Assignment 2 - 2DV513

Adam Elfström - ae223nv
Vilhelm Park - vp222dv

1. Idea

The idea is to create a database and some features for a new web-based album store. Their problem is that they were formerly a physical store, but want to branch out to become a web-based store as well.

While work has started on the website, they currently have no database that will handle their inventory. Their website is also missing all admin-related features and they are therefore in need of this database and its interface.

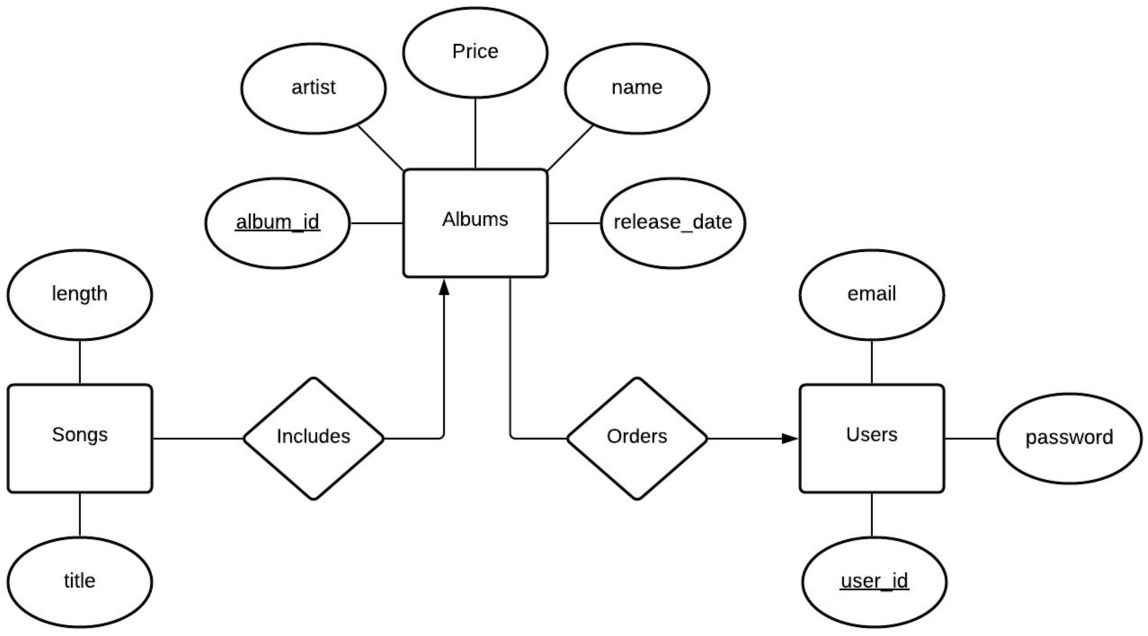
The user of this database and its user interface are the admins for the album store. Customers will use the separate store website instead.

For the basic functionality of the database, we need to be able to insert, read, and change data. For the application to be complete we also need to be able to gather various data from the database, for example, which orders a specific user has placed as well as the current shipping and payment status of each order.

The application will have a web-based interface that will handle all our intended features. The database consists of the tables Albums, Users, Orders, and Songs.

2. Logical Model

2.1 E/R diagram



2.2 Discussion

1. The relationship between the entities *Albums* and *Users* are many-to-one. Many albums can be ordered by one user and one user can order many albums.
2. The relationship between the entities *Songs* and *Albums are many-to-one. Many Songs are included in one Album and one Album includes many Songs.*

This diagram is simple since it only contains two relationships and that connects three entity sets. These entity sets do not contain all that many attributes either. This is because we decided to only include the attributes and entity sets we deemed as being the most important to illustrate the data model.

When we then create the SQL schemas of the database, we can include all the other attributes that are necessary for the database to be fit for the task of storing users, albums, and the orders that these users make. Attributes such as the date or the payment status of the order might be unnecessary for this data model while being vital for the final application to be able to provide the necessary information about orders to the customers. Some of these additional attributes only are necessary for the customers, while others might only be necessary for the admins of the database. We will discuss which attributes will be added to the SQL collection schemas in section 3.2.

3. Design in SQL

3.1 Collection Schemas

Albums(id: integer,
price: integer,
in_stock: boolean,
name: string,
artist: string,
release_date: date**)**

Songs(id: integer,
title: string,
length: integer,
album_id: integer**)**

Users(id: integer,
name: string,
password: string,
email: string**)**

Orders(id: integer,
user_id: integer,
album_id: integer,
order_date: datetime,
ship_status: string,
pay_status: boolean,
quantity: integer**)**

3.2 Discussion

When translating the E/R diagram of the data model into our SQL collections, there were two main options we could go with. Since there are many-to-one relationships, the first option was to create one collection per entity set and relation and then combine these into fewer collections with some added attributes. However, as mentioned in section 2.1, the diagram does not contain every attribute we found to be necessary for the final application, it only contains the most important ones. Combining these relations would therefore also include adding many new attributes, which would make each table in the database contain, in our opinion, too many attributes. Since this is the case, we chose to instead follow option two; which was to not combine the relations but instead keep all three relations separate. This allows us to keep each table smaller which makes the database both easier to understand and use.

Apart from the id attributes, the main difference between the diagram and the collections is that the *Orders* collection contains many new attributes that we felt were vital for the final application.

4. SQL Queries

4.1 Get all orders from one user

```
SELECT * FROM orders
WHERE orders.user_id = x
```

This query selects all orders that a specific user has done, to do this the WHERE clause is used to compare the current order.user_id with the id that is given as input from the web interface. The orders that have the matching user_id values will then be selected and output.

4.2 Get all songs from an album

```
SELECT albums.id, albums.name, songs.title, songs.length
FROM albums
INNER JOIN songs
ON albums.id = songs.album_id
WHERE albums.id = x
```

In this query, the goal is to get all *songs* from a specific *album*, accompanied by the id of the album and the name. To achieve this the keyword *INNER JOIN* is used and the *WHERE* clause. WHERE is used to compare the *albums.id* with the *id* that is input from the web interface. INNER JOIN is then used to group the songs that have matching *album_id* with the *id* that is input into a table.

4.3 User data and the total number of albums ordered

```
SELECT users.id, users.name, users.email,
(SELECT SUM(orders.quantity)
 FROM orders
 WHERE orders.user_id = users.id) AS num_of_albums_ordered
FROM users
```

In this query data about all users is gathered as well as the total number of albums each user has ordered. This is done by the use of the aggregate function *SUM* on the table orders and the column quantity in a subquery. To get the correct number for each user the *WHERE* clause will need to match on the columns *orders.user_id* and *users.id*. This query is used to give a bit more information about each user when viewing the list of users.

4.4 Average cost of all albums

```
SELECT AVG(albums.price) AS Average_Cost_Albums
FROM albums
```

This query displays some extra statistics about the albums we currently have in our database. The query gets the average cost of all the albums in the database, which is done by using the aggregate function *AVG* on the *price* column in the *albums* table.

4.5 Highest spending customers View

```
CREATE OR REPLACE VIEW highestSpenders AS
SELECT users.id, users.name, users.email,
      (SELECT SUM(orders.quantity *
                (SELECT albums.price
                 FROM albums
                 WHERE albums.id = orders.album_id))
       FROM orders
       WHERE users.id = orders.user_id) as amount_spent
FROM users
ORDER BY amount_spent DESC
LIMIT 3
```

```
SELECT name, amount_spent FROM highestspenders
```

This query aims to get the three users who have spent the most amount of money on the site. This is done by first getting the price of each album from the *albums* table in the innermost subquery. Then to see how many of this particular album was bought by the user the *WHERE* clause is used to compare the album id with the current orders album id, if they are the same multiply the *orders* quantity with the album's price. To make sure that we only calculate from a specific user the *WHERE* clause is used to compare the current *users* id with the user id in the order, if they are the same add the previous calculation to the sum.

After every album has been iterated over, the view will have the total cost that a specific user has spent in the webshop. All the users are then ordered in descending order with the *ORDER BY* clause and *DESC*. Finally, to only show the top three spenders the *LIMIT* clause is used.

The reason why this query, in particular, was chosen to be a *VIEW* instead of a normal query is that it is quite complicated and also contains data that might be valuable to use in several different places. For example, this view can be easily used by admins to perhaps send special promotions to the biggest spenders of the site. Instead of re-running the original query each time this data is to be accessed, a much simpler query to the view can instead be executed.

The use case shown in the second query is to display some statistics when viewing the */users* directory on the web interface, and in this case, only two columns are chosen to be displayed.

5. Implementation & Execution

The source code of the implementation can be viewed in the provided “*src.zip*” folder.

To run and test out the application yourself, please read the instructions in the “*README.md*” markdown file. This file can be found alongside the source code in the zip folder.

If you are unable to see the formatting, copy-pasting the file into an online markdown editor as https://dillinger.io/ should work.

6. Video

The video presentation can be viewed at this link:
https://www.youtube.com/watch?v=sx9e5AmBtgg