Milestone 4

In this milestone, the goal is to improve the performance of the last Query by appropriate precomputations. The query of my kind of cats, calculated the weighted likes for recommending videos to user X. The first query calculates recommendations for a user (user ID 0 in this example) without precomputing any data. It involves several intermediate tables and joins, which can be computationally expensive for large datasets. The second query addresses this issue by introducing a precomputed table, Precomputed_UserInteractions.

```
WITH
-- Table for the videos that are liked or watched by user X
LikedOrWatched_x AS (
 (SELECT video_id
   FROM Users u
     JOIN Likes l
       ON u.user_id = l.user_id
   WHERE u.user_id = 0 )
 UNION
 (SELECT w.video_id
   FROM Users u2
     JOIN Watches w
       ON u2.user_id = w.user_id
   WHERE u2.user_id = 0 )),
-- Table for creating a vector of likes for each user
Vectors AS (
 SELECT
   user_id,
   video_id
 FROM Likes),
-- Table for the like vector of user X
Liked_x AS (
 SELECT
   v.video_id
 FROM
   Vectors v
 WHERE v.user_id = 0),
-- Table for calculating the lc for user X and every other user
lcXandOthers AS (
 SELECT
   vec.user_id,
   LOG(1 + COUNT(*)) AS lc
 FROM
   Vectors vec
   JOIN Liked_x v_x
     ON v_x.video_id = vec.video_id
 GROUP BY vec.user_id
 ORDER BY lc DESC)
```

```
SELECT
  l.video_id,
  SUM(lc.lc) AS Weighted_Like
FROM
  Likes l
  JOIN lcXandOthers lc
    ON lc.user_id = l.user_id
  LEFT JOIN LikedOrWatched_x lw
    ON lw.video_id = l.video_id
WHERE lw.video_id IS NULL
GROUP BY l.video_id
ORDER BY Weighted_Like DESC
LIMIT 10;
```

```
Query complete 00:00:01.559
```

(cost=4011193.21..4011193.24 rows=10 width=12)

Adding indexes

Based on the experiments on milestone 3, I added the following indices to the query for cost and time reduction.

- *Users_user_id*, on user_id for Users table
- *Watches_user_id*, on user_id for Watches table

```
Query complete 00:00:01.464
```

cost=3997981.65..3997981.68 rows=10 width=12

Precomputation

According to the query plan, various options exist for creating a precomputed table. Certain tables, like lcXandOthers, heavily rely on others in the query, particularly for calculating the like count (lc) for user X. After careful consideration, I ultimately chose to precompute the LikedOrWatched_x table but in a modified format to maximize its impact on the results.

Although opting to precompute the LikedOrWatched_x table might appear as the most straightforward decision, it serves as more than just a standalone entity. Instead, it acts as a strategic placeholder for a more comprehensive precomputed table, encompassing all user interactions, whether they involve liked or watched videos. While this table may be substantially larger than a per-user liked table, its advantages are noteworthy:

- Universality: A single precomputed table containing liked/watched videos for all users can be efficiently joined with the Likes table to identify videos a specific user hasn't

interacted with. This eliminates the need for per-user liked video tables and simplifies the logic for any user.
- Efficiency for Filtering: By precomputing the information about videos a user has interacted with, the query can quickly filter those out during recommendation calculations. This pre-filtering step reduces the workload on the main query and potentially improves performance.

In essence, the precomputed table concept goes beyond LikedOrWatched_x and aims for a more comprehensive user interaction table that benefits recommendation calculations for all users.

Precomputed table

```
CREATE TABLE Precomputed_UserInteractions (
  user_id INT,
  video_id INT,
  FOREIGN KEY (user_id) REFERENCES Users(user_id)
);
```

Inserting data

```
INSERT INTO Precomputed_UserInteractions (user_id, video_id)
SELECT user_id, video_id
FROM Likes
UNION
SELECT user_id, video_id
FROM Watches;
```

Main query with precomputation

```
--Table for calculating the lc for user X and every other user
with lcXandOthers AS (
  SELECT
    vec.user_id,
    LOG(1 + COUNT(*)) AS lc
  FROM
    Likes vec
    JOIN
        (SELECT l.video_id
         FROM Likes l
         WHERE l.user_id = 0) as v_x
      ON v_x.video_id = vec.video_id
  GROUP BY vec.user_id
  ORDER BY lc DESC),
-- Table for user interactions (using precomputed table)
UserInteraction AS (SELECT video_id
                                        FROM Precomputed_UserInteractions
                                        WHERE user_id = 0)

SELECT
  l.video_id,
  SUM(lc.lc) AS Weighted_Like
FROM
  Likes l
  JOIN lcXandOthers lc
    ON lc.user_id = l.user_id
LEFT JOIN UserInteraction p_i
    ON l.video_id = p_i.video_id
  WHERE p_i.video_id IS NULL
GROUP BY l.video_id
ORDER BY Weighted_Like DESC
LIMIT 10;
```

Exploring the results

```
Query complete 00:00:00.524
```

cost=40895.14..40565.16 rows=10 width=12

Adding index

```
CREATE INDEX idx_userandvideo_id ON Precomputed_UserInteractions(user_id, video_id);
```

```
Query complete 00:00:00.382
```

cost=40655.13..40655.16 rows=10 width=12

Summary

Adding a precomputed table to query has the following benefits:

- **Reduced Redundancy:** The first query calculates the user's liked videos multiple times using joins in the Vectors and lcXandOthers CTEs. Precomputing user interactions in Precomputed_UserInteractions eliminates this redundancy.
- **Improved Performance**: Calculating the LC (like count) for all users and then filtering for the specific user in lcXandOthers can be inefficient. The second query directly joins

Likes with lcXandOthers (precomputed LC for all users), reducing the need for intermediate calculations.
- **Simplified Logic:** The first query uses separate CTEs for LikedOrWatched_x and user interactions (Vectors and Liked_x). The second query combines user interactions (likes in this case) into a single precomputed table (Precomputed_UserInteractions), simplifying the overall logic.

My kind of Cats- with preferences			
Experiment	Time	Cost	Modifications
0 (no indices)	Query complete 00:00:01.559	cost=4011193.21.. 4011193.24 rows=10 width=12	-
1	Query complete 00:00:01.464	cost=3997981.65.. 3997981.68 rows=10 width=12	Adding indexes to the query: Users_user_id, Watches_user_id
2	Query complete 00:00:00.524	cost=41895.14..40565.36 rows=10 width=12	Using precomputed table: Precomputed_UserInteractions
3	Query complete 00:00:00.382	cost=40655.13..40655.16 rows=10 width=12	*Adding index: idx_userandvideo_id*

Using a Precomputed Table

According to my observation, the percentage **cost** improvement is approximately 99.72% (comparing the 1st and 3rd experiments). This means that the query with precomputation is about **99.72%** more efficient in terms of cost compared to the original query without precomputation.
Furthermore, the percentage **time** improvement is approximately **75.59%**. This means that the query with precomputation is about 75.59% faster in terms of execution time compared to the original query without precomputation. As a result:

- **Faster Execution:** Precomputing allows for faster retrieval when calculating weighted likes for recommendations.
- **Scalability:** As the number of users and videos grows, the precomputed table approach can maintain better performance compared to the original query that dynamically calculates user interactions.
- **Maintainability:** The query with precomputed tables can be easier to understand and maintain due to its simplified structure, because of the precomputed table.