DM576 – Database Systems

Mathias B. Jensen

2025-05-03

Part 1: Database Design

Contents

1 Introduction

In this report i will go through the process of solving the first part of the five part database project for the DM576 course. Their is in the assignment given a relational schema that defines a streaming platform like Netflix or Disney+. The relational schema is to be used to create an Entity Relationship diagram (ER-diagram). By reverse engineering the relational schema, i could achieve an ER-diagram of the relations. There after the assignment called for a normalization given an assumption. At the end of the first part, their will be an outline of a database that could hold the content that is to be hosted on the streaming platform. This will at last be used to implement and deploy a database in part 2 of the project. In the next section i will explain the conceptual design and the normalization i achieved afterwards. In the assignment i used ChatGPT to solve repetitive tasks, and as a sparring partner, for example for information gathering based on context [1].

2 Conceptual Design

In this section, I explain the ER diagram and outline the key assumptions underlying the design. For clarity and readability, the diagram is divided into two parts: the content part and the user view part. The content part is shown in (Figure 1), and the user view part is presented in (Figure 2).

2.1 Content part

The content part includes the following entities: `Streaming_Platform`, `Content`, `Season`, `Movie`, `TV_Show`, and `Episode`. These entities have the defined attributes and keys as shown in the relational schema (Figure 3). The assumptions made are shown in the arrows used to show many-to-one relations, for example the one arrow going from `Episode` to `Season` where the many-to-one relation is shown as the 3 lines at the beginning of the `Episode` and the line at the end of `Season`. This is also the case between `Content` and `Streaming_Platform`. The reason for this is that there can be many different `Content` but only one host for it, in this case the `Streaming_Platform`. The same can be said for `Episode` and `Season`. There is also used and *IsA*-triangle to show the inheritance used between `Content` and `Movie`, `TV_Show`. There is also used relationship diamonds to show the relation between `Episode`, `TV_Show` and `Season`. The same is true between `Content` and `Streaming_Platform`

2.2 User view part

The user part includes the following entities: `User`, `Watch_History`, `Creator_User`, `Premium_User`, and `Free_User`. These entities have the defined attributes and keys as shown in the relational schema (Figure 3). The assumptions made are shown in the arrows used, as explained above in the content part. There is a many-to-one relationship between `Free_User`, `Premium_User`, `Creator_User`, and `User`. As there is many possible premium users, and so on. To enforce the inheritance between `User` and the user types, there is used an *IsA*-triangle. There is a many-to-one relation between `Watch_History` and `User`, because a single user can only have one watch history, but there may be many watch history entries, one for each user.

3 Normalization

In this section i will go through the normalization of a given relation. The given relation is

```
UserRecord(user_id, first_name, last_name,
      content_id, isUploaded, title,
          show_id, season_number,
     episode_number, number_of_views)
```

To begin the normalization, i look at the relation and check if i can infer some functional dependencies and based on them find out if the relations is in BCNF.

3.1 The functional dependencies

The first FD is clearly seen between `user_id` $\rightarrow$ `first_name`, `last_name`, as the `user_id` uniquely identifies each user, where it follows that the user has a first and a last name in the records. The next FD is between `content_id` $\rightarrow$ `title`, `show_id`, `season_number`, `episode_number`. This shows how one piece of content, fx. a show, defined with a `show_id`, `season_number` and `episode_number`. The last FD is between `user_id`, `content_id` $\rightarrow$ `isUploaded`, `number_of_views`. This allows for the tracking of each Creator users content uploads and the number of views for that content.

3.2 Converting to BCNF

To convert to Boyce-Codd Normal Form we need to check if every non-trivial functional dependency $X \to Y$, X must be a superkey. A superkey K being for a relation R, if K functionally determines all of R. This means that the superkey has to determine all other attributes.

The first FD is not a superkey as `first_name` and `last_name` does not determine all other attributes.

The second FD is not a superkey either as the `title`, `show_id`, `season_number` and `episode_number` does not determine all other attributes.

The last FD is a superkey, The third FD is a superkey because the combination of both `user_id` and `content_id` together uniquely identifies each tuple. Since the first two are not superkeys they have to be decomposed.

The first FD can be decomposed to a separate relation called `User`

$$\text{User}(\text{user\_id}, \text{first\_name}, \text{last\_name})$$

This helps with keeping things structured and separate. This is also seen used in the ER-diagram (Figure 2).

The second FD can be decomposed to a separate relation called `Content`

$$\text{Content}(\text{content\_id},\ \text{title},\ \text{show\_id},\ \text{season\_number},\ \text{episode\_number})$$

This also makes sure that the attributes is separate. This is also seen used in the ER-diagram Figure-(1).

The last thing to do is to rewrite the superkey.

$$\text{UserRecord}(\text{user\_id},\ \text{content\_id},\ \text{is\_Uploaded},\ \text{number\_of\_views})$$

Now all FD's have a left-hand side that is a superkey of their respective relation. Since every relation above is in BCNF, given by the FD property mentioned above. Then all relations is also in 3NF, as the BCNF, is just a more strict version of 3NF.

4 Conclusion

Their has now been created an ER-diagrams and normalized the given relation. The ER-diagrams will be used in the next part, to deploy the database. Their has been shown the steps taken and the assumptions made in the process of solving the assignment.

5 Bibliography

[1] OpenAI. *ChatGPT (GPT-4)*. Used as a sparring partner and for repetitive task assistance in this assignment. Accessed: 2025-04-30. 2025. URL: `https://chat.openai.com`.

6 Appendix

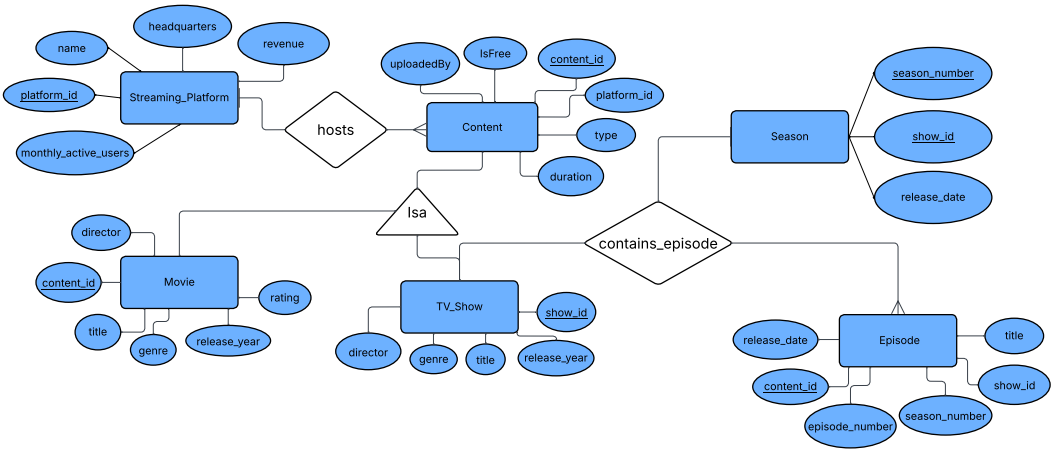


Figure 1: ER Diagram – Content Part

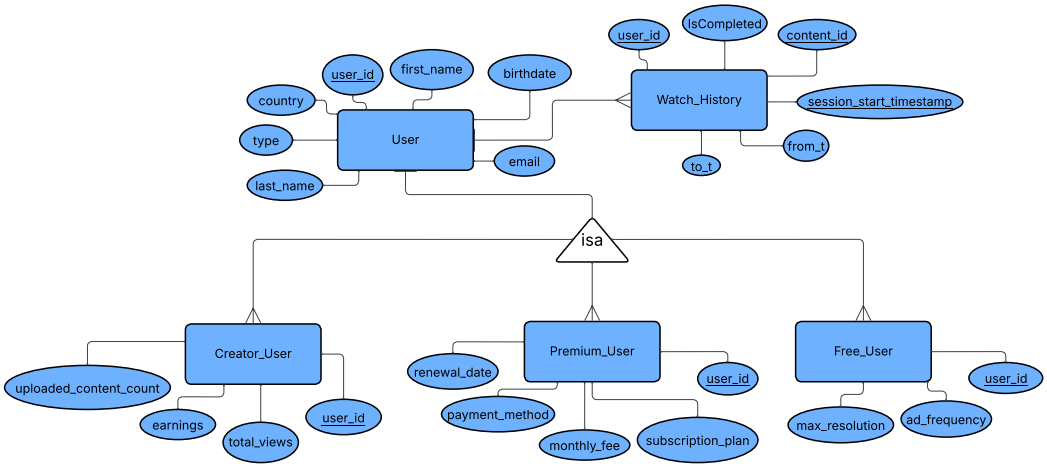


Figure 2: ER Diagram – User View Part

Streaming_Platform(platform_id, name, headquarters, monthly_active_users, revenue)

Content(content_id, platform_id, type, duration, IsFree, uploadedBy)

Movie(content_id, title, release_year, genre, director, rating)

Episode(content_id, show_id, season_number, episode_number, title, release_date)

Season(season_number, show_id, release_date)

TV_Show(show_id, title, release_year, genre, director)

User(user_id, first_name, last_name, email, birthdate, country, type)

Free_User(user_id, ad_frequency, max_resolution)

Premium_User(user_id, subscription_plan, monthly_fee, payment_method, renewal_date)

Creator_User(user_id, uploaded_content_count, total_views, earnings)

Watch_History(user_id, content_id, session_start_timestamp, from_t, to_t, IsCompleted)

Figure 3: Relational Schema

DM576 – Database Systems

Mathias B. Jensen

2025-05-10

Part 2: Database Implementation

Contents

1 Introduction

In this report i will go through the process of solving the second part of the five part database project for the DM576 course. Their is in the assignment given a relational schema that defines a streaming platform like Netflix or Disney+. The relational schema was used to produce an ER-diagram in part 1 of the project [1]. This allows for the implementation of a database using PostgreSQL. This creation and insertion of data in the database is explained and documented in the following sections. In the assignment i used ChatGPT to solve repetitive tasks like data insertion, to populate the tables, and test the constraints, and as a sparring partner, for example for information gathering based on the context of the project [2].

2 Constraints and Foreign keys

In this section i will cover the creation part of the database, this includes the constraints that i chose, and the foreign keys i used. The SQL script i used is in appendix A 6.1. There has been reinforced some constraints for consistency, this includes the NOT NULL used in critical fields, to prevent invalid input. This is for instance seen in the email, first name and last name attribute of the User[1] relation. There is also enforced an age requirement through the check found in the User relation, where a minimum age of 13 years is applied. The watch history relation uses checks to keep the timestamps valid, by using the Unix epoch as reference for a date, and as an extra precaution the from_t and to_t is forced to keep one bigger than the other, to ensure valid timestamps.

In the database their is used some foreign keys to keep data integrity and consistency. Such as in the Free_User, Premium_User and Creator_User relations, to keep the references to the User relation, and to enforce the inheritance shown in the ER-diagram from part 1. In the other relations there is also used foreign keys to keep the references between the different relations.

In appendix C 6.3 the database tables is shown as screenshots from my terminal, where i used PostgreSQL to deploy the database locally. I refrained from using pgAdmin 4 as it had a lot more functionality that i had no use for.

[1] The user keyword is reserved in PostgreSQL, so the relation is named "User"

3 Insertion of data

In this section i will shortly go through the process of the insertion of data into the database created with the SQL file found in appendix A 6.1. The insertion data is generated using Chat-GPT [2] as mentioned in the introduction, where the prompt specified the amount of rows to fill out. The insertion completed with no errors, therefore the constraints given in the creation of the database is upheld, and fulfilled. The insertion script can be found in appendix B 6.2

4 Conclusion

In conclusion there has been created a database, based on the ER-diagram from part 1. This was done using PostgreSQL, and the terminal. The creation script has implemented relevant constraints, and references to ensure data integrity and validity. The database has been populated with information, without violating the constraints set in the creation. It has all been documented in the appendix as referenced.

5 Bibliography

[1] Mathias Bonde Jensen. "DM576 – Database Systems Part 1: Database Design". In: (2025), p. 3.

[2] OpenAI. *ChatGPT (GPT-4)*. Used as a sparring partner and for repetitive task assistance in this assignment. Accessed: 2025-05-07. 2025. URL: `https://chat.openai.com`.

6 Appendix

6.1 Appendix A

```
CREATE TABLE Streaming_Platform(
        platform_id INTEGER PRIMARY KEY,
        name VARCHAR(100) NOT NULL,
        headquarters VARCHAR(100),
        monthly_active_users INTEGER,
        revenue DECIMAL(10,2)
);

CREATE TABLE Movie (
        content_id INTEGER PRIMARY KEY,
        title TEXT,
        release_year INTEGER,
        genre TEXT,
        director VARCHAR(30),
        rating INTEGER
);

CREATE TABLE TV_show (
        show_id INTEGER PRIMARY KEY,
        title TEXT NOT NULL,
        release_year INTEGER NOT NULL,
        genre TEXT NOT NULL,
        director TEXT NOT NULL
);

CREATE TABLE Season (
        season_number INTEGER,
        show_id INTEGER,
        PRIMARY KEY (season_number, show_id),
        release_date DATE,
        FOREIGN KEY (show_id) REFERENCES TV_show(show_id)
);

CREATE TABLE Episode (
```

```
        content_id INTEGER PRIMARY KEY,
        show_id INTEGER NOT NULL,
        season_number INTEGER,
        episode_number INTEGER,
        title TEXT,
        release_date DATE,
        FOREIGN KEY (season_number, show_id) REFERENCES Season(
    season_number, show_id)
);

CREATE TABLE "User" (
        user_id INTEGER PRIMARY KEY,
        first_name TEXT NOT NULL,
        last_name TEXT NOT NULL,
        email TEXT UNIQUE NOT NULL,
        birthdate DATE,
        country TEXT,
        type TEXT,
        CHECK (birthdate <= CURRENT_DATE - INTERVAL '13 years')
);

CREATE TABLE Free_User (
        user_id INTEGER PRIMARY KEY,
        ad_frequency INTEGER,
        max_resolution INTEGER,
        FOREIGN KEY (user_id) REFERENCES "User"(user_id)
);

CREATE TABLE Premium_User (
        user_id INTEGER PRIMARY KEY,
        subscription_plan TEXT,
        monthly_fee DECIMAL(10, 2),
        payment_method TEXT NOT NULL,
        renewal_date DATE NOT NULL,
        FOREIGN KEY (user_id) REFERENCES "User"(user_id)
);

CREATE TABLE Creator_User (
        user_id INTEGER PRIMARY KEY,
        uploaded_content_count INTEGER,
        total_views INTEGER,
        earnings DECIMAL(10,2),
        FOREIGN KEY (user_id) REFERENCES "User"(user_id)
);

CREATE TABLE Content (
        content_id INTEGER PRIMARY KEY,
        platform_ID INTEGER NOT NULL,
        type TEXT,
        duration NUMERIC(5,2),
```

```
        isFree BOOLEAN,
        uploadedBy INTEGER,
        FOREIGN KEY (platform_id) REFERENCES Streaming_Platform(
    platform_id),
        FOREIGN KEY (uploadedBy) REFERENCES "User"(user_id)
);

CREATE TABLE Watch_History (
    user_id INTEGER,
    content_id INTEGER,
    session_start_timestamp TIMESTAMP,
    from_t TIMESTAMP,
    to_t TIMESTAMP,
    isCompleted BOOLEAN,
        PRIMARY KEY (user_id, content_id, session_start_timestamp),
    FOREIGN KEY (user_id) REFERENCES "User"(user_id),
    FOREIGN KEY (content_id) REFERENCES Content(content_id),
    CHECK (from_t > '1970-01-01 00:00:00'),
    CHECK (to_t > '1970-01-01 00:00:00'),
    CHECK (from_t < to_t)
);
```

6.2 Appendix B

```
INSERT INTO "User" (user_id, first_name, last_name, email, birthdate,
    country, type) VALUES
  (1,  'John',   'Doe',      'john.doe@example.com',        '1990-01-15',
    'USA',       'Premium'),
  (2,  'Jane',   'Smith',    'jane.smith@example.com',      '1995-08-24',
    'UK',        'Free'),
  (3,  'Alice',  'Brown',    'alice.b@example.com',         '1985-03-12',
    'Canada',    'Creator'),
  (4,  'Bob',    'Jones',    'bob.j@example.com',           '1993-11-05',
    'Germany',   'Free'),
  (5,  'Eva',    'Green',    'eva.g@example.com',           '1998-07-22',
    'Japan',     'Premium'),
  (6,  'Zoe',    'Miller',   'zoe.miller@example.com',      '2000-02-14',
    'Australia', 'Creator'),
  (7,  'Mike',   'Taylor',   'mike.taylor@example.com',     '1992-05-20',
    'UK',        'Free'),
  (8,  'Laura',  'Wilson',   'laura.wilson@example.com',    '1988-11-10',
    'USA',       'Premium'),
  (9,  'Tom',    'Davis',    'tom.davis@example.com',       '1996-01-18',
    'Canada',    'Creator'),
  (10, 'Olivia', 'Martinez', 'olivia.martinez@example.com', '1994-03-07',
    'Mexico',    'Free'),
  (11, 'Tom',    'Lee',      'tom.lee@example.com',         '1991-02-02',
    'USA',       'Premium'),
  (12, 'Sara',   'King',     'sara.king@example.com',       '1989-07-07',
    'UK',        'Premium'),
  (13, 'Chris',  'Evans',    'chris.evans@example.com',     '1987-06-13',
    'Canada',    'Creator'),
  (14, 'Nina',   'Lopez',    'nina.lopez@example.com',      '1992-12-12',
    'Australia', 'Creator');

INSERT INTO Free_User (user_id, ad_frequency, max_resolution) VALUES
  (2, 3, 720),
  (4, 2, 480),
  (7, 1, 1080),
  (10, 5, 720),
  (6, 4, 1080);

INSERT INTO Premium_User (user_id, subscription_plan, monthly_fee,
    payment_method, renewal_date) VALUES
  (1,  'Standard',  9.99, 'Credit Card',  '2025-06-01'),
  (5,  'Gold',     14.99, 'PayPal',       '2025-06-15'),
  (8,  'Platinum', 19.99, 'Bank Transfer','2025-07-01'),
  (11, 'Standard',  9.99, 'Credit Card',  '2025-06-10'),
  (12, 'Gold',     14.99, 'Debit Card',   '2025-06-20');

INSERT INTO Creator_User (user_id, uploaded_content_count, total_views,
    earnings) VALUES
```

```
  (3,  20, 150000,  3500.50),
  (6,  15, 120000,  2750.00),
  (9,  25, 180000,  4500.00),
  (13, 10,  50000,  1200.00),
  (14, 30, 250000,  7250.00);

INSERT INTO Streaming_Platform (platform_id, name, headquarters,
    monthly_active_users, revenue) VALUES
  (1, 'StreamFlix',  'New York, USA',    120000000, 85000000.00),
  (2, 'WatchNow',    'London, UK',       45000000,  40000000.00),
  (3, 'CineWave',    'Toronto, Canada',  30000000,  28000000.00),
  (4, 'PrimeStream', 'Berlin, Germany',  60000000,  50000000.00),
  (5, 'MegaPlay',    'Tokyo, Japan',     70000000,  60000000.00);

INSERT INTO Content (content_id, platform_id, type, duration, isFree,
    uploadedBy) VALUES
  (101, 1, 'Movie',   120.00, FALSE, 3),
  (102, 1, 'Episode',  45.00, TRUE,  4),
  (103, 2, 'Episode',  50.00, TRUE,  5),
  (104, 3, 'Movie',    95.00, FALSE, 1),
  (105, 4, 'Episode',  60.00, TRUE,  3);

INSERT INTO Movie (content_id, title, release_year, genre, director,
    rating) VALUES
  (101, 'The Great Escape', 2021, 'Drama',       'Alice Johnson', 8),
  (104, 'Deep Dive',        2020, 'Documentary', 'Eric Hall',     7),
  (106, 'Quantum Chase',    2023, 'Sci-Fi',      'Ivy Brooks',    9),
  (107, 'Love Unfiltered',  2022, 'Romance',     'Rita Kim',      6),
  (108, 'Zero Hour',        2019, 'Thriller',    'Matt Lee',      7);

INSERT INTO TV_show (show_id, title, release_year, genre, director)
    VALUES
  (201, 'Galactic Odyssey', 2022, 'Sci-Fi',      'Ben Marsh'),
  (202, 'Kitchen Clash',    2021, 'Reality',     'Naomi Hill'),
  (203, 'Future Law',       2020, 'Drama',       'David Cole'),
  (204, 'Nature Unbound',   2022, 'Documentary', 'Clara Woods'),
  (205, 'Code Hunters',     2023, 'Action',      'Sam Bright');

INSERT INTO Season (season_number, show_id, release_date) VALUES
  (1,  201, '2022-04-01'),
  (2,  201, '2023-05-10'),
  (1,  202, '2021-06-01'),
  (1,  203, '2020-09-15'),
  (1,  204, '2022-02-20');

INSERT INTO Episode (content_id, show_id, season_number, episode_number,
    title, release_date) VALUES
  (102, 201, 1, 1, 'Arrival',          '2022-04-01'),
  (103, 201, 1, 2, 'The Awakening',    '2022-04-08'),
  (105, 202, 1, 1, 'First Flame',      '2021-06-01'),
```

```
  (109, 203, 1, 1, 'Opening Statement','2020-09-15'),
  (110, 204, 1, 1, 'Wild Waters',      '2022-02-20');

INSERT INTO Watch_History (user_id, content_id, session_start_timestamp,
    from_t, to_t, isCompleted) VALUES
  (1, 101, '2025-05-01 20:00:00','2025-05-01 20:00:00','2025-05-01
    22:00:00', TRUE),
  (2, 102, '2025-05-02 15:30:00','2025-05-02 15:30:00','2025-05-02
    16:15:00', FALSE),
  (1, 103, '2025-05-03 12:00:00','2025-05-03 12:00:00','2025-05-03
    12:50:00', TRUE),
  (5, 104, '2025-05-04 18:45:00','2025-05-04 18:45:00','2025-05-04
    20:20:00', TRUE),
  (4, 105, '2025-05-05 14:00:00','2025-05-05 14:00:00','2025-05-05
    15:00:00', FALSE);
```

6.3 Appendix C

```
dm576=> select * FROM "User";
 user_id | first_name | last_name |           email           | birthdate  |  country  |  type
---------+------------+-----------+---------------------------+------------+-----------+---------
       1 | John       | Doe       | john.doe@example.com      | 1990-01-15 | USA       | Premium
       2 | Jane       | Smith     | jane.smith@example.com    | 1995-08-24 | UK        | Free
       3 | Alice      | Brown     | alice.b@example.com       | 1985-03-12 | Canada    | Creator
       4 | Bob        | Jones     | bob.j@example.com         | 1993-11-05 | Germany   | Free
       5 | Eva        | Green     | eva.g@example.com         | 1998-07-22 | Japan     | Premium
       6 | Zoe        | Miller    | zoe.miller@example.com    | 2000-02-14 | Australia | Creator
       7 | Mike       | Taylor    | mike.taylor@example.com   | 1992-05-20 | UK        | Free
       8 | Laura      | Wilson    | laura.wilson@example.com  | 1988-11-10 | USA       | Premium
       9 | Tom        | Davis     | tom.davis@example.com     | 1996-01-18 | Canada    | Creator
      10 | Olivia     | Martinez  | olivia.martinez@example.com | 1994-03-07 | Mexico  | Free
      11 | Tom        | Lee       | tom.lee@example.com       | 1991-02-02 | USA       | Premium
      12 | Sara       | King      | sara.king@example.com     | 1989-07-07 | UK        | Premium
      13 | Chris      | Evans     | chris.evans@example.com   | 1987-06-13 | Canada    | Creator
      14 | Nina       | Lopez     | nina.lopez@example.com    | 1992-12-12 | Australia | Creator
(14 rows)
```

```
dm576=> select * FROM streaming_platform;
 platform_id |    name      |  headquarters   | monthly_active_users |   revenue
-------------+--------------+-----------------+----------------------+-------------
           1 | StreamFlix   | New York, USA   |            120000000 | 85000000.00
           2 | WatchNow     | London, UK      |             45000000 | 40000000.00
           3 | CineWave     | Toronto, Canada |             30000000 | 28000000.00
           4 | PrimeStream  | Berlin, Germany |             60000000 | 50000000.00
           5 | MegaPlay     | Tokyo, Japan    |             70000000 | 60000000.00
(5 rows)
```

```
dm576=> select * FROM Content;
 content_id | platform_id |  type   | duration | isfree | uploadedby
------------+-------------+---------+----------+--------+------------
        101 |           1 | Movie   |   120.00 | f      |          3
        102 |           1 | Episode |    45.00 | t      |          4
        103 |           2 | Episode |    50.00 | t      |          5
        104 |           3 | Movie   |    95.00 | f      |          1
        105 |           4 | Episode |    60.00 | t      |          3
(5 rows)
```

```
dm576=> select * FROM Movie;
 content_id |      title       | release_year |    genre    |   director    | rating
------------+------------------+--------------+-------------+---------------+--------
        101 | The Great Escape |         2021 | Drama       | Alice Johnson |      8
        104 | Deep Dive        |         2020 | Documentary | Eric Hall     |      7
        106 | Quantum Chase    |         2023 | Sci-Fi      | Ivy Brooks    |      9
        107 | Love Unfiltered  |         2022 | Romance     | Rita Kim      |      6
        108 | Zero Hour        |         2019 | Thriller    | Matt Lee      |      7
(5 rows)
```

```
dm576=> select * FROM watch_history;
 user_id | content_id | session_start_timestamp |       from_t        |        to_t         | iscompleted
---------+------------+-------------------------+---------------------+---------------------+-------------
       1 |        101 | 2025-05-01 20:00:00     | 2025-05-01 20:00:00 | 2025-05-01 22:00:00 | t
       2 |        102 | 2025-05-02 15:30:00     | 2025-05-02 15:30:00 | 2025-05-02 16:15:00 | f
       1 |        103 | 2025-05-03 12:00:00     | 2025-05-03 12:00:00 | 2025-05-03 12:50:00 | t
       5 |        104 | 2025-05-04 18:45:00     | 2025-05-04 18:45:00 | 2025-05-04 20:20:00 | t
       4 |        105 | 2025-05-05 14:00:00     | 2025-05-05 14:00:00 | 2025-05-05 15:00:00 | f
(5 rows)
```

```
dm576=> select * FROM Episode;
 content_id | show_id | season_number | episode_number |       title       | release_date
------------+---------+---------------+----------------+-------------------+--------------
        102 |     201 |             1 |              1 | Arrival           | 2022-04-01
        103 |     201 |             1 |              2 | The Awakening     | 2022-04-08
        105 |     202 |             1 |              1 | First Flame       | 2021-06-01
        109 |     203 |             1 |              1 | Opening Statement | 2020-09-15
        110 |     204 |             1 |              1 | Wild Waters       | 2022-02-20
(5 rows)
```

```
dm576=> select * FROM Season;
 season_number | show_id | release_date
---------------+---------+--------------
             1 |     201 | 2022-04-01
             2 |     201 | 2023-05-10
             1 |     202 | 2021-06-01
             1 |     203 | 2020-09-15
             1 |     204 | 2022-02-20
(5 rows)
```

```
dm576=> select * FROM TV_Show;
 show_id |      title       | release_year |    genre    |  director
---------+------------------+--------------+-------------+-------------
     201 | Galactic Odyssey |         2022 | Sci-Fi      | Ben Marsh
     202 | Kitchen Clash    |         2021 | Reality     | Naomi Hill
     203 | Future Law       |         2020 | Drama       | David Cole
     204 | Nature Unbound   |         2022 | Documentary | Clara Woods
     205 | Code Hunters     |         2023 | Action      | Sam Bright
(5 rows)
```

```
dm576=> select * FROM Free_User;
 user_id | ad_frequency | max_resolution
---------+--------------+----------------
       2 |            3 |            720
       4 |            2 |            480
       7 |            1 |           1080
      10 |            5 |            720
       6 |            4 |           1080
(5 rows)
```

```
dm576=> select * FROM Premium_User;
 user_id | subscription_plan | monthly_fee | payment_method | renewal_date
---------+-------------------+-------------+----------------+-------------
       1 | Standard          |        9.99 | Credit Card    | 2025-06-01
       5 | Gold              |       14.99 | PayPal         | 2025-06-15
       8 | Platinum          |       19.99 | Bank Transfer  | 2025-07-01
      11 | Standard          |        9.99 | Credit Card    | 2025-06-10
      12 | Gold              |       14.99 | Debit Card     | 2025-06-20
(5 rows)
```

```
dm576=> select * FROM Creator_User;
 user_id | uploaded_content_count | total_views | earnings
---------+------------------------+-------------+----------
       3 |                     20 |      150000 |  3500.50
       6 |                     15 |      120000 |  2750.00
       9 |                     25 |      180000 |  4500.00
      13 |                     10 |       50000 |  1200.00
      14 |                     30 |      250000 |  7250.00
(5 rows)
```

DM576 – Database Systems

Mathias B. Jensen

2025-05-17

Part 3: Querying

Contents

1 Introduction

In this report i will go through the process of solving the third part of the five part database project for the DM576 course. Their is in the assignment given a relational schema that defines a streaming platform like Netflix or Disney+. The relational schema was used to produce an ER-diagram in part 1 of the project [1]. This allowed for the deployment of a database in part 2. This now allows for querying the existing database. The following sections will go through and document the process of creating these queries. In the assignment i used ChatGPT to reevaluate my queries, find documentation on PostgreSQL and reformulate a sentence [2].

2 Assignments

In this section i will go through each of the assignments given. There will for each section be an output of the query, found in the appendix, as well as the SQL script used.

2.1 Retrieve a list

The first queries goal was to retrieve a list with all the movies and TV shows offered by streaming platforms. The result was to include the platform name, the title of the movie or TV show, and the type of the content. The SQL script is seen in appendix A figure 1, where the goal was to grab the platform name, content type and the title of the content. This is done by querying the content relation, where the movie and streaming platform relations are joined based on their shared keys. This is in turn combined with another query by using the UNION keyword. The query grabs the platform name, content type, and title from the content relation, where the episode and TV show relations are joined based on their shared keys. This is then grouped by streaming platform and title. The output is seen in appendix B figure 6.

2.2 Calculate amount of uploads

The second queries goal is to assume that the relation Creator_User was not available. The query needs to calculate for each user how many content files she/he has uploaded. The query needs to return the user id, the users first name, and last name plus the number of uploaded files. The SQL script is seen in appendix A figure 2. This is achieved by selecting the user id, first name and last name, while using the COUNT keyword to count the amount of content,

by looking at the content id in the content relation. Then we left join the user table to the content table by the uploadedBy attribute. This ensures that even if the user had no content uploaded, it would show as zero in the table. The last thing it does is group it by the information that we want to show. The output is shown in appendix B figure 7.

2.3 Retrieve complete views

The third queries goal is to retrieve for each content id, the number of complete views, where the number of complete views is less than 2. The query should return the content id, and the number of complete views. The SQL script is seen in appendix A figure 3. This is achieved by grabbing the content id, and count the complete views using the COUNT keyword. This is grabbed from the watch history relation, where the isCompleted boolean is true. Then it is grouped by the content id and constricted to only include those that have a count less than 2. The output is shown in appendix B figure 8.

2.4 Finding dates with maximum content

The fourth queries goal is to find the date/dates of 2024 with the maximum number of content views. The query is expected to return the date and number of views. If two dates have a tie, both is to be shown. The SQL script is seen in appendix A figure 4. To find these dates, we start by using the WITH keyword for improved readability. This uses a sub query where we find the date from the session_start_timestamp attribute, where we then count the views using the COUNT keyword. Then we filter it with the WHERE EXTRACT keywords to match the date asked for. In the assignment there is asked for 2024, but the input data used in part 2 of the project does not have any dates in 2024, only 2025, therefore the SQL query is modified to grab those instead. At last we group by date. The next part of the query selects the maximum views using MAX, and with views aliased from the count we did before, we in turn get the view count. The last part of the query selects the view date attribute and the number of views from the view_counts attribute, joining it with max_views on the view count to grab only the dates with the maximum number of views[1]. This ensures that if there are multiple dates tied, they are all included. The output is shown in appendix B figure 9.

[1] Reformulated this sentence with the help of Chat-GPT [2]

2.5 Creating a view

The fifth and last queries goal is to create a view for the UserRecord of the first part of the project. For simplification of the query, we assume that the number of views corresponds to how many times the user has completely watched the content. The SQL script is seen in appendix A figure 5. The query used the CREATE VIEW keywords to create the UserRecord view, with some given parameters. These parameters are from part 1 of the project, where a given relation where to be normalized. Using a CASE WHEN keyword, to check if the user was the one who uploaded it and returns true or false based on it. The COALESCE keyword selects the first non-null value, so in this case it uses either the movie or the episode title, based on which is not null. Then it counts the content id in the watch history relation. The next part of the query joins watch history to get viewing activity, and joins content to get the content data. Then it left joins movie and episode to get the title and episode details. The last part is filtering with the WHERE keyword and the GROUP BY to group the output. The output is shown in appendix B figure 10.

3 Conclusion

In conclusion there has been shown and documented the creation of five queries, that each answered the given assignment. The five queries and their output has been referenced and shown in the appendix section. Every query has grabbed the desired data, and been explained in each section.

4 Bibliography

[1] Mathias Bonde Jensen. "DM576 – Database Systems Part 1: Database Design". In: (2025), p. 3.

[2] OpenAI. *ChatGPT (GPT-4)*. Used as a sparring partner and for repetitive task assistance in this assignment. Accessed: 2025-05-16. 2025. URL: `https://chat.openai.com`.

5 Appendix

Appendix A: SQL Queries

```sql
SELECT
    sp.name AS platform_name,
    'Movie' AS content_type,
    m.title AS content_title
FROM Content c
JOIN Movie m ON c.content_id = m.content_id
JOIN Streaming_Platform sp ON c.platform_id = sp.platform_id

UNION

SELECT
    sp.name AS platform_name,
    'TV Show' AS content_type,
    tv.title AS content_title
FROM Content c
JOIN Episode e ON c.content_id = e.content_id
JOIN TV_Show tv ON e.show_id = tv.show_id
JOIN Streaming_Platform sp ON c.platform_id = sp.platform_id
GROUP BY sp.name, tv.title;
```

Figure 1: SQL Query 1

```sql
SELECT
    "User"."user_id",
    "User"."first_name",
    "User"."last_name",
    COUNT(Content.content_id) AS uploaded_file_count
FROM "User"
LEFT JOIN Content ON "User"."user_id" = Content.uploadedBy
GROUP BY "User"."user_id", "User"."first_name", "User"."last_name"
```

Figure 2: SQL Query 2

```sql
SELECT
    content_id,
    COUNT(*) AS complete_views
FROM Watch_History
WHERE isCompleted = TRUE
GROUP BY content_id
HAVING COUNT(*) < 2;
```

Figure 3: SQL Query 3

```sql
WITH view_counts AS (
    SELECT
        DATE(session_start_timestamp) AS view_date,
        COUNT(*) AS views
    FROM Watch_History
    WHERE EXTRACT(YEAR FROM session_start_timestamp) = 2025
    GROUP BY DATE(session_start_timestamp)
),
max_views AS (
    SELECT MAX(views) AS max_view_count FROM view_counts
)
SELECT
    vc.view_date,
    vc.views
FROM view_counts vc
JOIN max_views mv ON vc.views = mv.max_view_count;
```

Figure 4: SQL Query 4

```sql
CREATE VIEW UserRecord AS
SELECT
    u.user_id,
    u.first_name,
    u.last_name,
    c.content_id,
    CASE WHEN c.uploadedBy = u.user_id THEN TRUE ELSE FALSE END AS
    isUploaded,
    COALESCE(m.title, e.title) AS title,
    e.show_id,
    e.season_number,
    e.episode_number,
    COUNT(wh.content_id) AS number_of_views
FROM
    "User" u
JOIN
    Watch_History wh ON u.user_id = wh.user_id
JOIN
    Content c ON wh.content_id = c.content_id
LEFT JOIN
    Movie m ON c.content_id = m.content_id
LEFT JOIN
    Episode e ON c.content_id = e.content_id
WHERE
    wh.IsCompleted = TRUE
GROUP BY
    u.user_id, u.first_name, u.last_name,
    c.content_id, c.uploadedBy,
    m.title, e.title, e.show_id, e.season_number, e.episode_number;
```

Figure 5: SQL Query 5

Appendix B: Screenshots

platform_name	content_type	content_title
CineWave	Movie	Deep Dive
StreamFlix	TV Show	Galactic Odyssey
WatchNow	TV Show	Galactic Odyssey
PrimeStream	TV Show	Kitchen Clash
StreamFlix	Movie	The Great Escape

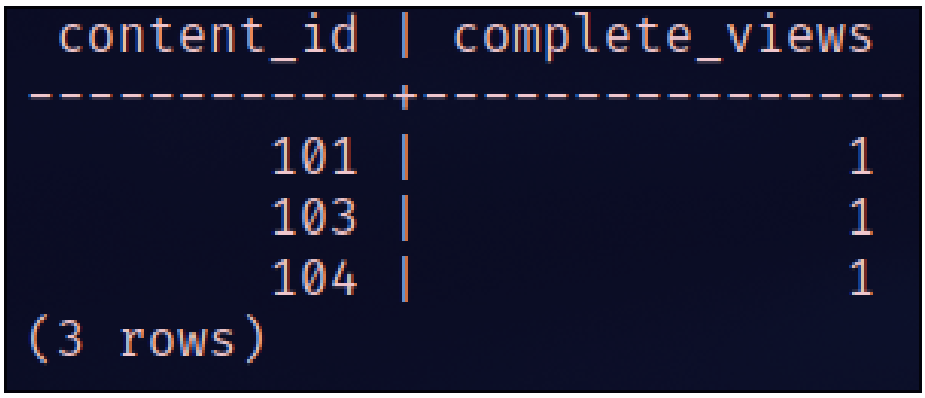
(5 rows)

Figure 6: Screenshot 1 Output

user_id	first_name	last_name	uploaded_file_count
5	Eva	Green	1
4	Bob	Jones	1
10	Olivia	Martinez	0
6	Zoe	Miller	0
14	Nina	Lopez	0
13	Chris	Evans	0
2	Jane	Smith	0
7	Mike	Taylor	0
1	John	Doe	1
8	Laura	Wilson	0
11	Tom	Lee	0
9	Tom	Davis	0
3	Alice	Brown	2
12	Sara	King	0

(14 rows)

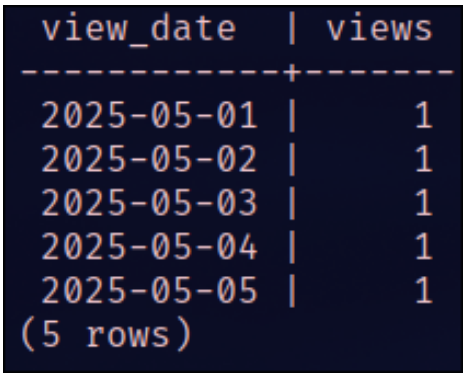
Figure 7: Screenshot 2 Output



content_id	complete_views
101	1
103	1
104	1

(3 rows)

Figure 8: Screenshot 3 Output



view_date	views
2025-05-01	1
2025-05-02	1
2025-05-03	1
2025-05-04	1
2025-05-05	1

(5 rows)

Figure 9: Screenshot 4 Output

```
 user_id | first_name | last_name | content_id | isuploaded |      title      | show_id | season_number | episode_number | number_of_views
---------+------------+-----------+------------+------------+-----------------+---------+---------------+----------------+-----------------
       5 | Eva        | Green     |        104 | f          | Deep Dive       |         |               |                |               1
       1 | John       | Doe       |        103 | f          | The Awakening   |     201 |             1 |              2 |               1
       1 | John       | Doe       |        101 | f          | The Great Escape |         |               |                |               1
(3 rows)
```

Figure 10: Screenshot 5 Output (Rotated for Readability)

DM576 – Database Systems

Mathias B. Jensen

2025-05-24

Part 4: Triggers

Contents

1 Introduction

In this report i will go through the process of solving the fourth part of the five part database project for the DM576 course. Their is in the assignment given a relational schema that defines a streaming platform like Netflix or Disney+. The relational schema was used to produce an ER-diagram in part 1 of the project [1]. This has in turn allowed for the creation of the UserRecord relation, as the part 3 version only was a view, which does not directly support data insertions per definition. The UserRecord creation script can be seen in appendix A figure 1. This part of the project called for the creation for triggers that could keep the UserRecord relation updated, and furthermore updating the attributes uploaded_content_count and total_views of the relation Creator_User, so that they reflect reality. The following sections will go through and document the process of creating and testing these triggers. In the assignment i used ChatGPT to create testing statements for insertion of data and find documentation on PostgreSQL [2].

2 Triggers

In this section i will go through the different triggers that has been created. The triggers and functions are all collected in one file and can be found in appendix A listing 1. The testing of these triggers and functions can be seen in Appendix B.

In creation of these triggers and functions i use the CREATE OR REPLACE keyword to ensure that if there is already a trigger active, the trigger is replaced, and not created as that would lead to an error complaining about it already existing. I also use the "meta" prefix to make it clear if i grab data from another relation, like the movie or the episode relation.

A general structure is used in the functions, where there is used aliases where possible, and metadata to make it clearer what information is pulled from elsewhere, and what is coming directly from the relations in question.

The first set of triggers and functions is used to update the UserRecord relation and the Creator_User relation based on the Content relation. The one called trg_content_after_insert and is used to update the UserRecord relation after the insertion into Content. The trigger uses an if statement to select either Movie, Episode or NULL as the metadata for the insertion into UserRecord. By using this metadata i can insert the given data from Content into the UserRecord in the specified columns. Then at last the Creator_User is updated, and the uploaded_content_count is incremented by one, to match the addition of new content. The the trigger calls the function if there is executed an insertion

on the content relation. To test these triggers i started with an overview of the empty UserRecord and the populated Creator_User relation, seen in figure 2. Then i insert some content i generated with the help of ChatGPT-4 [2]. This then updates the relations seen in figure 3. To make it possible to delete any content and in turn update the User_Record and Creator_User i needed to create an equivalent trigger and function that does the reverse. The function is called trg_content_after_delete and updates the UserRecord relation based on the deletion of data in the Content relation. When content is deleted in the relation Content, it decrements the value of uploaded_content_count and removes the user from the UserRecord relation. This is seen in figure 4.

The second set of triggers and functions is used to update the UserRecord relation and the Creator_User relation based on the update of the Watch_History relation. The one called trg_watch_after_insert handles the update of the UserRecord and Creator_User based on the insertion of data into the Watch_History relation. By grabbing data from the Content relation, UserRecord relation, it allows the function to use an if statement to decided where to pull the metadata from based on if the content is a movie, episode or NULL. This in turn makes it possible to update the UserRecord relation with the new data, and at last increment the total_views part in the Creator_User relation. The trigger is executed based on if there is an insertion on the Watch_History relation. Like the Content relation triggers, i needed to support the deletion in the Watch_History relation by using a trigger that is the equivalent to the insertion one. The function is called trg_watch_after_delete and uses the uploadedBy and number_of_views attributes to evaluate an if statement that decides the specific entry to delete from the UserRecord, and at last update the Creator_User to decrement the total_views attribute. This is triggered by the deletion of an entry in the Watch_History. The testing of these triggers and functions is seen in appendix B, and structured the same as the Content relation triggers. Before insertion the overview is as seen in figure 5, and the relation after insertion is seen in figure 6. The deletion of an entry and the update of the UserRecord and Creator_User is seen in figure 7.

These are the triggers that i choose to keep the database up to date based on the assignment given. There is a lot of possibilities and one of them is a trigger for updating the Watch_History directly, but in general a watch history is an append only structure. When a user watches content on the platform the watch history is updated by the two triggers, that either inserts new data, or removes new data. Since the alteration of the table directly without going through content is assumed to not be supported because of the append only structure, there is no chance for the Watch_History relation to be inconsistent. There could also be created triggers for the Premium_User and Free_User relations too support the same, and a lot more.

3 Conclusion

In conclusion there has been created four triggers and functions, to keep the database up to date based on the Content, UserRecord and Creator_User relations. This has been shown, documented, discussed and tested in this report.

4 Bibliography

[1] Mathias Bonde Jensen. "DM576 – Database Systems Part 1: Database Design". In: (2025), p. 3.

[2] OpenAI. *ChatGPT (GPT-4)*. Used as a sparring partner and for repetitive task assistance in this assignment. Accessed: 2025-05-23. 2025. URL: `https://chat.openai.com`.

5 Appendix

Appendix A: SQL Scripts

```sql
DROP VIEW IF EXISTS UserRecord;

CREATE TABLE UserRecord (
        user_id INT
                REFERENCES "User"(user_id),
        first_name TEXT,
        last_name TEXT,
        content_id INT
                REFERENCES Content(content_id)
                ON DELETE CASCADE,
        isUploaded BOOLEAN,
        title TEXT,
        show_id INT,
        season_number INT,
        episode_number INT,
        number_of_views NUMERIC,
        PRIMARY KEY (user_id, content_id)
)
```

Figure 1: SQL script for UserRecord relation

```
CREATE OR REPLACE FUNCTION trg_content_after_insert() RETURNS trigger AS
    $$
DECLARE
  meta_title       TEXT;
  meta_show_id     INTEGER;
  meta_season_no   INTEGER;
  meta_episode_no  INTEGER;
BEGIN

  IF NEW.type = 'Movie' THEN
    SELECT title, NULL, NULL, NULL
      INTO meta_title, meta_show_id, meta_season_no, meta_episode_no
    FROM Movie
    WHERE content_id = NEW.content_id;

  ELSIF NEW.type = 'Episode' THEN
    SELECT title, show_id, season_number, episode_number
      INTO meta_title, meta_show_id, meta_season_no, meta_episode_no
    FROM Episode
    WHERE content_id = NEW.content_id;
  ELSE
    meta_title := NULL;
    meta_show_id := NULL;
    meta_season_no := NULL;
    meta_episode_no := NULL;
  END IF;

  -- This updates the UserRecord row after insert into the Content
  -- Where the uploadedBy is matched to the user_id.
  INSERT INTO UserRecord (
    user_id, first_name, last_name,
    content_id, isUploaded,
    title, show_id, season_number, episode_number,
    number_of_views
  )
  SELECT
    u.user_id, u.first_name, u.last_name,
    NEW.content_id, TRUE,
    meta_title, meta_show_id, meta_season_no, meta_episode_no,
    0
  FROM "User" AS u
  WHERE u.user_id = NEW.uploadedby;

  -- This increments the uploaded_content_count in Creator_User by 1
  -- Where the user_id is matched to the uploadedBy attribute.
  UPDATE Creator_User
    SET uploaded_content_count = uploaded_content_count + 1
    WHERE user_id = NEW.uploadedby;
```

```
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;

  -- This triggers the function above based on if there has been an
     insert into Content.
CREATE TRIGGER content_after_insert
  AFTER INSERT ON Content
  FOR EACH ROW EXECUTE FUNCTION trg_content_after_insert();

  -- This decreases the uploaded_content_count in Creator_User by 1
  -- Where the user_id is matched to the uploadedBy attribute.
CREATE OR REPLACE FUNCTION trg_content_after_delete() RETURNS trigger AS
    $$
BEGIN
  DELETE FROM UserRecord
    WHERE content_id = OLD.content_id
      AND isUploaded = TRUE;

  UPDATE Creator_User
    SET uploaded_content_count = uploaded_content_count - 1
    WHERE user_id = OLD.uploadedby;

  RETURN NULL;
END;
$$ LANGUAGE plpgsql;

  -- This triggers the function above based on if there has been a
     deletion in Content.
CREATE TRIGGER content_after_delete
  AFTER DELETE ON Content
  FOR EACH ROW EXECUTE FUNCTION trg_content_after_delete();

CREATE OR REPLACE FUNCTION trg_watch_after_insert() RETURNS trigger AS $$
DECLARE
  uploader          INTEGER;
  existing_views    INTEGER;
  meta_title        TEXT;
  meta_show_id      INTEGER;
  meta_season_no    INTEGER;
  meta_episode_no   INTEGER;
  content_type      TEXT;
BEGIN
  SELECT uploadedby, type INTO uploader, content_type
  FROM Content
  WHERE content_id = NEW.content_id;

  SELECT COUNT(*) INTO existing_views
  FROM UserRecord
  WHERE user_id = NEW.user_id
```

```
    AND content_id = NEW.content_id
    AND isUploaded = FALSE;

  IF existing_views > 0 THEN
    -- bump the count
    UPDATE UserRecord
    SET number_of_views = number_of_views + 1
    WHERE user_id = NEW.user_id
      AND content_id = NEW.content_id
      AND isUploaded = FALSE;
  ELSE
    -- fetch metadata based on content type
    IF content_type = 'Movie' THEN
      SELECT title, NULL, NULL, NULL
      INTO meta_title, meta_show_id, meta_season_no, meta_episode_no
      FROM Movie
      WHERE content_id = NEW.content_id;

    ELSIF content_type = 'Episode' THEN
      SELECT title, show_id, season_number, episode_number
      INTO meta_title, meta_show_id, meta_season_no, meta_episode_no
      FROM Episode
      WHERE content_id = NEW.content_id;
    ELSE
      meta_title := NULL;
      meta_show_id := NULL;
      meta_season_no := NULL;
      meta_episode_no := NULL;
    END IF;

    -- insert first watch record
    INSERT INTO UserRecord (
      user_id, first_name, last_name,
      content_id, isUploaded,
      title, show_id, season_number, episode_number,
      number_of_views
    )
    SELECT
      u.user_id, u.first_name, u.last_name,
      NEW.content_id, FALSE,
      meta_title, meta_show_id, meta_season_no, meta_episode_no,
      1
    FROM "User" AS u
    WHERE u.user_id = NEW.user_id;
  END IF;

  UPDATE Creator_User
  SET total_views = total_views + 1
  WHERE user_id = uploader;
```

```
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;

  -- This triggers the function above based on if there has been an
     insert into Watch_History.
CREATE TRIGGER watch_after_insert
  AFTER INSERT ON Watch_History
  FOR EACH ROW EXECUTE FUNCTION trg_watch_after_insert();

 -- This does the same as the trg_watch_after_insert(), but in reverse,
     based on deletion.
CREATE OR REPLACE FUNCTION trg_watch_after_delete() RETURNS trigger AS $$
DECLARE
  uploader  INTEGER;
  views     INTEGER;
BEGIN
  SELECT uploadedby INTO uploader
    FROM Content
    WHERE content_id = OLD.content_id;

  UPDATE UserRecord
    SET number_of_views = number_of_views - 1
    WHERE user_id = OLD.user_id
      AND content_id = OLD.content_id
      AND isUploaded = FALSE;

  SELECT number_of_views INTO views
    FROM UserRecord
    WHERE user_id = OLD.user_id
      AND content_id = OLD.content_id
      AND isUploaded = FALSE;

  IF views <= 0 THEN
    DELETE FROM UserRecord
      WHERE user_id = OLD.user_id
        AND content_id = OLD.content_id
        AND isUploaded = FALSE;
  END IF;

  UPDATE Creator_User
    SET total_views = total_views - 1
    WHERE user_id = uploader;

  RETURN NULL;
END;
$$ LANGUAGE plpgsql;

  -- This triggers the function above based on if there has been a
     delete in Watch_History.
```

```sql
CREATE TRIGGER watch_after_delete
  AFTER DELETE ON Watch_History
  FOR EACH ROW EXECUTE FUNCTION trg_watch_after_delete();
```

Listing 1: SQL triggers

Appendix B: Screenshots

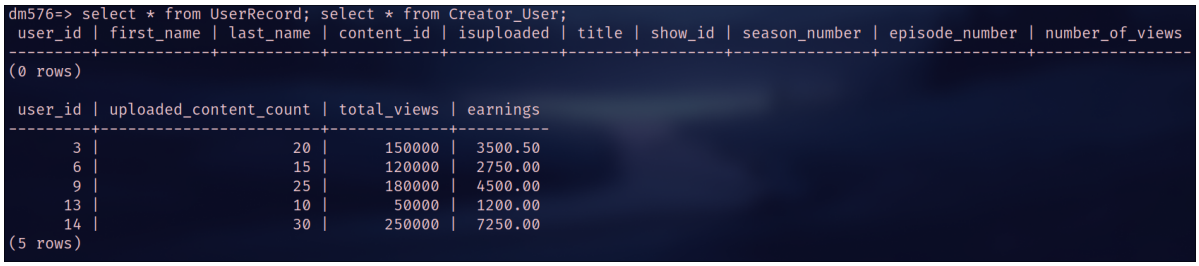


Figure 2: Content trigger before insert

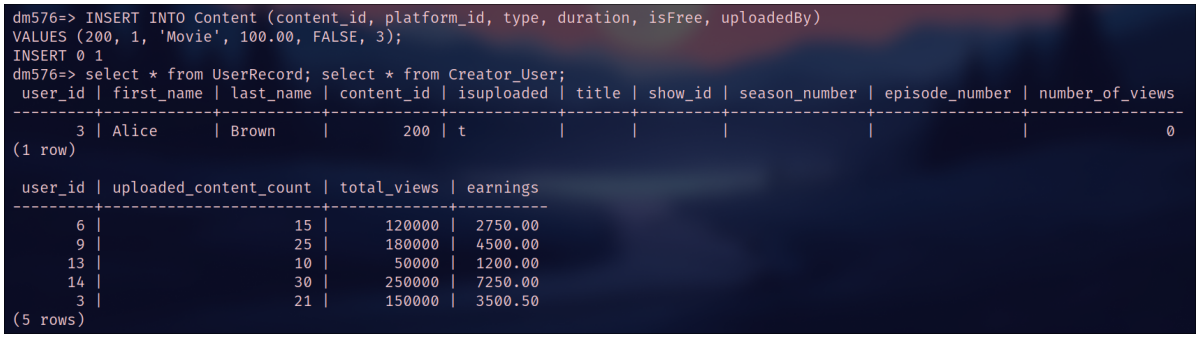


Figure 3: Content trigger after insert

```
dm576=> delete from Content where content_id = 200;
DELETE 1
dm576=> select * from UserRecord;
 user_id | first_name | last_name | content_id | isuploaded | title | show_id | season_number | episode_number | number_of_views
---------+------------+-----------+------------+------------+-------+---------+---------------+----------------+-----------------
(0 rows)

dm576=> select * from Creator_User;
 user_id | uploaded_content_count | total_views | earnings
---------+------------------------+-------------+----------
       6 |                     15 |      120000 |  2750.00
       9 |                     25 |      180000 |  4500.00
      13 |                     10 |       50000 |  1200.00
      14 |                     30 |      250000 |  7250.00
       3 |                     20 |      150000 |  3500.50
(5 rows)
```

Figure 4: Content trigger after deletion

```
dm576=> select * from UserRecord; select * from Creator_User;
 user_id | first_name | last_name | content_id | isuploaded | title | show_id | season_number | episode_number | number_of_views
---------+------------+-----------+------------+------------+-------+---------+---------------+----------------+-----------------
(0 rows)

 user_id | uploaded_content_count | total_views | earnings
---------+------------------------+-------------+----------
       6 |                     15 |      120000 |  2750.00
       9 |                     25 |      180000 |  4500.00
      13 |                     10 |       50000 |  1200.00
      14 |                     30 |      250000 |  7250.00
       3 |                     20 |      150000 |  3500.50
(5 rows)
```

Figure 5: Watch History trigger before insert

```
dm576=> INSERT INTO Watch_History (user_id, content_id, session_start_timestamp, from_t, to_t, isCompleted)
VALUES (1, 101, '2025-05-10 10:00:00', '2025-05-10 10:00:00', '2025-05-10 12:00:00', TRUE);
INSERT 0 1
dm576=> select * from UserRecord; select * from Creator_User;
 user_id | first_name | last_name | content_id | isuploaded |      title       | show_id | season_number | episode_number | number_of_views
---------+------------+-----------+------------+------------+------------------+---------+---------------+----------------+-----------------
       1 | John       | Doe       |        101 | f          | The Great Escape |         |               |                |               1
(1 row)

 user_id | uploaded_content_count | total_views | earnings
---------+------------------------+-------------+----------
       6 |                     15 |      120000 |  2750.00
       9 |                     25 |      180000 |  4500.00
      13 |                     10 |       50000 |  1200.00
      14 |                     30 |      250000 |  7250.00
       3 |                     20 |      150001 |  3500.50
(5 rows)
```

Figure 6: Watch History trigger after insert

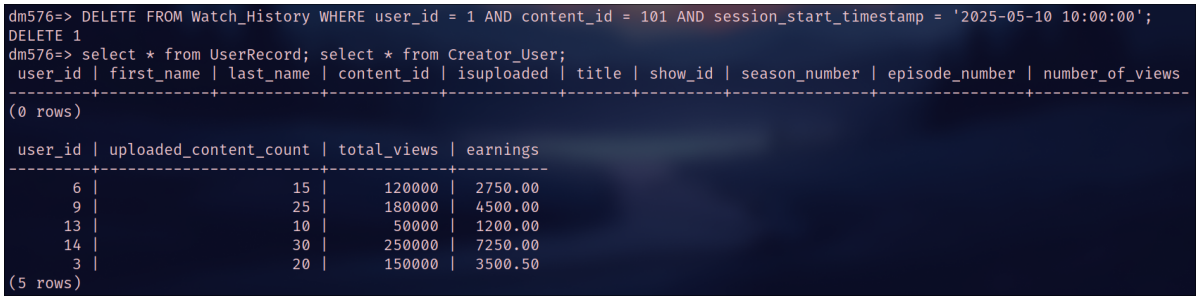


Figure 7: Watch History trigger after deletion

DM576 – Database Systems

Mathias B. Jensen

2025-05-31

Part 5: Indexing

Contents

1 Introduction

In this report i will go through the process of solving the fifth and final part of the five part database project for the DM576 course. Their is in the assignment given a relational schema that defines a streaming platform like Netflix or Disney+. The fifth part called for the use of indexing to optimize the database queries from part 3. In the assignment i used ChatGPT to reevaluate the indexing, as a sparring partner and to find more information on PostgreSQL [1].

2 Indexing

In this section i will go through the three indexes i have created to optimize the queries from part 3 of the project. The first query needed to grab the amount of content files that was uploaded by a `Creator_User`. The query is seen in Appendix A figure 1. To optimize this we can target the join condition used, and then index the `Content.uploadedBy` attribute. This will in turn improve the performance of the join action. The index script is seen below.

```
CREATE INDEX idx_content_uploadedby ON Content(uploadedBy);
```

The second queries goal was to retrieve the `content id`, the number of complete views, where the number is less than 2. The query has a filter based on if the `isCompleted` attribute is true and the grouping by `content_id`. A composite index on (`isCompleted, content_id`) would optimize both the filtering and the grouping part. The index script is seen below.

```
CREATE INDEX idx_watchhistory_completed_contentid
ON Watch_History(isCompleted, content_id);
```

The third query had the goal of finding the dates of 2024[1] with the maximum number of content views. The query has a filter on the `session_start_timestamp` attribute and the group by date clause. An index on just the `session_start_timestamp` is enough for the `WHERE` clause and `GROUP BY DATE` part. The index script is seen below.

```
CREATE INDEX idx_watchhistory_sessiontimestamp
ON Watch_History(session_start_timestamp);
```

[1] In part 3 of the project i used 2025 because of my testing data not having 2024

3 Conclusion

In conclusion there have been created three different index scripts. The indexes has been presented and explained. The use of indexing in this project would give a very small difference as the database does not contain a lot of data. If the database had a lot of data, the difference would be much more noticeable.

4 Bibliography

[1] OpenAI. *ChatGPT (GPT-4)*. Used as a sparring partner and for repetitive task assistance in this assignment. Accessed: 2025-05-30. 2025. URL: `https://chat.openai.com`.

5 Appendix

Appendix A: SQL Scripts

```sql
SELECT
    "User"."user_id",
    "User"."first_name",
    "User"."last_name",
    COUNT(Content.content_id) AS uploaded_file_count
FROM "User"
LEFT JOIN Content ON "User"."user_id" = Content.uploadedBy
GROUP BY "User"."user_id", "User"."first_name", "User"."last_name"
```

Figure 1: SQL Query 1

```sql
SELECT
    content_id,
    COUNT(*) AS complete_views
FROM Watch_History
WHERE isCompleted = TRUE
GROUP BY content_id
HAVING COUNT(*) < 2;
```

Figure 2: SQL Query 2

```sql
WITH view_counts AS (
    SELECT
        DATE(session_start_timestamp) AS view_date,
        COUNT(*) AS views
    FROM Watch_History
    WHERE EXTRACT(YEAR FROM session_start_timestamp) = 2025
    GROUP BY DATE(session_start_timestamp)
),
max_views AS (
    SELECT MAX(views) AS max_view_count FROM view_counts
)
SELECT
    vc.view_date,
    vc.views
FROM view_counts vc
JOIN max_views mv ON vc.views = mv.max_view_count;
```

Figure 3: SQL Query 3