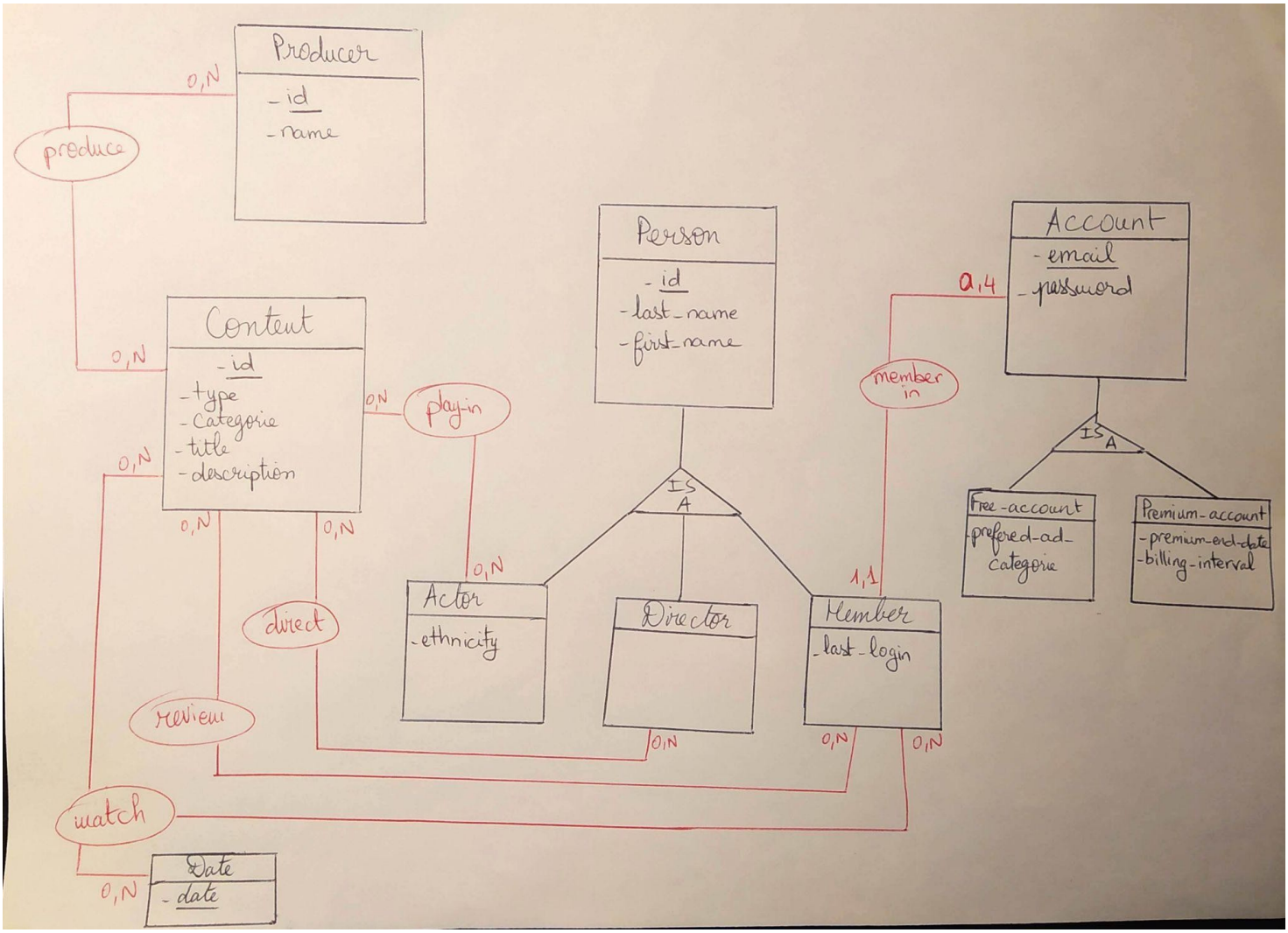
Table review :

Nom	NULL ?	Type	semantics
IDM	NOT NULL	NUMBER(38)	The identification number of the member reviewing the content. Also a foreign key referencing the Table member

		and a part of the primary key of the Table review.
IDC	NOT NULL NUMBER(38)	The identification number of the content being/ has been reviewed. Also a foreign key referencing the Table content and a part of the primary key of the Table review.
RATING	NOT NULL NUMBER(38)	How much the member rated the content out of ten.
COMMENTR	VARCHAR2(255)	The comment the member chose to write after rating the content.

Entity-Association schema :

To represent this data base, we chose the following Entity-Association schema :



The entities are represented in black squares and the relations are represented in red circles with lines connecting all its parties.

Relational schema :

As for the relational schema, this is what we suggest the following :

person(id,first_name,last_name,age)
member(#id,#email,last_login)
actor(#id,ethnicity)
director(#id)
producer(id,name)
content(id,category,type,title,discription)
account(Email,password)
free_account(#Email,prefered_ad_category)
premium_account(#Email,premium_end_date,billing_interval)

produce(#idP,#idC)
direct(#idD,#idC)
play_in(#idA,#idC)
watch(#idC,#idM,#date)
review(#idM,#idC,rating,comment)

The foreign keys are preceded by # and the primary keys are underlined.

QUERIES :

After creating the data base and inserting information in the different tables, we decided to look for specific components in it. For that, we used the following queries :

1. A query that shows all the user's id that watched a film that starts with the letter "w" (no duplicates) :

```
SELECT idM FROM watch JOIN Content ON idC=content.id where type='film'
                                    AND title like'w%'
                                        GROUP BY idM;
```

2. A query that shows actors that participated in all the films/series:

```
SELECT id FROM actor WHERE NOT EXISTS(
    SELECT * FROM content WHERE NOT EXISTS(
        SELECT * FROM play_in WHERE
            play_in.idC=content.id AND
                play_in.idA=actor.id));
```

3. A query that shows The accounts (email and billing interval) with highest bailling interval:

```
SELECT Email, billing_interval FROM  premium_account WHERE
        billing_interval=
                (select max(billing_interval) FROM premium_account);
```

4. A query that shows people that have the same last name :

```
SELECT * FROM person p1 WHERE EXISTS(
        SELECT * FROM person p2 WHERE p1.last_name=p2.last_name AND
                                                        p1.id=p2.id);
```

5. A query that shows the titles of contents that have never been watched:

```
SELECT DISTINCT title FROM content WHERE id NOT IN
                                        (SELECT idC from watch);
```

6. A query that shows the titles of contents that have been watched by at least two different members:

```
SELECT title FROM content c, watch w1 , watch w2
        WHERE w1.idC=c.id
                AND w2.idC=c.id
                        AND w1.idM<>w2.idM;
```

7. A query that gives the number of actors that have a wanted ethnicity (in this example, it's turkish):

```
SELECT count(*) FROM actor WHERE ethnicity like 'turkish';
```

Procedure, Function and Triggers:

After testing the information we had on our database, we created a procedure, a function and some triggers because there were conditions that had to be met. In the following part, we'll be explaining them.

Procedure and function :

1. Procedure director_actor:

```
DELIMITER //

CREATE PROCEDURE director_actor(idP INT)
BEGIN
  DECLARE does_belong BOOLEAN;

  SELECT COUNT(*) INTO does_belong
  FROM actor
  WHERE id = idP;

  IF does_belong THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'Person is already an actor.';
  END IF;

  SELECT COUNT(*) INTO does_belong
  FROM director
  WHERE id = idP;

  IF does_belong THEN
  SIGNAL SQLSTATE '45000'
  SET MESSAGE_TEXT = 'Person is already a director.';
  END IF;
END //

DELIMITER ;
```

The procedure is created to verify if the person is already an actor or a director.

The first thing we do in this procedure is to declare a variable of type boolean : does_belong, then we inject in it the result of a query that checks if our given id : idP, already exists in the table actor. This query uses COUNT(*), and if we get 0 from this count, our boolean is FALSE, otherwise, it's TRUE. If does_belong has the value TRUE, we get an error and a message informing the user that the person is already an actor.

We use the same logic in the second part of the procedure to check if the idP given belongs to a director's id.

2. Function has_watched:

```
DELIMITER //

CREATE FUNCTION has_watched(idM INT)
RETURNS BOOLEAN
BEGIN
  DECLARE has_watched BOOLEAN;

  SELECT COUNT(*) INTO has_watched
  FROM watch
  WHERE idM = idM;

  RETURN has_watched;
END //

DELIMITER ;
```

This function is created to verify if the member has watched content.

It gets one parameter of type integer, we call it idM. And, at the end, it returns a boolean. To get the wanted result, we declare a boolean variable has_watched. We call our query that counts how many times the Id given in the parameter equals a member Id in the table watch. If COUNT(*) gives 0, we get FALSE, else we get TRUE.

Triggers:

1. Trigger check_account_free :

```
DELIMITER //

CREATE TRIGGER check_account_free
BEFORE INSERT ON premium_account
FOR EACH ROW
BEGIN
    DECLARE is_free BOOLEAN;

    SELECT COUNT(*) INTO is_free
    FROM free_account
    WHERE email = NEW.email;

    IF is_free THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'The account cannot be free and paid at the same time.';
    END IF;
END //

DELIMITER ;
```

This trigger is created to check if the account is already a free account before adding it to the premium_account table.

In this trigger we use the same logic we used with the function and the procedure. The only difference is NEW.email which replaces their parameter and it contains the email we want to check.

2. Trigger check_account_premium

```
DELIMITER //

CREATE TRIGGER check_account_premium
BEFORE INSERT ON free_account
FOR EACH ROW
BEGIN
    DECLARE is_premium BOOLEAN;

    SELECT COUNT(*) INTO is_premium
    FROM premium_account
    WHERE email = NEW.email;

    IF is_premium THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'The account cannot be free and paid at the same time.';
    END IF;
END //

DELIMITER ;
```

This trigger is created to check if the account is already a premium account before adding it to the free_account table.

It uses the same logic explained for trigger 1. .

3. Trigger check_num_members:

```
DELIMITER //

CREATE TRIGGER check_num_members
BEFORE INSERT ON member
FOR EACH ROW
BEGIN
    DECLARE num_members INT;

    SELECT COUNT(*) INTO num_members
    FROM member
    WHERE email = NEW.email;

    IF num_members >= 4 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'The account that this member will be connected to already has 4 members.';
    END IF;
END //

DELIMITER ;
```

This trigger is created to keep our condition of having a maximum of four members in one account. It checks if the number of members in one account has reached 4 or more and if that's the case, the user gets an error for wanting to add more members.

4. Trigger check_watch_before_insert :

```
DELIMITER //

CREATE TRIGGER check_watch_before_insert
BEFORE INSERT ON review
FOR EACH ROW
BEGIN
  IF NOT has_watched(NEW.idM) THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'Member has not watched the content';
  END IF;
END //

DELIMITER ;
```

This trigger is created so that a member can't review a content that they haven't watched.

It calls the function has_watched to verify if the member in concern has watched the content and if they haven't, it returns an error.

5. Triggers to call the procedure:

```
DELIMITER //

CREATE TRIGGER check_actor_is_director
BEFORE INSERT ON actor
FOR EACH ROW
BEGIN
  CALL director_actor(NEW.id);
END //

DELIMITER ;
```

```
DELIMITER //

CREATE TRIGGER check_director_is_actor
BEFORE INSERT ON director
FOR EACH ROW
BEGIN
  CALL director_actor(NEW.id);
END //

DELIMITER ;
```

These two triggers are created to call our procedure and to check if an actor is a director or vice versa.

Testing our triggers:

```sql
INSERT INTO person VALUES (16, 'Emma', 'Louise');
INSERT INTO person VALUES (17, 'Arthur', 'Young');
INSERT INTO person VALUES (18, 'James', 'Young');
INSERT INTO person VALUES (19, 'Eddy', 'Black');
INSERT INTO person VALUES (20, 'Gabriel', 'Pattinson');
INSERT INTO account VALUES ('theYoungs@gmail.com', 'YounG222!');

INSERT INTO member VALUES (16, 'theYoungs@gmail.com', NULL);
INSERT INTO member VALUES (17, 'theYoungs@gmail.com', NULL);
INSERT INTO member VALUES (18, 'theYoungs@gmail.com', NULL);
INSERT INTO member VALUES (19, 'theYoungs@gmail.com',NULL);
-- Testing Trigger check_num_members --
-- INSERT INTO member VALUES (20, 'theYoungs@gmail.com', NULL); -- ERROR

INSERT INTO free_account VALUES ('theYoungs@gmail.com', NULL);

--Testing Trigger check_account_free--
-- INSERT INTO premium_account VALUES ('theYoungs@gmail.com', '1000-01-02', 'YEARLY'); -- ERROR

INSERT INTO director VALUES (16);
--Testing Trigger check_actor_is_director--
-- INSERT INTO actor VALUES (16); -- ERROR

INSERT INTO actor VALUES (17);
--Testing Trigger check_director_is_actor--
-- INSERT INTO director VALUES (17); -- ERROR


--Testing Trigger check_check_watch_before_insert--
--INSERT INTO review VALUES (18,3,5,'a so-so movie'); -- ERROR
```