Assignment 6: Altering Tables, Updating Data, Views, and Triggers

The purpose of this assignment is to give you some practice with intermediate SQL techniques related to updating data and abstracting database structures and operations. You'll also see a few more use cases for views and triggers.

In the starter files, you'll find a database file called **movies.db.** Import this into a new blank Codespace. (Database attribution: https://cs50.harvard.edu/sql/license/, with contributions by Tomoko Okochi.) Take a moment to explore this database and its schema. (It has a LOT more data than the previous databases we've worked with; the fact that we can efficiently process this data speaks to SQLite's speed!)

Create a new file called **a6.sql** to write your queries. Create a comment with the number of the task to label each part of this file.

Task 1: Altering the Ratings Table

As you can see, the **ratings** table has data about the average rating (out of 10) for each movie and the number of votes contributing to that average. You may notice that each row has a unique **id,** and also the **movie_id** of the movie the rating belongs to. But since each movie can have only one rating, and each movie has a unique ID, we could get rid of the **id** column and use the **movie_id** column as a primary key instead. Try removing the **id** column using the following query:

```
ALTER TABLE ratings
DROP COLUMN id;
```

You'll see that it doesn't work; SQLite doesn't allow you to do this! You can comment out that query. The workaround is to rename this table, create a new **ratings** table with the schema you want, and copy the data from the old ratings table to the new one.

- **Figure out how** to use **ALTER TABLE** to change the name of **ratings** to **ratings_temp**
- Create a new table called **ratings** with the same schema as the old one EXCEPT:
    - it should not have an **id** column;
    - the **movie_id** column should be the primary key.

      Hint: you can get the query for the original schema by running **.schema ratings** on the command line.

- Copy the data from **ratings_temp** to **ratings** with the following query:

```
INSERT INTO ratings (movie_id, rating, votes)
SELECT movie_id, rating, votes FROM ratings_temp;
```

- Drop the **ratings_temp** table since you won't be using it anymore.

- The next alteration is to add a new column to **ratings** called **last_rated;** the purpose of this column is to record the date that each movie received its most recent vote. Use the **DEFAULT** keyword to set the default value of each row to **'2024-01-01'** (that's the default date format YYYY-MM-DD).

Task 2: Views

In this task, we'll create a view that presents information about movie ratings to users in a way that is perhaps more useful and interesting.

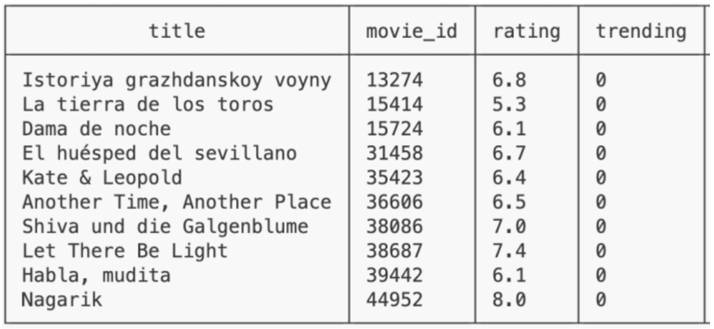
Create a new view called **movie_ratings.** It should have the following properies:

- The **title, movie_id,** and **rating** columns from the tables where they originate;
- A new column called **trending** that has a value of **1** if the movie received a vote in the past 30 days, and **0** if it did not. To do this:
  - Use a **CASE** clause to implement this (check your previous assignment solution if you've forgotten how this works). To get the difference between today's date and the date from the **last_rated** column, you can use this code:

    ```
    JULIANDAY(DATE('now')) - JULIANDAY(last_rated)
    ```

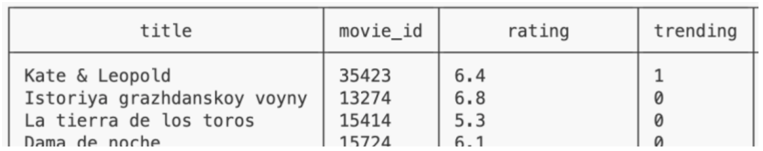
    The **DATE()** function gets today's date, formatted properly.
- The result set should be sorted in **descending order** of **last_rated,** and only the first 10 results should be shown.

Write a query to select all columns from the view; you should see something like this:



```
title                         movie_id  rating  trending

Istoriya grazhdanskoy voyny   13274     6.8     0
La tierra de los toros        15414     5.3     0
Dama de noche                 15724     6.1     0
El huésped del sevillano      31458     6.7     0
Kate & Leopold                35423     6.4     0
Another Time, Another Place   36606     6.5     0
Shiva und die Galgenblume     38086     7.0     0
Let There Be Light            38687     7.4     0
Habla, mudita                 39442     6.1     0
Nagarik                       44952     8.0     0
```

Note that all movies currently have the same **last_rated** date so the order you see might not be the same as above. Also, at this point none of the movies have received a recent vote, so **trending** will be 0 for all rows. Later, we'll implement a way to add new votes, but for now, write a query to artificially set the last_rated date of '**Kate & Leopold'** to today's date so you can test this functionality:



```
title                        movie_id  rating  trending

Kate & Leopold               35423     6.4     1
Istoriya grazhdanskoy voyny  13274     6.8     0
La tierra de los toros       15414     5.3     0
Dama de noche                15724     6.1     0
```

Task 3: Awards

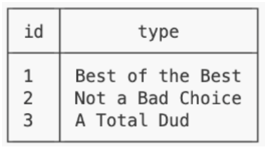
To make this rating data even more interesting, let's implement an awards system, where each movie gets ONE award based on the following:

- If rated less than 5.5, the award is **'A Total Dud'**
- If rated between 5.5 and 7, the award is **'Not a Bad Choice'**
- If higher than 7, it's **'Best of the Best'**

Use the following queries to create a new table for awards and populate the above types:

```
DROP TABLE IF EXISTS awards;

CREATE TABLE awards (
   id        INTEGER,
   type      TEXT,
   PRIMARY KEY (id)
);

INSERT INTO awards (type) VALUES
('Best of the Best'), ('Not a Bad Choice'), ('A Total Dud');
```

Write a query to view the awards table:



```
id  type

1   Best of the Best
2   Not a Bad Choice
3   A Total Dud
```

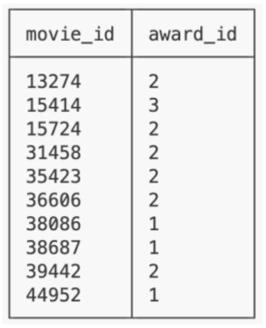
We'll of course need a junction table to connect awards to the correct movies. Create this with the following queries (but fill in the correct code at the TODO comments):

```
DROP TABLE IF EXISTS awarded;

CREATE TABLE awarded (
    movie_id    INTEGER,
    award_id    INTEGER,
    -- TODO: set up a composite primary key with both columns
    -- TODO: set up separate foreign keys connecting the above columns to the     appropriate tables
);
```

Now, we just have to populate the **awarded** table with the correct data to connect each movie to its award, based on its rating. You can do this by using JOIN with CASE; complete the following query to implement this:

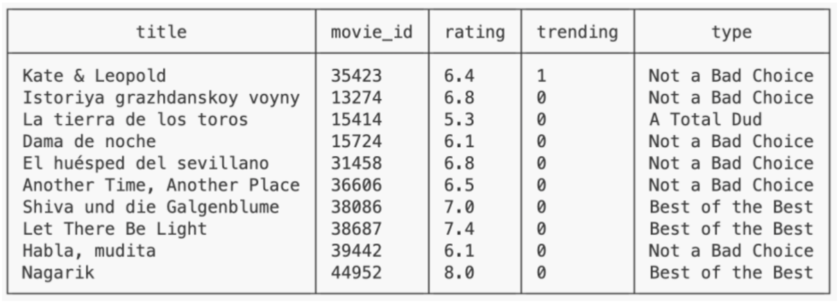
```
INSERT INTO awarded (movie_id, award_id)
SELECT ratings.movie_id, awards.id FROM ratings
JOIN awards ON awards.id = (
    -- TODO: use CASE to get the correct award ID value
);
```

Write a query to view the first 10 rows of this table (sorted in ascending order of movie_id):



movie_id	award_id
13274	2
15414	3
15724	2
31458	2
35423	2
36606	2
38086	1
38687	1
39442	2
44952	1

Now, update your **movie_ratings** view to include the award type (you can't alter views in SQLite, so just go back up to that part of the code and change the query.) The view should now look like this:



title	movie_id	rating	trending	type
Kate & Leopold	35423	6.4	1	Not a Bad Choice
Istoriya grazhdanskoy voyny	13274	6.8	0	Not a Bad Choice
La tierra de los toros	15414	5.3	0	A Total Dud
Dama de noche	15724	6.1	0	Not a Bad Choice
El huésped del sevillano	31458	6.8	0	Not a Bad Choice
Another Time, Another Place	36606	6.5	0	Not a Bad Choice
Shiva und die Galgenblume	38086	7.0	0	Best of the Best
Let There Be Light	38687	7.4	0	Best of the Best
Habla, mudita	39442	6.1	0	Not a Bad Choice
Nagarik	44952	8.0	0	Best of the Best

Task 4: Triggers

The interesting thing about this database is that individual votes are not recorded (as they are in our **longlist-advance** database), so there isn't a straightforward way to add a new vote to the database, at least not without doing some math to calculate the new average rating.

We'd like to implement an abstraction that hides this concern from users and allows them to easily insert new votes and update the average rating.

We already have an abstraction for viewing data: our **movie_ratings** view. We can also attach the vote insertion functionality to this view (and the behind-the-scenes logic to implement it), but it will take a bit of work. Note that SQLite does NOT allow one to insert data into views! (This makes sense when you think about it; a view isn't a table, it's just an abstraction of a SELECT query!) We'll have to use a **trigger** to update the **ratings** table when one tries to insert a new vote into the **movie_ratings** view.

Use this code to initialize the query for creating this trigger:

```
CREATE TRIGGER insert_new_vote
INSTEAD OF INSERT ON movie_ratings
BEGIN
-- TODO: update the following columns of ratings table: rating, votes, last_rated
END;
```

Your job is to replace the TODO comment above with the actual query that updates the **ratings** table appropriately. A few hints:

- The formula for calculating the new average when one more vote is added is:

  (old_votes*old_rating + new_vote_rating)/(old_votes + 1)

- The number of votes should be increased by 1.
- the **last_rated** date should be set to today's date.

To test this, try inserting a new vote to see if it shows up, using the following query!

```
INSERT INTO movie_ratings (movie_id, rating) VALUES
(31458, 10);
```

Of course, there's one more thing to do: update the award if the rating goes above or below the threshold! Write a **trigger** called **update_awards** that ensures the award type is correct after the average rating has been updated for a given movie through the previous trigger. Note that this trigger will be triggered by the previous trigger!

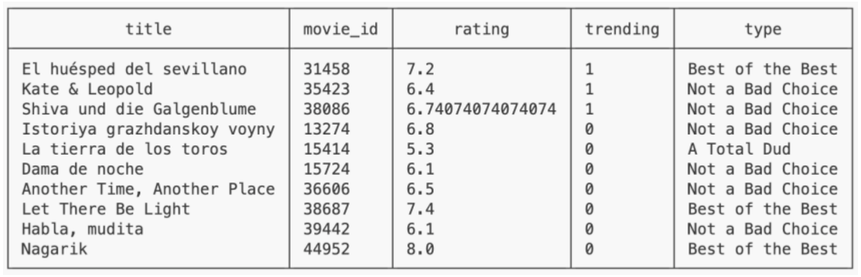
When you're done that, run the following queries to test:

```
INSERT INTO movie_ratings (movie_id, rating) VALUES
(31458, 10);

INSERT INTO movie_ratings (movie_id, rating) VALUES
(31458, 10);

INSERT INTO movie_ratings (movie_id, rating) VALUES
(38086, 0);
```

Since you're now making permanent updates to the database and will therefore see a different result each time you run **a6.sql,** (these operations are not idempotent) you may want to reset the above movies to their previous ratings and vote counts by pasting the following queries at the top of your script (we'll learn better ways of rolling back later):

```
UPDATE ratings
SET
  rating = 6.8,
  votes = 14
WHERE movie_id = 31458;
UPDATE ratings
SET
  rating = 7.0,
  votes = 26
WHERE movie_id = 38086;
```

Now write a query to select the **movie_ratings** view; you should see something like this:



title	movie_id	rating	trending	type
El huésped del sevillano	31458	7.2	1	Best of the Best
Kate & Leopold	35423	6.4	1	Not a Bad Choice
Shiva und die Galgenblume	38086	6.74074074074074	1	Not a Bad Choice
Istoriya grazhdanskoy voyny	13274	6.8	0	Not a Bad Choice
La tierra de los toros	15414	5.3	0	A Total Dud
Dama de noche	15724	6.1	0	Not a Bad Choice
Another Time, Another Place	36606	6.5	0	Not a Bad Choice
Let There Be Light	38687	7.4	0	Best of the Best
Habla, mudita	39442	6.1	0	Not a Bad Choice
Nagarik	44952	8.0	0	Best of the Best

Grading

- Task 1
  - [1 mark] new **ratings** table set up with different PK
  - [1 mark] **last_rated** with default value added to **ratings**
- Task 2
  - [1 mark] **movie_ratings** view with **trending** table and test update to **Kate & Leopold**
- Task 3
  - [1 mark] **awards** and **awarded** tables set up
  - [1 mark] **awarded** populated with correct data
  - [1 mark] **movie_ratings** view updated to include award **type**
- Task 3
  - [1 mark] **insert_new_vote** trigger
  - [1 mark] **update_awards** trigger

Total: **8**

As usual, up to -25% may be deducted for improper hand in, poor code formatting, etc. Ask me if in doubt.

Hand In

Remove all unecessary files (such as these instructions) but keep **movies.db** and your **a6.sql** file. Rename the project folder to **a6-firstname-lastname,** zip the folder, and hand it in to Brightspace.