A7: Integrity constraints. Indexes, triggers and user functions

In this artifact we present the physical scheme, where we study the database load; some queries and updates about the database; the most useful indexes and some triggers we will use to grant the integrity and consistency of the database.

Physical Schema

In this section, we predict the number of tuples in each table.

Table	**Tuples**	**Description**
Format	Dozens	An entry for each format of movie in store.
Studio	Hundreds	An entry for each Production studio.
Actor	Thousands	An entry for each actor.
Director	Hundreds	An entry for each director.
Movie	Hundreds of thousands	An entry for each movie in store.
Language	Hundreds	An entry for each Language.
Subtitles for the hearing Impaired	Hundreds	An entry for each Subtitle for the hearing Impaired. Never more entries tan those of the **Language** table.
Audio	Hundreds	An entry for each Audio language. Never more entries tan those of the **Language** table.
Subtitles	Hundreds	An entry for each Subtitle. Never more entries tan those of the **Language** table.
PurchaseMovie	Millions	An entry for each bought movie.
Purchase	Millions	An entry for each purchase. Never more entries than those of the **Purchase** table.
Cart	Hundreds of thousands	An entry for each User.
MovieCart	Millions	An entry for each combination of Movie and Cart that they're in.
Review	Millions	An entry for each review made on a mobie by a user.
User	Hundreds of Thousands	An entry for each User.
Billing Information	Hundreds of Thousands	At most an entry for each user.
Delivery Information	Hundreds of Thousands	At most an entry for each user.
Payment Information	Hundreds of Thousands	At most an entry for each user.
City	Thousands	An entry for each city.
Country	Hundreds	An entry for each Country.
Post-Code	Thousands	An entry for each post-code.

Queries

In this section, we list the most pertinent queries to the website.

1. List movies by genre

```
SELECT movie.imagePath, movie.name, movie.price, format.name
FROM movie, format
WHERE movie.genreID = $genreID AND format.formatID = movie.formatID
```

2. List movies by price

```
SELECT movie.imagePath, movie.name, movie.price, format.name
FROM movie, format
WHERE format.formatID = movie.formatID
ORDER BY movie.price;
```

3. List movies by date

```
SELECT movie.imagePath, movie.name, movie.price, format.name
FROM movie
WHERE movie.releaseDate BETWEEN $date1 AND $date2 AND format.formatID =
movie.formatID
```

4. List featured products (products with most sales)

```
SELECT movie.imagePath, movie.name
FROM (SELECT movieID, COUNT(*) AS purchases FROM PurchaseMovie GROUP BY
movieID) subquery
WHERE subquery.movieID = movie.movieID
ORDER BY subquery.purchases DESC
LIMIT 4;
```

5. List user's transactions

```
SELECT movie.name, purchase.price, purchase.dateOfPurchase,
purchasemovie.quantity
FROM purchase, purchasemovie, movie
WHERE purchase.purchaseID = purchasemovie.purchaseID AND
purchasemovie.movieID = movie.movieID AND purchase.memberID = $memberID;
```

6. List user's cart

```
SELECT movie.name, moviecart.quantity, movie.price, cart.totalCost
FROM movie, cart
JOIN moviecart ON moviecart.cartID = cart.cartID
WHERE cart.cartID = $cartID;
```

7. List movie's reviews

```
SELECT review.title, review.description, review.rating, member.firstName,
member.lastName
FROM Review, Member
WHERE review.memberID = member.memberID AND review.movieID = $movieID;
```

8. Search movies by name

```
SELECT movie.imagePath, movie.name, movie.price, format.name
FROM movie, format
WHERE format.formatID = movie.formatID AND movie.name LIKE $searchString;
```

Changes (UPDATE/DELETE)

In this section, we list the most pertinent changes to the website, whether they are updates on the existent information or the deletion of an entry in the database.

Updates

1. Edit user fields

```
UPDATE member
SET email = $newEmail, password = $newPassword
WHERE (memberID = $memberID
    AND NOT EXISTS (SELECT * FROM member WHERE email = $newEmail));
```

2. Edit movie info

```
UPDATE movie
SET description = $newDescription, name = $newName, runtime = $newRuntime,
releaseDate = $newReleaseDate
WHERE movieID = $movieID;
```

3. Edit delivery address

```
UPDATE deliveryaddress
SET fullName = $newFullname, address = $newAddress
WHERE deliveryAddressID = $deliveryAddressID;
```

4. Edit quantity in purchase

```
UPDATE purchasemovie
SET quantity = $quantity
WHERE purchaseID = $purchaseID AND movieID = $movieID;
```

5. Ban user

```
UPDATE member
```

```
SET bannedMember = 1
WHERE memberID = $memberID;
```

DELETE

1. Delete user

```
DELETE FROM member
WHERE memberID = $memberID;
```

2. Delete movie

```
DELETE FROM movie
WHERE movieID = $movieID;
```

3. Delete movie from cart

```
DELETE FROM cartmovie
WHERE movieID = $movieID AND cartID = $cartID;
```

4. Delete delivery address

```
DELETE FROM deliveryaddress
WHERE deliveryAddressID = $deliveryAddressID;
```

Indexes/Clusters

In this section, we will analyze possible indexes that the database will use for a faster access to certain tables. The chosen triggers will be presented in the following table:

Table	Attribute	Description
Movie	genreID	Hash
Movie	price	B-Tree
Movie	releaseDate	B-Tree
Movie	name	B-Tree
Cart	cartID	Cluster
Format	formatID	Cluster
Genre	genreID	Cluster
Movie	movieID	Cluster
Purchase	purchaseID	Cluster
Review	reviewID	Cluster
Member	memberID	Cluster
Movie	description	GIN (Full Text Search)

B-trees were used in indexes where only the operators <, < = , =, > and >= are required. Hash indexes were used in the cases where only the = operator is needed. Postgre sql automatically creates indexes for each primary key, so we only had to create the following code:

```
CREATE INDEX MovieGenre ON movie USING hash(genreID);

CREATE INDEX MoviePrice ON movie(price);

CREATE INDEX MovieReleaseDate ON movie(releaseDate);

CREATE INDEX MovieName ON movie(name);

CLUSTER cart USING cart_pkey;

CLUSTER format USING format_pkey;

CLUSTER genre USING genre_pkey;

CLUSTER movie USING movie_pkey;

CLUSTER purchase USING purchase_pkey;

CLUSTER review USING review_pkey;

CLUSTER member USING member_pkey;

CREATE INDEX MovieDescription ON movie USING gin(to_tsvector('english',
description));
```

Triggers

In this section, we describe the system's triggers. These triggers guarantee the integrity of the rules specified in the previous artifacts.

1. Add cart to new member

This trigger adds a cart every time a member is added and associates the cart with the member.

```
CREATE OR REPLACE FUNCTION createCart() RETURNS TRIGGER AS $cart_table$
    BEGIN
        INSERT INTO Cart (totalCost) VALUES ();
        UPDATE Member
            SET cartID = (
                SELECT Cart.cartid FROM Cart WHERE Cart.CartID NOT IN (
                    SELECT Cart.cartID FROM Member, Cart WHERE Cart.cartID =
Member.cartID)
                )
            WHERE Member.cartID IS NULL;
        RETURN NEW;
    END;
$cart_table$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS createCartOnInsertMember ON Member;
CREATE TRIGGER createCartOnInsertMember AFTER INSERT ON Member EXECUTE
```

```
PROCEDURE createCart();
```

2. Update average score of movie with changes on reviews

This trigger updates the average score on a movie everyTime a review is added, removed or updated.

```
CREATE OR REPLACE FUNCTION updateAvgScore() RETURNS TRIGGER AS $movie_table$
    BEGIN
        IF
            TG_OP = 'INSERT' OR TG_OP = 'UPDATE' THEN
                UPDATE Movie
                    SET averagescore = (
                        SELECT avg(rating) FROM Review WHERE movieID =
NEW.movieID)
                    WHERE movieID = NEW.MovieID;
        END IF;
        IF
            TG_OP = 'DELETE' OR (TG_OP = 'UPDATE' AND OLD.movieID <>
NEW.movieID) THEN
                UPDATE Movie
                    SET averagescore = (
                        SELECT avg(rating) FROM Review WHERE movieID =
OLD.movieID)
                    WHERE movieID = OLD.movieID;
        END IF;
        IF
            TG_OP = 'INSERT' OR TG_OP = 'UPDATE' THEN
                RETURN NEW;
        ELSE
            RETURN OLD;
        END IF;
    END;
$movie_table$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS updateAverageScoreOfMovieAfterInsert ON Member;
CREATE TRIGGER updateAverageScoreOfMovieAfterInsert AFTER INSERT OR UPDATE
OR DELETE ON Review FOR EACH ROW EXECUTE PROCEDURE updateAvgScore();
```

3. Update total cost on cart after a change on MovieCart

This trigger updates the totalCost on Cart after an insert, update or delete on MovieCart.

```
CREATE OR REPLACE FUNCTION updateCostOnCart() RETURNS TRIGGER AS
$cart_table$
    BEGIN
        IF
            TG_OP = 'INSERT' OR TG_OP = 'UPDATE' THEN
                UPDATE Cart
                    SET totalCost = (
                        SELECT SUM(Movie.price) FROM Movie, MovieCart WHERE
MovieCart.cartID = NEW.cartID AND Movie.movieID = MovieCart.movieID)
```

```
                    WHERE CartID = NEW.CartID;
        END IF;
        IF
            TG_OP = 'DELETE' OR (TG_OP = 'UPDATE' AND OLD.movieID <>
NEW.movieID) THEN
                UPDATE Cart
                    SET totalCost = (
                        SELECT SUM(Movie.price) FROM Movie, MovieCart WHERE
MovieCart.cartID = OLD.cartID AND Movie.movieID = MovieCart.movieID)
                    WHERE CartID = OLD.CartID;
        END IF;
        IF
            TG_OP = 'INSERT' OR TG_OP = 'UPDATE' THEN
                RETURN NEW;
        ELSE
            RETURN OLD;
        END IF;
    END;
$cart_table$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS updateCostOnCart ON MovieCart;
CREATE TRIGGER updateCostOnCart AFTER INSERT OR UPDATE OR DELETE ON
MovieCart FOR EACH ROW EXECUTE PROCEDURE updateCostOnCart();
```

4. Delete Cart of Member when the member is deleted

```
CREATE OR REPLACE FUNCTION deleteCartOfMember() RETURNS TRIGGER AS
$member_table$
    BEGIN
        DELETE FROM Cart WHERE OLD.cartid = Cart.cartid;
        RETURN OLD;
    END;
$member_table$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS deleteCartOfMember ON Member;
CREATE TRIGGER deleteCartOfMember AFTER DELETE ON Member FOR EACH ROW
EXECUTE PROCEDURE deleteCartOfMember();
```