Data Management Systems- Cats' videos

1. Data entry

After importing data into CSV tables, I organized it according to my schema and generated well-structured CSV files. As an illustration, I utilized the following Python code to populate the Videos table.

```
import psycopg2
import csv
import os

# Connect to PostgreSQL database
conn = psycopg2.connect(
    host="localhost",
    database="201Cats",
    user="postgres",
    password="**********"
)
cursor = conn.cursor()

videos_file = '/Users/zeinab/Documents/DSE/SQL/HW/videos_cats.csv'

with open(videos_file, 'r') as f:
    cursor.copy_expert(f"COPY Videos FROM STDIN WITH CSV HEADER;", file=f)

# Commit changes and close the connection
conn.commit()
conn.close()
```

2. Performance Optimization

Data size

Users table: 266397 rows, user_id, user_name (first name + last name), facebook_login
Videos table: 1000000 rows, video_id, video_title
Likes table: 2633266 rows, like_id, user_id, video_id, like_timestamp
Watches table: 1053306 rows, watch_id, user_id, video_id, watch_timestamp
Friends table: 2640282 rows, friendship_id, user_id1, user_id2

For all the questions, I assumed user X is the user with user_id = 0.

1. Overall Likes: The Top-10 cat videos with highest numbers of likes

In this query, 2633266 rows of like_id will be grouped by number of videos (1000000), excluding those that have been liked or watched by a specific user (assumed to be identified by user_id = 0). The goal is to identify and retrieve the top 10 videos based on the number of likes. The final output of the query consists of tuples containing video_id and the corresponding number of likes, providing insights into the popularity of videos within the dataset.

```
-- table for the videos that liked or watched by X
WITH 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)
SELECT
   v.video_id,
   COUNT(l.like_id) as likes_count
FROM
   Videos v
JOIN
   Likes l ON v.video_id = l.video_id
WHERE
   NOT EXISTS (
      SELECT *
      FROM LikedOrWatched_x lw
      WHERE lw.video_id = v.video_id
   )
GROUP BY
   v.video_id, v.video_title
ORDER BY
   likes_count DESC
LIMIT 10;
```

Adding indexes

The key factor is to have indexes on the columns involved in the join conditions, filtering, and grouping. In this case, for the first experiment I considered indexes on the following columns:

1. Index on Users.user_id: Since we are using the user_id column in the Users table for joining and filtering, creating an index on this column can improve the performance.

2. Index for Likes table on user_id and video_id: we are joining on the user_id and video_id columns in the Likes table, creating an index on them can be helpful.

3. Index on Watches.user_id: Similarly, for the Watches table, creating an index on the user_id and video_id column may be useful.

4. Index on Videos.video_id: because of joining on the video_id column in the Videos table, having an index on this column can improve the join performance.

Experiment 1

```
-- Index on Users table
CREATE INDEX idx_users_user_id ON Users(user_id);

-- Index on Likes table
CREATE INDEX idx_likes_user_id ON Likes(user_id);
CREATE INDEX idx_likes_video_id ON Likes(video_id);

-- Index on Watches table
CREATE INDEX idx_watches_user_id ON Watches(user_id);
CREATE INDEX idx_watches_video_id ON Watches(video_id);

-- Index on Videos table
CREATE INDEX idx_videos_video_id ON Videos(video_id);
```

After executing the query with the recommended indexes, the overall cost of running the query witnessed a reduction. Subsequently, I conducted an in-depth examination of the query execution plan using the EXPLAIN statement. Consequently, I identified certain indexes that were not effectively utilized in the execution plan. To enhance the efficiency of the database, I proceeded to drop the indexes that proved to be unnecessary based on the analysis. Following these optimizations, I reran the query, further refining the indexing strategy for improved performance.

Experiment 2

```
-- Index on Users table
CREATE INDEX idx_users_user_id ON Users(user_id);

-- Index on Likes table
CREATE INDEX idx_likes_video_id ON Likes(video_id);

-- Index on Watches table
CREATE INDEX idx_watches_user_id ON Watches(user_id);

-- Index on Videos table
CREATE INDEX idx_videos_video_id ON Videos(video_id);
```

Summary

1- Overall likes			
Experiment	Time	Cost	Indexes
0 (no indices)	00:00:02.896	cost=414170.24..414170.27	-
1	00:00:02.031	cost=268275.88..268275.91	likes_video_id watches_video_id videos_video_id users_user_id likes_user_id watches_user_id
2	00:00:01.969	cost=268275.88..268275.91	Likes_video_id Watches_user_id Users_user_id videos_video_id

- Likes_video_id: The index on Likes(video_id) helps optimize the join operation, as it allows the database engine to quickly locate and retrieve rows from the Likes table based on the video_id values.
- Watches_user_id: It facilitates efficient retrieval of rows from the Watches table where the user_id matches the specified value.
- Users_user_id: It optimizes the join operation by quickly accessing rows from the Users table based on the specified user_id.
- Videos_video_id: This index is used for the join condition v.video_id = l.video_id in the main query when joining the Videos and Likes tables. It enhances the efficiency of the join operation by facilitating quick access to rows in the Videos table based on the video_id.

With the implementation of appropriate indexes, both the cost and execution time of the query have been significantly reduced. **The cost has been reduced by 35% and time by 32%**. This optimization has led to improved efficiency in query processing, resulting in faster and effective execution.

2. Friends' Likes: The Top-10 cat videos with the highest numbers of likes from the friends of X.

In this query, first we separate the user_id of X's friends from a table with 2640282 rows. Then based on like_id of X's friends (from Likes table with 2633266 rows), they will be grouped by number of videos (1000000), excluding those that have been liked or watched by a specific user (assumed to be identified by user_id = 0). The goal is to identify and retrieve the top 10 videos based on the number of likes.

```
-- table for the videos that liked or watched by X
WITH 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),

-- create a table for X's friends by entering the user's name.
x_friends AS (SELECT *
                     FROM Friendships fsh
                                        JOIN Users u
                                        ON u.user_id = fsh.user_id1
              WHERE u.user_id = 0)
SELECT
          v.video_id,
          v.video_title,
          COUNT(l.like_id) as likes_count
FROM  x_friends
            JOIN Likes l
            ON x_friends.user_id2 = l.user_id
            JOIN Videos v ON v.video_id = l.video_id
WHERE
   NOT EXISTS (
      SELECT *
      FROM LikedOrWatched_x lw
      WHERE lw.video_id = v.video_id)
GROUP BY v.video_id
ORDER BY COUNT(l.like_id) DESC
LIMIT 10;
```

Adding indexes

Similar to the previous question, I strategically identified indexes tailored to enhance the performance of the query. The chosen indexes include:

1. Index for Likes table on user_id and video_id: joining the Likes table on the user_id and video_id columns
2. Index for Watches table on user_id and video_id
3. Index on Friendships.user_id1 and user_id2

4. Index on Videos.video_id

Experiment 1

```
-- Index on Users table
CREATE INDEX idx_users_user_id ON Users(user_id);

-- Index on Likes table
CREATE INDEX idx_likes_user_id ON Likes(user_id);
CREATE INDEX idx_likes_video_id ON Likes(video_id);

-- Index on Watches table
CREATE INDEX idx_watches_user_id ON Watches(user_id);
CREATE INDEX idx_watches_video_id ON Watches(video_id);

-- Index on Videos table
CREATE INDEX idx_videos_video_id ON Videos(video_id);

-- Index on Friendships
CREATE INDEX idx_friends_user_id1 ON Friendships(user_id1);
CREATE INDEX idx_friends_user_id2 ON Friendships(user_id2);
```

After observing the query results, the implemented indexes indeed contributed to the efficiency of the execution. However, a subsequent examination using the EXPLAIN statement revealed that some of these indexes were not being effectively utilized in the execution plan. To refine the indexing strategy and eliminate unnecessary overhead, I dropped those indexes that were deemed redundant based on the EXPLAIN analysis.

Experiment 2

```
-- Index on Users table
CREATE INDEX idx_users_user_id ON Users(user_id);

-- Index on Likes table
CREATE INDEX idx_likes_user_id ON Likes(user_id);

-- Index on Watches table
CREATE INDEX idx_watches_user_id ON Watches(user_id);

-- Index on Videos table
CREATE INDEX idx_videos_video_id ON Videos(video_id);

-- Index on Friendships
CREATE INDEX idx_friends_user_id1 ON Friendships(user_id1);
```

In another experiment, I implemented a composite index on two columns of Likes and Watches tables to explore the results.

Experiment 3

```
-- Index on Users table
CREATE INDEX idx_users_user_id ON Users(user_id);

-- Index on Likes table
CREATE INDEX idx_likes_userandvideo_id ON Likes(user_id, video_id);

-- Index on Watches table
CREATE INDEX idx_watches_userandvideo_id ON Watches(user_id, video_id);

-- Index on Videos table
CREATE INDEX idx_videos_video_id ON Videos(video_id);

-- Index on Friendships
CREATE INDEX idx_friends_user_id1 ON Friendships(user_id1);
```

Summary

2- Friends' likes			
Experiment	Time	Cost	Indexes
0 (no indices)	00:00:00.443	cost=48482.80..48482.83	-
1	00:00:00.061	cost=1069.65..1069.68	user_user_id likes_user_id watches_user_id friends_user_id1 Friends_user_id2 likes_video_id watches_video_id videos_video_id
2	00:00:00.140	cost=1069.65..1069.68	user_user_id likes_user_id Watches_user_id videos_video_id Friends_user_id1
3	00:00:00.148	cost=1050.17..1050.20	user_user_id likes_userandvideo_id watches_userandvideo_id videos_video_id Friends_user_id1

In the final indexing strategy for the query:

- User_user_id: This index, likely on the Users table, is intended to optimize the join condition involving the user's ID in various parts of the query.
- Likes_userandvideo_id: This composite index on the Likes table includes user_id and video_id, aiming to enhance the efficiency of joins and filtering related to user likes.
- Watches_userandvideo_id: Similar to the likes_userandvideo_id index, this composite index on the Watches table includes user_id and video_id to optimize joins and filtering for user watches.
- Videos_video_id: This index on the Videos table, likely on the video_id column, is implemented to improve the efficiency of joins involving video information.
- Friends_user_id1: This index, likely on the Friendships table, includes user_id1 and aims to optimize joins involving the user's friendships.

The implementation of these indexes has yielded a remarkable reduction in query execution cost, **achieving a substantial improvement of 97% in the cost of query execution**. This decrease in cost reflects the effectiveness of the indexing strategy in enhancing the efficiency of the query execution.

3. Friends-of-Friends Likes: The Top-10 cat videos with the highest numbers of likes from friends and friends-of-friends

Here first we separate the user_id of X's friends and their friends from a table with 2640282 rows. Then based on like_id of the result (from Likes table with 2633266 rows), they will be grouped by number of videos (1000000), excluding those that have been liked or watched by a specific user (assumed to be identified by user_id = 0). The goal is to identify and retrieve the top 10 videos based on the number of likes.

```
-- table for the videos that liked or watched by X
WITH 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 finding X's friends
x_friends AS (
        SELECT user_id2 AS x_f, user_id1
        FROM Friendships fsh
        JOIN Users u
        ON u.user_id = fsh.user_id1
        WHERE u.user_id = 0),
```

```
-- table for finding friends of X's friends
        friends_of_friends  AS (
        SELECT *
        FROM Friendships fsh
        JOIN x_friends xf ON xf.x_f = fsh.user_id1
        JOIN Likes l
        ON xf.x_f = l.user_id)
SELECT v.video_id,
          v.video_title,
          COUNT (DISTINCT(l.like_id)) as likes_count_3
FROM friends_of_friends ff,
         x_friends, Likes l
JOIN Videos v
ON v.video_id = l.video_id
WHERE
        (l.user_id = ff.user_id2 OR l.user_id = ff.x_f)
        AND ff.user_id2 <> x_friends.user_id1
        AND NOT EXISTS (
      SELECT *
      FROM LikedOrWatched_x lw
      WHERE lw.video_id = v.video_id)
GROUP BY v.video_id
ORDER BY COUNT (DISTINCT(l.like_id)) DESC
LIMIT 10;
```

Adding Indexes

Based on the query structure, the suggested indexes are similar to the last question as following:

Experiment 1

```
-- Index on Users table
CREATE INDEX idx_users_user_id ON Users(user_id);

-- Index on Likes table
CREATE INDEX idx_likes_user_id ON Likes(user_id);
CREATE INDEX idx_likes_video_id ON Likes(video_id);

-- Index on Watches table
CREATE INDEX idx_watches_user_id ON Watches(user_id);
CREATE INDEX idx_watches_video_id ON Watches(video_id);

-- Index on Videos table
CREATE INDEX idx_videos_video_id ON Videos(video_id);

-- Index on Friendships
CREATE INDEX idx_friends_user_id1 ON Friendships(user_id1);
CREATE INDEX idx_friends_user_id2 ON Friendships(user_id2);
```

After implementing the indexes, I observed improvement in query performance. However, a subsequent analysis using the EXPLAIN statement revealed that some of these indexes were not used in the query execution plan. Therefore, I drop those indexes based on the EXPLAIN analysis.

Experiment 2

```
-- Index on Users table
CREATE INDEX idx_users_user_id ON Users(user_id);

-- Index on Likes table
CREATE INDEX idx_likes_user_id ON Likes(user_id);

-- Index on Watches table
CREATE INDEX idx_watches_user_id ON Watches(user_id);

-- Index on Videos table
CREATE INDEX idx_videos_video_id ON Videos(video_id);

-- Index on Friendships
CREATE INDEX idx_friends_user_id1 ON Friendships(user_id1);
```

In another experiment, I have tried two composite indexes on the Like and Watches tables.

Experiment 3

```
-- Index on Users table
CREATE INDEX idx_users_user_id ON Users(user_id);

-- Index on Likes table
CREATE INDEX idx_likes_userandvideo_id ON Likes(user_id, video_id);

-- Index on Watches table
CREATE INDEX idx_watches_userandvideo_id ON Watches(user_id, video_id);

-- Index on Videos table
CREATE INDEX idx_videos_video_id ON Videos(video_id);

-- Index on Friendships
CREATE INDEX idx_friends_user_id1 ON Friendships(user_id1);
```

Summary

3- Friends of friends' likes			
Experiment	Time	Cost	Indexes
0 (no indices)	00:00:00.700	cost=482184.17..4	-

		82184.19	
1	00:00:00.133	cost=375943.39..375943.41	user_user_id likes_user_id watches_user_id friends_user_id1 Friends_user_id2 likes_video_id watches_video_id videos_video_id
2	00:00:00.230	cost=375943.39..375943.41	user_user_id likes_user_id Watches_user_id videos_video_id Friends_user_id1
3	00:00:00.128	cost=375925.18..375925.21	user_user_id likes_userandvideo_id watches_userandvideo_id videos_video_id Friends_user_id1

- The selection of the final indexes for the query, encompassing user_user_id, likes_userandvideo_id, watches_userandvideo_id, videos_video_id, and Friends_user_id1, has proven to be a highly effective optimization strategy.
- The user_user_id index has streamlined the query's ability to efficiently join and filter based on the user's ID.
- The composite index likes_userandvideo_id and watches_userandvideo_id has provided a substantial boost to the efficiency of join operations and filtering related to user likes and watches.
- The videos_video_id index has contributed to the optimization of join conditions involving video information.
- The Friends_user_id1 index has played a crucial role in optimizing join conditions related to the user's friendships.

This holistic indexing approach has demonstrated its effectiveness by not only reducing the cost of execution but also significantly improving the query's response time. **The 22% cost reduction and the 80% reduction in time has been observed.**

It is worth noting that the cost reduction of Experiment 3 was not noticeable.

4. My kind of cats

For this question, we need to find users with mutual like_id with user X from the Likes table with 2633266 rows. Then, with the mutual_friends table we count likes for each video (table with 1000000 rows), excluding those that have been liked or watched by a specific user (assumed to be

identified by user_id = 0). The goal is to identify and retrieve the top 10 videos based on the number of likes.

Revised version:

```
-- table for the videos that liked or watched by X
WITH 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 of videos which user X liked.
x_likes AS (SELECT l.video_id AS x_videos,
                                    l.user_id AS x_id
                                    FROM Users u
                                    JOIN Likes l
                                    ON l.user_id = u.user_id
                                    WHERE u.user_id = 0),
-- table of users who share at least one mutual like with user X.
mutual_users AS (SELECT DISTINCT(l.user_id) AS mutual
                                    FROM Likes l
                                    JOIN x_likes
                                    ON l.video_id = x_likes.x_videos)
SELECT v.video_id,
    v.video_title,
        COUNT (DISTINCT(l.like_id)) as likes_count_4
FROM mutual_users AS mu, Likes l
WHERE l.user_id = mu.mutual
AND NOT EXISTS (
    SELECT *
    FROM LikedOrWatched_x lw
    WHERE lw.video_id = l.video_id)
GROUP BY l.video_id
ORDER BY COUNT(DISTINCT(l.like_id)) DESC
LIMIT 10;
```

Adding Indexes

In this question, the suggested indexes on Likes, Watches, Videos, and Users tables are similar to the previous question, but we do not need Friendship here.

Experiment 1

```
-- Index on Users table
CREATE INDEX idx_users_user_id ON Users(user_id);

-- Index on Likes table
CREATE INDEX idx_likes_user_id ON Likes(user_id);
CREATE INDEX idx_likes_video_id ON Likes(video_id);

-- Index on Watches table
CREATE INDEX idx_watches_user_id ON Watches(user_id);
CREATE INDEX idx_watches_video_id ON Watches(video_id);

-- Index on Videos table
CREATE INDEX idx_videos_video_id ON Videos(video_id);
```

Experiment 2

```
-- Index on Users table
CREATE INDEX idx_users_user_id ON Users(user_id);

-- Index on Likes table
CREATE INDEX idx_likes_user_id ON Likes(user_id);
CREATE INDEX idx_likes_video_id ON Likes(video_id);

-- Index on Watches table
CREATE INDEX idx_watches_user_id ON Watches(user_id);
```

The unused indexes have been dropped and try another run with composite indexes.

Experiment 3

```
-- Index on Users table
CREATE INDEX idx_users_user_id ON Users(user_id);

-- Index on Likes table
CREATE INDEX idx_likes_userandvideo_id ON Likes(user_id, video_id);

-- Index on Watches table
CREATE INDEX idx_watches_user_id ON Watches(user_id);
```

Summary

4- My kind of cats			
Experiment	Time	Cost	Indexes
0 (no indices)	00:00:00.120	cost=54284.10..54284.13	-

1	00:00:00.075	cost=3383.22..3383.25	user_user_id likes_user_id watches_user_id likes_video_id watches_video_id videos_video_id
2	00:00:00.082	cost=3383.22..3383.25	user_user_id Likes_user_id likes_video_id Watches_user_id
3	00:00:00.113	cost=41072.55..41072.57	user_user_id likes_userandvideo_id watches_user_id

- In analyzing this query, I utilized the EXPLAIN statement and observed the involvement of user_id from the Users table, like_id and video_id from the Likes table, and user_id from the Watches table. However, despite incorporating these indexes, the query execution cost remained unchanged, and unexpectedly, the runtime increased.
- In a subsequent attempt, I experimented with a composite index for the Likes table, but contrary to expectations, **both the cost and runtime experienced an increase**.

5. My kind of cats – with preference

For my kind of cats with preference, we need to find the cosine for the likes vector between any user and user X. So, the 2633266 rows of the Likes table and 1000000 rows of the Videos table will be used, excluding those that have been liked or watched by a specific user (assumed to be identified by user_id = 0). The goal is to identify and retrieve the top 10 videos based on the weighted likes.

I improved my code for this question from the last milestone to reduce the running time.

```
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;
```

Adding indexes

Experiment 1

```
-- Index on Users table
CREATE INDEX idx_users_user_id ON Users(user_id);

-- Index on Likes table
CREATE INDEX idx_likes_user_id ON Likes(user_id);
CREATE INDEX idx_likes_video_id ON Likes(video_id);

-- Index on Watches table
```

CREATE INDEX idx_watches_user_id ON Watches(user_id);
CREATE INDEX idx_watches_video_id ON Watches(video_id);

-- Index on Videos table
CREATE INDEX idx_videos_video_id ON Videos(video_id);

Experiment 2

-- Index on Users table
CREATE INDEX idx_users_user_id ON Users(user_id);

-- Index on Watches table
CREATE INDEX idx_watches_user_id ON Watches(user_id);

Summary

5- My kind of cats- with preference			
Experiment	Time	Cost	Indexes
0 (no indices)	00:00:01.827	cost=4011193.21.. 4011193.24	-
1	00:00:01.474	cost=3997981.65.. 3997981.68	User_user_id likes_user_id Likes_video_id watches_user_id watches_video_id videos_video_id
2	00:00:01.339	cost=3997981.65.. 3997981.68	User_user_id watches_user_id

- The selection of the final indexes for the query was user_user_id and watches_user_id has proven to be effective.
- The user_user_id index has streamlined the query's ability to efficiently join and filter based on the user's ID.
- The index watches_user_id has provided a substantial boost to the efficiency of join operations and filtering related to user watches.

This indexing approach has demonstrated its effectiveness by **0.3% cost reduction and the 26% reduction in time.**