Connection:

```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to cs411-nddp-095.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
david_lin4171@cloudshell:~ (cs411-nddp-095)$ gcloud sql connect nddp-095 --user=root --quiet
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2358
Server version: 8.0.31-google (Google)

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use youtube_trending_data
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+--------------------------------+
| Tables_in_youtube_trending_data |
+--------------------------------+
| category                       |
| channel                        |
| country                        |
| manages                        |
| user                           |
| video                          |
| video_stats                    |
+--------------------------------+
7 rows in set (0.00 sec)
```

DDL Commands

```sql
CREATE TABLE `country` (
    `country_code` INT(11) PRIMARY KEY NOT NULL,
    `country` VARCHAR(20) NOT NULL
);
```

```sql
CREATE TABLE `user` (
    `user_id` INT(11) PRIMARY KEY NOT NULL,
    `country_code` INT(11) NOT NULL,
    `username` VARCHAR(50) NOT NULL,
    `password` VARCHAR(50) NOT NULL,
    `email_id` VARCHAR(50) NOT NULL,
    FOREIGN KEY(`country_code`) REFERENCES `country`(`country_code`)
    ON UPDATE CASCADE ON DELETE CASCADE
);
```

```sql
CREATE TABLE `channel` (
    `channel_id` VARCHAR(50) PRIMARY KEY NOT NULL,
    `channel_title` VARCHAR(255) NOT NULL
);
```

```sql
CREATE TABLE `manages` (
    `user_id` INT(11) NOT NULL,
    `channel_id` VARCHAR(50) NOT NULL,
    PRIMARY KEY (`user_id`, `channel_id`),
    FOREIGN KEY (`user_id`) REFERENCES `user` (`user_id`)
    ON UPDATE CASCADE ON DELETE CASCADE,
    FOREIGN KEY (`channel_id`) REFERENCES `channel` (`channel_id`)
    ON UPDATE CASCADE ON DELETE CASCADE
);
```

```sql
CREATE TABLE `video` (
    `video_id` VARCHAR(50) PRIMARY KEY NOT NULL,
    `channel_id` VARCHAR(50) NOT NULL,
    `publishedAt` VARCHAR(50) NOT NULL,
    `title` TEXT NOT NULL,
    `thumbnail_link` VARCHAR(255) NOT NULL,
    `description` TEXT,
    FOREIGN KEY(`channel_id`) REFERENCES `channel`(`channel_id`)
    ON UPDATE CASCADE ON DELETE CASCADE
);
```

```sql
CREATE TABLE `video_stats` (
    `trending_date` VARCHAR(50) NOT NULL,
    `video_id` VARCHAR(50) NOT NULL,
    `view_count` INT(11),
    `likes` INT(11),
    `dislikes` INT(11),
    `comment_count` INT(11),
    `comments_disabled` INT(11),
    `ratings_disabled` INT(11),
    PRIMARY KEY (`trending_date`, `video_id`),
    FOREIGN KEY (`video_id`) REFERENCES `video` (`video_id`)
    ON UPDATE CASCADE ON DELETE CASCADE
);
```

```sql
CREATE TABLE `category` (
    `categoryId` INT(11) NOT NULL,
    `video_id` VARCHAR(50) NOT NULL,
    `tags` TEXT,
    PRIMARY KEY (`categoryId`, `video_id`),
    FOREIGN KEY (`video_id`) REFERENCES `video` (`video_id`)
    ON UPDATE CASCADE ON DELETE CASCADE
);
```

Tables

```
mysql> select count(*) from channel;
+----------+
| count(*) |
+----------+
|     2137 |
+----------+
1 row in set (0.01 sec)

mysql> select count(*) from video;
+----------+
| count(*) |
+----------+
|     3692 |
+----------+
1 row in set (0.01 sec)

mysql> select count(*) from category;
+----------+
| count(*) |
+----------+
|     3692 |
+----------+
1 row in set (0.00 sec)

mysql> select count(*) from video_stats;
+----------+
| count(*) |
+----------+
|    16505 |
+----------+
1 row in set (0.00 sec)
```

ADVANCED QUERY

Advanced query 1: Select the most common categories that appear on Youtube's trending page by selecting the categories that have appeared more than the average number of times each category has appeared on Youtube trending page since the last 20000 trending videos across 5 countries. Only considers videos that have been posted after 10-15-2023 (date can be changed). Output only has 5 rows.

```
SELECT categoryId, SUM(times_trending) AS num_trending
FROM video NATURAL JOIN (SELECT video_id, COUNT(video_id) AS times_trending
                    FROM video_stats natural join video
                    WHERE publishedAt > '2023-10-15 00:00:00'
                    GROUP BY video_id) AS tt NATURAL JOIN category
GROUP BY categoryId
HAVING SUM(times_trending) > (SELECT AVG(b.num_trending)
                        FROM (SELECT SUM(a.times_trending) AS num_trending
                             FROM (SELECT video_id, COUNT(video_id) AS times_trending
                                  FROM video_stats natural join video
                                  WHERE publishedAt > '2023-10-15 00:00:00'
                                  GROUP BY video_id) AS a NATURAL JOIN category
                             GROUP BY categoryId) AS b)
ORDER BY num_trending DESC;
```

```
mysql> SELECT categoryId, SUM(times_trending) AS num_trending
    -> FROM video NATURAL JOIN (SELECT video_id, COUNT(video_id) AS times_trending
    ->                          FROM video_stats natural join video
    ->                          WHERE publishedAt > '2023-10-15 00:00:00'
    ->                          GROUP BY video_id) AS tt NATURAL JOIN category
    -> GROUP BY categoryId
    -> HAVING SUM(times_trending) > (SELECT AVG(b.num_trending)
    ->                               FROM (SELECT SUM(a.times_trending) AS num_trending
    ->                                     FROM (SELECT video_id, COUNT(video_id) AS times_trending
    ->                                           FROM video_stats natural join video
    ->                                           WHERE publishedAt > '2023-10-15 00:00:00'
    ->                                           GROUP BY video_id) AS a NATURAL JOIN category
    ->                                     GROUP BY categoryId) AS b)
    -> ORDER BY num_trending DESC;
+------------+--------------+
| categoryId | num_trending |
+------------+--------------+
|         24 |          979 |
|         17 |          444 |
|         20 |          428 |
|         22 |          324 |
|         10 |          317 |
+------------+--------------+
5 rows in set (0.03 sec)
```

Indexing:

Initial EXPLAIN ANALYZE

```
| -> Sort: num_trending DESC  (actual time=27.530..27.530 rows=5 loops=1)
    -> Filter: (sum(tt.times_trending) > (select #3))  (actual time=27.502..27.509 rows=5 loops=1)
        -> Table scan on <temporary>  (actual time=15.086..15.089 rows=14 loops=1)
            -> Aggregate using temporary table  (actual time=15.084..15.084 rows=14 loops=1)
                -> Nested loop inner join  (cost=2152.20 rows=0) (actual time=9.290..14.377 rows=939 loops=1)
                    -> Nested loop inner join  (cost=1182.49 rows=0) (actual time=9.276..11.452 rows=939 loops=1)
                        -> Table scan on tt  (cost=2.50..2.50 rows=0) (actual time=9.260..9.510 rows=939 loops=1)
                            -> Materialize  (cost=0.00..0.00 rows=0) (actual time=9.259..9.259 rows=939 loops=1)
                                -> Table scan on <temporary>  (actual time=8.856..9.038 rows=939 loops=1)
                                    -> Aggregate using temporary table  (actual time=8.853..8.853 rows=939 loops=1)
                                        -> Nested loop inner join  (cost=1050.44 rows=3877) (actual time=0.130..6.708 rows=3068 loops=1)
                                            -> Filter: (video.publishedAt > '2023-10-15 00:00:00')  (cost=439.57 rows=717) (actual time=0.113..2.987 rows=939 loops=1)
                                                -> Table scan on video  (cost=439.57 rows=2152) (actual time=0.109..2.438 rows=3692 loops=1)
                                            -> Covering index lookup on video_stats using video_id (video_id=video.video_id)  (cost=0.31 rows=5) (actual time=0.003..0.004 rows=3 loops=939)
                        -> Single-row covering index lookup on video using PRIMARY (video_id=tt.video_id)  (cost=0.30 rows=1) (actual time=0.002..0.002 rows=1 loops=939)
                    -> Covering index lookup on category using video_id (video_id=tt.video_id)  (cost=0.25 rows=1) (actual time=0.002..0.003 rows=1 loops=939)
        -> Select #3 (subquery in condition; run only once)
            -> Aggregate: avg(b.num_trending)  (cost=2.50..2.50 rows=1) (actual time=12.366..12.366 rows=1 loops=1)
                -> Table scan on b  (cost=2.50..2.50 rows=0) (actual time=12.356..12.358 rows=14 loops=1)
                    -> Materialize  (cost=0.00..0.00 rows=0) (actual time=12.356..12.356 rows=14 loops=1)
                        -> Table scan on <temporary>  (actual time=12.325..12.327 rows=14 loops=1)
                            -> Aggregate using temporary table  (actual time=12.324..12.324 rows=14 loops=1)
                                -> Nested loop inner join  (cost=972.21 rows=0) (actual time=9.489..11.890 rows=939 loops=1)
                                    -> Table scan on a  (cost=2.50..2.50 rows=0) (actual time=9.470..9.631 rows=939 loops=1)
                                        -> Materialize  (cost=0.00..0.00 rows=0) (actual time=9.469..9.469 rows=939 loops=1)
                                            -> Table scan on <temporary>  (actual time=9.093..9.263 rows=939 loops=1)
                                                -> Aggregate using temporary table  (actual time=9.091..9.091 rows=939 loops=1)
                                                    -> Nested loop inner join  (cost=1050.44 rows=3877) (actual time=0.085..6.745 rows=3068 loops=1)
                                                        -> Filter: (video.publishedAt > '2023-10-15 00:00:00')  (cost=439.57 rows=717) (actual time=0.069..2.871 rows=939 loops=1)
                                                            -> Table scan on video  (cost=439.57 rows=2152) (actual time=0.065..2.190 rows=3692 loops=1)
                                                        -> Covering index lookup on video_stats using video_id (video_id=video.video_id)  (cost=0.31 rows=5) (actual time=0.003..0.004 rows=3 loops=939)
                                    -> Covering index lookup on category using video_id (video_id=a.video_id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=939)
 |
```

```
1 row in set (0.04 sec)
```

Procedures such as "Table scan on temporary" and "Table scan on video_stats" demonstrate how the database must search through the whole table in order to locate the pertinent entries, resulting in increased processing time and cost. The procedures "Aggregate using temporary table" and "Nested loop join" also point to an inefficient execution as they must process more rows, which lengthens the query's execution time.

Explain analyze with only index for category.categoryId

```
| -> Sort: num_trending DESC  (actual time=24.374..24.374 rows=5 loops=1)
    -> Filter: (sum(tt.times_trending) > (select #3))  (actual time=24.346..24.354 rows=5 loops=1)
        -> Table scan on <temporary>  (actual time=13.121..13.125 rows=14 loops=1)
            -> Aggregate using temporary table  (actual time=13.120..13.120 rows=14 loops=1)
                -> Nested loop inner join  (cost=2152.20 rows=0) (actual time=8.524..12.507 rows=939 loops=1)
                    -> Nested loop inner join  (cost=1182.49 rows=0) (actual time=8.512..10.282 rows=939 loops=1)
                        -> Table scan on tt  (cost=2.50..2.50 rows=0) (actual time=8.499..8.698 rows=939 loops=1)
                            -> Materialize  (cost=0.00..0.00 rows=0) (actual time=8.499..8.499 rows=939 loops=1)
                                -> Table scan on <temporary>  (actual time=8.117..8.299 rows=939 loops=1)
                                    -> Aggregate using temporary table  (actual time=8.115..8.115 rows=939 loops=1)
                                        -> Nested loop inner join  (cost=1050.44 rows=3877) (actual time=0.109..6.001 rows=3068 loops=1)
                                            -> Filter: (video.publishedAt > '2023-10-15 00:00:00')  (cost=439.57 rows=717) (actual time=0.089..2.552 rows=939 loops=1)
                                                -> Table scan on video  (cost=439.57 rows=2152) (actual time=0.084..2.028 rows=3692 loops=1)
                                            -> Covering index lookup on video_stats using video_id (video_id=video.video_id)  (cost=0.31 rows=5) (actual time=0.002..0.003 rows=3 loops=939)
                        -> Single-row covering index lookup on video using PRIMARY (video_id=tt.video_id)  (cost=0.30 rows=1) (actual time=0.001..0.001 rows=1 loops=939)
                    -> Covering index lookup on category using video_id (video_id=tt.video_id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=939)
        -> Select #3 (subquery in condition; run only once)
            -> Aggregate: avg(b.num_trending)  (cost=2.50..2.50 rows=1) (actual time=11.174..11.174 rows=1 loops=1)
                -> Table scan on b  (cost=2.50..2.50 rows=0) (actual time=11.164..11.165 rows=14 loops=1)
                    -> Materialize  (cost=0.00..0.00 rows=0) (actual time=11.163..11.163 rows=14 loops=1)
                        -> Table scan on <temporary>  (actual time=11.134..11.136 rows=14 loops=1)
                            -> Aggregate using temporary table  (actual time=11.133..11.133 rows=14 loops=1)
                                -> Nested loop inner join  (cost=972.21 rows=0) (actual time=8.275..10.683 rows=939 loops=1)
                                    -> Table scan on a  (cost=2.50..2.50 rows=0) (actual time=8.260..8.416 rows=939 loops=1)
                                        -> Materialize  (cost=0.00..0.00 rows=0) (actual time=8.260..8.260 rows=939 loops=1)
                                            -> Table scan on <temporary>  (actual time=7.903..8.053 rows=939 loops=1)
                                                -> Aggregate using temporary table  (actual time=7.901..7.901 rows=939 loops=1)
                                                    -> Nested loop inner join  (cost=1050.44 rows=3877) (actual time=0.071..5.870 rows=3068 loops=1)
                                                        -> Filter: (video.publishedAt > '2023-10-15 00:00:00')  (cost=439.57 rows=717) (actual time=0.060..2.394 rows=939 loops=1)
                                                            -> Table scan on video  (cost=439.57 rows=2152) (actual time=0.057..1.873 rows=3692 loops=1)
                                                        -> Covering index lookup on video_stats using video_id (video_id=video.video_id)  (cost=0.31 rows=5) (actual time=0.002..0.003 rows=3 loops=939)
                                    -> Covering index lookup on category using video_id (video_id=a.video_id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=939)
 |
```

```
1 row in set (0.03 sec)
```

categoryId is used in the GROUP BY clause and is essential for aggregating data based on different categories. An index on categoryId can make the search for distinct categories faster and more efficient, improving the overall performance of the grouping and aggregation in the query.

The database is effectively employing the index on categoryId for faster data access, as seen by the "Covering index lookup on category using video_id" action that is seen. In contrast, more generic and laborious techniques like table scans and materialisations are employed in the non-indexed case. It is evident that there is a decrease in the cost and real time of several procedures, such as "Aggregate using temporary table" and "Nested loop inner join". Lower expenses and real times are shown in the indexed version, suggesting a more efficient and quick query execution.However, the difference is not that significant due to lack of variations in data.

Explain analyze with only index for video.publishedAt

```
| -> Sort: num_trending DESC  (actual time=20.990..20.990 rows=5 loops=1)
    -> Filter: (sum(tt.times_trending) > (select #3))  (actual time=20.962..20.970 rows=5 loops=1)
        -> Table scan on <temporary>  (actual time=11.243..11.247 rows=14 loops=1)
            -> Aggregate using temporary table  (actual time=11.241..11.241 rows=14 loops=1)
                -> Nested loop inner join  (cost=2817.19 rows=0) (actual time=6.434..10.638 rows=939 loops=1)
                    -> Nested loop inner join  (cost=1547.51 rows=0) (actual time=6.419..8.210 rows=939 loops=1)
                        -> Table scan on tt  (cost=2.50..2.50 rows=0) (actual time=6.407..6.592 rows=939 loops=1)
                            -> Materialize  (cost=0.00..0.00 rows=0) (actual time=6.406..6.406 rows=939 loops=1)
                                -> Table scan on <temporary>  (actual time=6.054..6.207 rows=939 loops=1)
                                    -> Aggregate using temporary table  (actual time=6.051..6.051 rows=939 loops=1)
                                        -> Nested loop inner join  (cost=1019.91 rows=5075) (actual time=0.036..4.034 rows=3068 loops=1)
                                            -> Filter: (video.publishedAt > '2023-10-15 00:00:00')  (cost=220.20 rows=939) (actual time=0.020..0.476 rows=939 loops=1)
                                                -> Covering index range scan on video using publishedAt_idx over ('2023-10-15 00:00:00' < publishedAt)  (cost=220.20 rows=939) (actual time=0.018..0.318 rows=939 loops=1)
                                            -> Covering index lookup on video_stats using video_id (video_id=video.video_id)  (cost=0.31 rows=5) (actual time=0.003..0.003 rows=3 loops=939)
                        -> Single-row covering index lookup on video using PRIMARY (video_id=tt.video_id)  (cost=0.30 rows=1) (actual time=0.002..0.002 rows=1 loops=939)
                    -> Covering index lookup on category using video_id (video_id=tt.video_id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=939)
        -> Select #3 (subquery in condition; run only once)
            -> Aggregate: avg(b.num_trending)  (cost=2.50..2.50 rows=1) (actual time=9.675..9.675 rows=1 loops=1)
                -> Table scan on b  (cost=2.50..2.50 rows=0) (actual time=9.666..9.667 rows=14 loops=1)
                    -> Materialize  (cost=0.00..0.00 rows=0) (actual time=9.665..9.665 rows=14 loops=1)
                        -> Table scan on <temporary>  (actual time=9.627..9.629 rows=14 loops=1)
                            -> Aggregate using temporary table  (actual time=9.626..9.626 rows=14 loops=1)
                                -> Nested loop inner join  (cost=1272.18 rows=0) (actual time=6.313..9.088 rows=939 loops=1)
                                    -> Table scan on a  (cost=2.50..2.50 rows=0) (actual time=6.294..6.479 rows=939 loops=1)
                                        -> Materialize  (cost=0.00..0.00 rows=0) (actual time=6.292..6.292 rows=939 loops=1)
                                            -> Table scan on <temporary>  (actual time=5.944..6.091 rows=939 loops=1)
                                                -> Aggregate using temporary table  (actual time=5.942..5.942 rows=939 loops=1)
                                                    -> Nested loop inner join  (cost=1019.91 rows=5075) (actual time=0.030..3.997 rows=3068 loops=1)
                                                        -> Filter: (video.publishedAt > '2023-10-15 00:00:00')  (cost=220.20 rows=939) (actual time=0.018..0.487 rows=939 loops=1)
                                                            -> Covering index range scan on video using publishedAt_idx over ('2023-10-15 00:00:00' < publishedAt)  (cost=220.20 rows=939) (actual time=0.016..0.318 rows=939 loops=1)
                                                        -> Covering index lookup on video_stats using video_id (video_id=video.video_id)  (cost=0.31 rows=5) (actual time=0.003..0.003 rows=3 loops=939)
                                    -> Covering index lookup on category using video_id (video_id=a.video_id)  (cost=0.25 rows=1) (actual time=0.002..0.003 rows=1 loops=939)
```

```
1 row in set (0.02 sec)
```

A key property in the WHERE clause that filters movies according to their release date is publishedAt. When working with a big dataset, indexing this property can greatly expedite the filtering process by enabling the database to rapidly discover and get rows that fulfill the WHERE clause's requirement.

Following indexing, the database may effectively use the index to immediately acquire the required data, as seen by the EXPLAIN ANALYZE result, which displays a "Covering index lookup" on video utilizing publishedAt. Compared to a complete table scan, this specialized lookup is more efficient, requiring less time and processing power.
By indexing this property, the database was able to more efficiently locate the relevant films throughout the filtering process, ultimately leading to an optimized query overall.

Explain analyze with both index for video.publishedAt and category.categoryId

```
| -> Sort: num_trending DESC  (actual time=21.391..21.391 rows=5 loops=1)
    -> Filter: (sum(tt.times_trending) > (select #3))  (actual time=21.362..21.370 rows=5 loops=1)
        -> Table scan on <temporary>  (actual time=11.430..11.433 rows=14 loops=1)
            -> Aggregate using temporary table  (actual time=11.428..11.428 rows=14 loops=1)
                -> Nested loop inner join  (cost=2817.19 rows=0) (actual time=6.528..10.811 rows=939 loops=1)
                    -> Nested loop inner join  (cost=1547.51 rows=0) (actual time=6.513..8.405 rows=939 loops=1)
                        -> Table scan on tt  (cost=2.50..2.50 rows=0) (actual time=6.500..6.709 rows=939 loops=1)
                            -> Materialize  (cost=0.00..0.00 rows=0) (actual time=6.498..6.498 rows=939 loops=1)
                                -> Table scan on <temporary>  (actual time=6.109..6.290 rows=939 loops=1)
                                    -> Aggregate using temporary table  (actual time=6.105..6.105 rows=939 loops=1)
                                        -> Nested loop inner join  (cost=1019.91 rows=5075) (actual time=0.035..4.149 rows=3068 loops=1)
                                            -> Filter: (video.publishedAt > '2023-10-15 00:00:00')  (cost=220.20 rows=939) (actual time=0.020..0.483 rows=939 loops=1)
                                                -> Covering index range scan on video using publishedAt_idx over ('2023-10-15 00:00:00' < publishedAt)  (cost=220.20 rows=939) (actual time=0.018..0.322 rows=939 loops=1)
                                            -> Covering index lookup on video_stats using video_id (video_id=video.video_id)  (cost=0.31 rows=5) (actual time=0.003..0.004 rows=3 loops=939)
                        -> Single-row covering index lookup on video using PRIMARY (video_id=tt.video_id)  (cost=0.30 rows=1) (actual time=0.002..0.002 rows=1 loops=939)
                    -> Covering index lookup on category using video_id (video_id=tt.video_id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=939)
        -> Select #3 (subquery in condition; run only once)
            -> Aggregate: avg(b.num_trending)  (cost=2.50..2.50 rows=1) (actual time=9.884..9.884 rows=1 loops=1)
                -> Table scan on b  (cost=2.50..2.50 rows=0) (actual time=9.874..9.875 rows=14 loops=1)
                    -> Materialize  (cost=0.00..0.00 rows=0) (actual time=9.873..9.873 rows=14 loops=1)
                        -> Table scan on <temporary>  (actual time=9.846..9.847 rows=14 loops=1)
                            -> Aggregate using temporary table  (actual time=9.845..9.845 rows=14 loops=1)
                                -> Nested loop inner join  (cost=1272.18 rows=0) (actual time=6.954..9.378 rows=939 loops=1)
                                    -> Table scan on a  (cost=2.50..2.50 rows=0) (actual time=6.938..7.088 rows=939 loops=1)
                                        -> Materialize  (cost=0.00..0.00 rows=0) (actual time=6.937..6.937 rows=939 loops=1)
                                            -> Table scan on <temporary>  (actual time=6.524..6.722 rows=939 loops=1)
                                                -> Aggregate using temporary table  (actual time=6.521..6.521 rows=939 loops=1)
                                                    -> Nested loop inner join  (cost=1019.91 rows=5075) (actual time=0.033..4.413 rows=3068 loops=1)
                                                        -> Filter: (video.publishedAt > '2023-10-15 00:00:00')  (cost=220.20 rows=939) (actual time=0.021..0.514 rows=939 loops=1)
                                                            -> Covering index range scan on video using publishedAt_idx over ('2023-10-15 00:00:00' < publishedAt)  (cost=220.20 rows=939) (actual time=0.018..0.340 rows=939 loops=1)
                                                        -> Covering index lookup on video_stats using video_id (video_id=video.video_id)  (cost=0.31 rows=5) (actual time=0.003..0.004 rows=3 loops=939)
                                    -> Covering index lookup on category using video_id (video_id=a.video_id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=939)
 |
```

```
1 row in set (0.02 sec)
```

A composite index could be useful as publishedAt and categoryId are essential for filtering and organizing the data. When both properties are utilized in the query, a composite index may help to optimize query execution by facilitating quicker data retrieval. It expedites the query's execution by decreasing the amount of rows that must be scanned and processed, which improves the efficiency of the filtering and grouping procedures.

Since each and every other attribute were foreign keys/primary keys or they did not make sense to index, we tried to combine these two and see if the computational cost reduced. However, as we can see from the results, the difference was not much significant. Only some parts of the query reduced runtime but increased the cost too.

Advanced query 2: The purpose of this SQL query is to get the average ratio of video views to likes for each channel, sorted in ascending order, and a list of up to 15 channel titles. In order to prevent division by zero, it filters to only include channels with videos that have at least one like. It also restricts the selection to channels with three or more videos. Further filtering these channels is done by using a HAVING condition, which selects only those with an average view-to-like ratio that is higher than the average ratio of views to likes for all videos (again, just liking videos). The result is a list of channels that not only have a higher engagement rate compared to the average but also have a significant presence on the platform in terms of content volume.

```
SELECT
  c.channel_title,
  AVG(vs.view_count/vs.likes) AS avg_likes_ratio
FROM
  channel c
JOIN
  video v ON c.channel_id = v.channel_id
JOIN
  video_stats vs ON v.video_id = vs.video_id
Where vs.likes > 0 and c.channel_id in (select channel_id
                      from channel natural join video natural join video_stats
                      group by channel_id
                      having count(channel_id) >= 3)
GROUP BY
  c.channel_id
HAVING
  AVG(vs.view_count/vs.likes) > (SELECT AVG(view_count/likes) FROM video_stats
 Where likes > 0)
ORDER BY
  avg_likes_ratio
LIMIT 15;
```

Explain analyze before indexing

```
| -> Limit: 15 row(s)  (actual time=122.140..122.142 rows=15 loops=1)
    -> Sort: avg_likes  (actual time=122.139..122.141 rows=15 loops=1)
        -> Filter: (avg((vs.view_count / vs.likes)) > (select #3))  (actual time=46.218..121.420 rows=527 loops=1)
            -> Stream results  (cost=5427.85 rows=3879) (actual time=34.608..109.672 rows=1792 loops=1)
                -> Group aggregate: avg((vs.view_count / vs.likes)), avg((vs.view_count / vs.likes))  (cost=5427.85 rows=3879) (actual time=34.601..107.451 rows=1792 loops=1)
                    -> Nested loop inner join  (cost=5040.00 rows=3879) (actual time=34.568..96.287 rows=15591 loops=1)
                        -> Nested loop inner join  (cost=967.15 rows=2153) (actual time=34.530..47.648 rows=3368 loops=1)
                            -> Filter: <in_optimizer>(c.channel_id,c.channel_id in (select #2))  (cost=217.45 rows=2137) (actual time=34.504..40.610 rows=1820 loops=1)
                                -> Index scan on c using PRIMARY  (cost=217.45 rows=2137) (actual time=0.106..1.279 rows=2137 loops=1)
                                -> Select #2 (subquery in condition; run only once)
                                    -> Filter: ((c.channel_id = `<materialized_subquery>`.channel_id))  (cost=5128.23..5128.23 rows=1) (actual time=0.018..0.018 rows=1 loops=2138)
                                        -> Limit: 1 row(s)  (cost=5128.13..5128.13 rows=1) (actual time=0.017..0.017 rows=1 loops=2138)
                                            -> Index lookup on <materialized_subquery> using <auto_distinct_key> (channel_id=c.channel_id)  (actual time=0.017..0.017 rows=1 loops=2138)
                                                -> Materialize with deduplication  (cost=5128.13..5128.13 rows=11637) (actual time=34.366..34.366 rows=1820 loops=1)
                                                    -> Filter: (count(`channel`.channel_id) >= 3)  (cost=3964.46 rows=11637) (actual time=0.085..32.603 rows=1820 loops=1)
                                                        -> Group aggregate: count(`channel`.channel_id)  (cost=3964.46 rows=11637) (actual time=0.072..32.196 rows=2136 loops=1)
                                                            -> Nested loop inner join  (cost=2800.79 rows=11637) (actual time=0.053..27.563 rows=16505 loops=1)
                                                                -> Nested loop inner join  (cost=967.15 rows=2153) (actual time=0.041..8.801 rows=3692 loops=1)
                                                                    -> Covering index scan on channel using PRIMARY  (cost=217.45 rows=2137) (actual time=0.020..0.755 rows=2137 loops=1)
                                                                    -> Covering index lookup on video using channel_id (channel_id=`channel`.channel_id)  (cost=0.25 rows=1) (actual time=0.003..0.004 rows=2 loops=2
137)

                                                                -> Covering index lookup on video_stats using video_id (video_id=video.video_id)  (cost=0.31 rows=5) (actual time=0.003..0.005 rows=4 loops=3692)
                            -> Covering index lookup on v using channel_id (channel_id=c.channel_id)  (cost=0.25 rows=1) (actual time=0.003..0.004 rows=2 loops=1820)
                        -> Filter: (vs.likes > 0)  (cost=1.35 rows=2) (actual time=0.012..0.014 rows=5 loops=3368)
                            -> Index lookup on vs using video_id (video_id=v.video_id)  (cost=1.35 rows=5) (actual time=0.011..0.013 rows=5 loops=3368)
            -> Select #3 (subquery in condition; run only once)
                -> Aggregate: avg((video_stats.view_count / video_stats.likes))  (cost=2589.12 rows=1) (actual time=11.203..11.204 rows=1 loops=1)
                    -> Filter: (video_stats.likes > 0)  (cost=1951.95 rows=6372) (actual time=0.038..6.979 rows=16077 loops=1)
                        -> Table scan on video_stats  (cost=1951.95 rows=19117) (actual time=0.037..5.433 rows=16505 loops=1)
 |
+-----------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------+
1 row in set (0.12 sec)
```

The SQL query initially showed a total real execution time of between 122.140 ms and 122.142 ms without any applied indexes. It's clear from looking at the different processes that the query's costs increased at several stages due to the absence of indexes. For example, certain actions added to the overall execution time, including materialisations and nested loops, were quite expensive.

Explain analyze with only index for video_stats.likes

```
| -> Limit: 15 row(s)  (actual time=110.675..110.678 rows=15 loops=1)
    -> Sort: avg_likes  (actual time=110.675..110.676 rows=15 loops=1)
        -> Filter: (avg((vs.view_count / vs.likes)) > (select #3))  (actual time=40.232..110.388 rows=527 loops=1)
            -> Stream results  (cost=5621.80 rows=5818) (actual time=29.754..99.656 rows=1792 loops=1)
                -> Group aggregate: avg((vs.view_count / vs.likes)), avg((vs.view_count / vs.likes))  (cost=5621.80 rows=5818) (actual time=29.747..97.564 rows=1792 loops=1)
                    -> Nested loop inner join  (cost=5040.00 rows=5818) (actual time=29.714..86.587 rows=15591 loops=1)
                        -> Nested loop inner join  (cost=967.15 rows=2153) (actual time=29.682..41.575 rows=3368 loops=1)
                            -> Filter: <in_optimizer>(c.channel_id,c.channel_id in (select #2))  (cost=217.45 rows=2137) (actual time=29.670..35.106 rows=1820 loops=1)
                                -> Index scan on c using PRIMARY  (cost=217.45 rows=2137) (actual time=0.049..1.027 rows=2137 loops=1)
                                -> Select #2 (subquery in condition; run only once)
                                    -> Filter: ((c.channel_id = `<materialized_subquery>`.channel_id))  (cost=5128.23..5128.23 rows=1) (actual time=0.015..0.015 rows=1 loops=2138)
                                        -> Limit: 1 row(s)  (cost=5128.13..5128.13 rows=1) (actual time=0.015..0.015 rows=1 loops=2138)
                                            -> Index lookup on <materialized_subquery> using <auto_distinct_key> (channel_id=c.channel_id)  (actual time=0.015..0.015 rows=1 loops=2138)
                                                -> Materialize with deduplication  (cost=5128.13..5128.13 rows=11637) (actual time=29.592..29.592 rows=1820 loops=1)
                                                    -> Filter: (count(`channel`.channel_id) >= 3)  (cost=3964.46 rows=11637) (actual time=0.073..28.303 rows=1820 loops=1)
                                                        -> Group aggregate: count(`channel`.channel_id)  (cost=3964.46 rows=11637) (actual time=0.061..27.959 rows=2136 loops=1)
                                                            -> Nested loop inner join  (cost=2800.79 rows=11637) (actual time=0.044..23.658 rows=16505 loops=1)
                                                                -> Nested loop inner join  (cost=967.15 rows=2153) (actual time=0.036..7.479 rows=3692 loops=1)
                                                                    -> Covering index scan on channel using PRIMARY  (cost=217.45 rows=2137) (actual time=0.019..0.637 rows=2137 loops=1)
                                                                    -> Covering index lookup on video using channel_id (channel_id=`channel`.channel_id)  (cost=0.25 rows=1) (actual time=0.002..0.003 rows=2 loops=2
137)

                                                                -> Covering index lookup on video_stats using video_id (video_id=video.video_id)  (cost=0.31 rows=5) (actual time=0.003..0.004 rows=4 loops=3692)
                            -> Covering index lookup on v using channel_id (channel_id=c.channel_id)  (cost=0.25 rows=1) (actual time=0.002..0.003 rows=2 loops=1820)
                        -> Filter: (vs.likes > 0)  (cost=1.35 rows=3) (actual time=0.011..0.013 rows=5 loops=3368)
                            -> Index lookup on vs using video_id (video_id=v.video_id)  (cost=1.35 rows=5) (actual time=0.011..0.012 rows=5 loops=3368)
            -> Select #3 (subquery in condition; run only once)
                -> Aggregate: avg((video_stats.view_count / video_stats.likes))  (cost=2907.75 rows=1) (actual time=10.166..10.167 rows=1 loops=1)
                    -> Filter: (video_stats.likes > 0)  (cost=1951.95 rows=9558) (actual time=0.032..6.212 rows=16077 loops=1)
                        -> Table scan on video_stats  (cost=1951.95 rows=19117) (actual time=0.031..4.683 rows=16505 loops=1)
 |
+-----------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------+
1 row in set (0.12 sec)
```

In order to rapidly compute the average likes ratio per channel, the database engine must be able to easily retrieve and filter the video statistics depending on the amount of likes. This is ensured by using the index video_stats.likes in the WHERE clause of the subquery. This is especially crucial when working with huge datasets, since non-indexed searches may cause serious problems with performance.

We would select this index because the query execution time was improved by adding an index to video_stats.likes. The positive effect of the index was evident when the total actual time was lowered to between 110.675 and 110.676 milliseconds. This index reduced the amount of rows processed sequentially, which in turn decreased the overall cost to around 5810. It did this by optimizing the database's capacity to identify certain rows. Because it makes it simple for the operating system to cache a large number of indexes into memory for quicker access and for the file system to read a large number of records at once rather than reading them from disc, this index contributed to the speed of the result.

Explain analyze with only index for video_stats.view_count

```
| -> Limit: 15 row(s)  (actual time=110.410..110.413 rows=15 loops=1)
    -> Sort: avg_likes_ratio  (actual time=110.409..110.412 rows=15 loops=1)
        -> Filter: (avg((vs.view_count / vs.likes)) > (select #3))  (actual time=39.209..110.095 rows=527 loops=1)
            -> Stream results  (cost=5427.85 rows=3879) (actual time=28.903..99.638 rows=1792 loops=1)
                -> Group aggregate: avg((vs.view_count / vs.likes)), avg((vs.view_count / vs.likes))  (cost=5427.85 rows=3879) (actual time=28.896..97.505 rows=1792 loops=1)
                    -> Nested loop inner join  (cost=5040.00 rows=3879) (actual time=28.866..86.451 rows=15591 loops=1)
                        -> Nested loop inner join  (cost=967.15 rows=2153) (actual time=28.833..40.599 rows=3368 loops=1)
                            -> Filter: <in_optimizer>(c.channel_id,c.channel_id in (select #2))  (cost=217.45 rows=2137) (actual time=28.815..34.283 rows=1820 loops=1)
                                -> Index scan on c using PRIMARY  (cost=217.45 rows=2137) (actual time=0.064..1.071 rows=2137 loops=1)
                                -> Select #2 (subquery in condition; run only once)
                                    -> Filter: ((c.channel_id = `<materialized_subquery>`.channel_id))  (cost=5128.23..5128.23 rows=1) (actual time=0.015..0.015 rows=1 loops=2138)
                                        -> Limit: 1 row(s)  (cost=5128.13..5128.13 rows=1) (actual time=0.015..0.015 rows=1 loops=2138)
                                            -> Index lookup on <materialized_subquery> using <auto_distinct_key> (channel_id=c.channel_id)  (actual time=0.014..0.014 rows=1 loops=2138)
                                                -> Materialize with deduplication  (cost=5128.13..5128.13 rows=11637) (actual time=28.719..28.719 rows=1820 loops=1)
                                                    -> Filter: (count(`channel`.channel_id) >= 3)  (cost=3964.46 rows=11637) (actual time=0.125..27.526 rows=1820 loops=1)
                                                        -> Group aggregate: count(`channel`.channel_id)  (cost=3964.46 rows=11637) (actual time=0.112..27.212 rows=2136 loops=1)
                                                            -> Nested loop inner join  (cost=2800.79 rows=11637) (actual time=0.094..23.123 rows=16505 loops=1)
                                                                -> Nested loop inner join  (cost=967.15 rows=2153) (actual time=0.084..7.444 rows=3692 loops=1)
                                                                    -> Covering index scan on channel using PRIMARY  (cost=217.45 rows=2137) (actual time=0.018..0.606 rows=2137 loops=1)
                                                                    -> Covering index lookup on video using channel_id (channel_id=`channel`.channel_id)  (cost=0.25 rows=1) (actual time=0.002..0.003 rows=2 loops=2
137)
                                                                -> Covering index lookup on video_stats using video_id (video_id=video.video_id)  (cost=0.31 rows=5) (actual time=0.003..0.004 rows=4 loops=3692)
                            -> Covering index lookup on v using channel_id (channel_id=c.channel_id)  (cost=0.25 rows=1) (actual time=0.002..0.003 rows=2 loops=1820)
                        -> Filter: (vs.likes > 0)  (cost=1.35 rows=2) (actual time=0.011..0.013 rows=5 loops=3368)
                            -> Index lookup on vs using video_id (video_id=v.video_id)  (cost=1.35 rows=5) (actual time=0.011..0.013 rows=5 loops=3368)
            -> Select #3 (subquery in condition; run only once)
                -> Aggregate: avg((video_stats.view_count / video_stats.likes))  (cost=2589.12 rows=1) (actual time=9.962..9.962 rows=1 loops=1)
                    -> Filter: (video_stats.likes > 0)  (cost=1951.95 rows=6372) (actual time=0.169..6.216 rows=16077 loops=1)
                        -> Table scan on video_stats  (cost=1951.95 rows=19117) (actual time=0.168..4.807 rows=16505 loops=1)
 |
+------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------+
1 row in set (0.11 sec)
```

When working with big datasets, this is especially helpful because view_count is frequently used as a filter or sorting criterion. The index would be especially useful in the subquery that computes the average view count to likes ratio and in the ORDER BY clause that arranges the results according to the view count-derived avg_likes_ratio. Reducing query execution times and enhancing database operations' overall performance depend heavily on effective indexing.

Indexing has improved speed, which is especially noticeable in some query execution phases. For example, after indexing, the "Nested loop join" operation performs more efficiently and shows a discernible decrease in real time. The operation "Index lookup on vs using video_id" has also greatly profited from indexing; it has gone from a "Table scan on video_stats" to a "Index lookup," resulting in a notable reduction in both real time and cost.

In particular, indexing video_stats.view_count increased the query's efficiency by speeding up the time it takes to get rows based on view_count, which is essential to the query's functionality.

It made it possible to locate required rows more quickly, which improved the flow and efficiency of the query execution—especially in cases where video_stats.view_count plays a crucial role. But if we look at overall execution time, the difference is not sufficient.

Explain analyze with index for video_stats.view_count and video_stats.likes

```
| -> Limit: 15 row(s)  (actual time=118.421..118.424 rows=15 loops=1)
    -> Sort: avg_likes_ratio  (actual time=118.421..118.422 rows=15 loops=1)
        -> Filter: (avg((vs.view_count / vs.likes)) > (select #3))  (actual time=39.184..117.864 rows=527 loops=1)
            -> Stream results  (cost=5621.80 rows=5818) (actual time=28.533..106.927 rows=1792 loops=1)
                -> Group aggregate: avg((vs.view_count / vs.likes)), avg((vs.view_count / vs.likes))  (cost=5621.80 rows=5818) (actual time=28.518..104.514 rows=1792 loops=1)
                    -> Nested loop inner join  (cost=5040.00 rows=5818) (actual time=28.482..92.639 rows=15591 loops=1)
                        -> Nested loop inner join  (cost=967.15 rows=2153) (actual time=28.451..42.625 rows=3368 loops=1)
                            -> Filter: <in_optimizer>(c.channel_id,c.channel_id in (select #2))  (cost=217.45 rows=2137) (actual time=28.439..35.273 rows=1820 loops=1)
                                -> Index scan on c using PRIMARY  (cost=217.45 rows=2137) (actual time=0.049..1.292 rows=2137 loops=1)
                                -> Select #2 (subquery in condition; run only once)
                                    -> Filter: ((c.channel_id = `<materialized_subquery>`.channel_id))  (cost=5128.23..5128.23 rows=1) (actual time=0.015..0.015 rows=1 loops=2138)
                                        -> Limit: 1 row(s)  (cost=5128.13..5128.13 rows=1) (actual time=0.015..0.015 rows=1 loops=2138)
                                            -> Index lookup on <materialized_subquery> using <auto_distinct_key> (channel_id=c.channel_id)  (actual time=0.014..0.014 rows=1 loops=2138)
                                                -> Materialize with deduplication  (cost=5128.13..5128.13 rows=11637) (actual time=28.360..28.360 rows=1820 loops=1)
                                                    -> Filter: (count(`channel`.channel_id) >= 3)  (cost=3964.46 rows=11637) (actual time=0.070..27.183 rows=1820 loops=1)
                                                        -> Group aggregate: count(`channel`.channel_id)  (cost=3964.46 rows=11637) (actual time=0.059..26.862 rows=2136 loops=1)
                                                            -> Nested loop inner join  (cost=2800.79 rows=11637) (actual time=0.043..22.758 rows=16505 loops=1)
                                                                -> Nested loop inner join  (cost=967.15 rows=2153) (actual time=0.037..7.207 rows=3692 loops=1)
                                                                    -> Covering index scan on channel using PRIMARY  (cost=217.45 rows=2137) (actual time=0.020..0.599 rows=2137 loops=1)
                                                                    -> Covering index lookup on video using channel_id (channel_id=`channel`.channel_id)  (cost=0.25 rows=1) (actual time=0.002..0.003 rows=2 loops=2
137)

                                                                -> Covering index lookup on video_stats using video_id (video_id=video.video_id)  (cost=0.31 rows=5) (actual time=0.003..0.004 rows=4 loops=3692)
                            -> Covering index lookup on v using channel_id (channel_id=c.channel_id)  (cost=0.25 rows=1) (actual time=0.003..0.004 rows=2 loops=1820)
                        -> Filter: (vs.likes > 0)  (cost=1.35 rows=3) (actual time=0.012..0.014 rows=5 loops=3368)
                            -> Index lookup on vs using video_id (video_id=v.video_id)  (cost=1.35 rows=5) (actual time=0.012..0.014 rows=5 loops=3368)
            -> Select #3 (subquery in condition; run only once)
                -> Aggregate: avg((video_stats.view_count / video_stats.likes))  (cost=2907.75 rows=1) (actual time=10.342..10.342 rows=1 loops=1)
                    -> Filter: (video_stats.likes > 0)  (cost=1951.95 rows=9558) (actual time=0.034..6.392 rows=16077 loops=1)
                        -> Table scan on video_stats  (cost=1951.95 rows=19117) (actual time=0.033..4.936 rows=16505 loops=1)
 |
+------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.12 sec)
```

Indexing on video_stats.view_count and video_stats.likes has greatly improved the efficiency of the SQL query. After indexing, functions like "Index lookup on vs using video_id" were more effective. This was a change from a "Table scan on video_stats," which was a faster and more accurate way to get data, which reduced the query's overall execution time.

Furthermore, the "Nested loop join" procedure demonstrated improved efficiency as well as indexing-induced processing time optimization. This optimisation makes it possible to retrieve entries from the video_stats table more quickly based on view_count and likes, which is necessary for carrying out computations and filtering inside the query. As a result, adding indexes has improved overall speed, improved the execution strategy, and reduced needless computational costs.