STAGE 3

Note: We also fixed errors in STAGE 1 and STAGE 2

Implemented the database tables locally or on GCP

Below is a screenshot of of our five database tables implemented on GCP:

```
evansimon2001@cloudshell:~ (cs411-403119)$ gcloud sql connect cs411-stage3 --user=root
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 20300
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> show tables
    -> ;
ERROR 1046 (3D000): No database selected
mysql> use youtubeData
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_youtubeData |
+-----------------------+
| Category              |
| Region                |
| Statistics            |
| Video                 |
| YouTuber              |
+-----------------------+
5 rows in set (0.00 sec)
```

Provided the DDL commands for your tables

```
CREATE TABLE YouTuber (
    channelId VARCHAR(255) PRIMARY KEY,
    channelTitle VARCHAR(255)
```

```
);
```

```
CREATE TABLE Region (
    regionId INT PRIMARY KEY,
    regionName VARCHAR(255)
);
```

```
CREATE TABLE Category (
    categoryId INT PRIMARY KEY,
    categoryName VARCHAR(255)
);
```

```
CREATE TABLE Video (
    videoId VARCHAR(255) PRIMARY KEY,
    title VARCHAR(255),
    thumbnailLink VARCHAR(255),
    commentsDisabled BOOLEAN,
    ratingsDisabled BOOLEAN,
    description TEXT,
    channelId VARCHAR(255),
    regionId INT,
    categoryId INT,
    FOREIGN KEY (channelId) REFERENCES YouTuber(channelId) ON DELETE
CASCADE,
    FOREIGN KEY (regionId) REFERENCES Region(regionId) ON DELETE
CASCADE,
    FOREIGN KEY (categoryId) REFERENCES Category(categoryId) ON
DELETE CASCADE
);
```

```
CREATE TABLE Statistics (
    statisticId VARCHAR(255) PRIMARY KEY,
    publishedAt VARCHAR(255),
    trendingDate VARCHAR(255),
    viewCount INT,
    commentCount INT,
```

```
    likes INT,
    dislikes INT,
    videoId VARCHAR(255),
    FOREIGN KEY (videoId) REFERENCES Video(videoId) ON DELETE CASCADE
);
```

```
INSERT INTO Region (regionId, regionName) VALUES
    (1, 'USA'),
    (2, 'India'),
    (3, 'Great Britain'),
    (4, 'Germany'),
    (5, 'Canada'),
    (6, 'France'),
    (7, 'Russia'),
    (8, 'Brazil'),
    (9, 'Mexico'),
    (10, 'South Korea'),
    (11, 'Japan');
```

Two advanced queries

Advanced query 1:

```
SELECT  y.channelTitle, COUNT(v.video_id) AS number_of_trending_videos
FROM YouTuber y
JOIN Video v ON y.channelId = v.channelId
GROUP BY y.channelTitle
ORDER BY number_of_trending_videos DESC
LIMIT 15;
```

Screenshots with the top 15 rows:

channelTitle	number_of_trending_videos
NBA	377
NFL	358
NBC Sports	150
Champions League on CBS Sports	134
ESPN	119
SSSniperWolf	116
SpaceX	101
Saturday Night Live	96
Big Hit Labels	96
First We Feast	89
Skip and Shannon: UNDISPUTED	88
UFC - Ultimate Fighting Champio...	87
Nintendo	87
NBA on TNT	86
The Game Theorists	86

ADVANCED query 2:

```
SELECT
    y.channelTitle,
    AVG(s.view_count) AS avg_views
FROM Video v
JOIN YouTuber y ON v.channelId = y.channelId
JOIN Statistics s ON v.video_id = s.video_id
WHERE s.trending_date = (
    SELECT MAX(trending_date)
    FROM Statistics s2
    WHERE s2.video_id = s.video_id
)
GROUP BY y.channelTitle
HAVING avg_views > (SELECT AVG(view_count) FROM Statistics)
ORDER BY avg_views DESC
LIMIT 15;
```

Screenshots with the top 15 rows:

channelTitle	avg_views
Cardi B	53144629.0000
BLACKPINK	45553354.0000
Zee Music Company	44649281.0000
Big Hit Labels	43952838.5556
DJKhaledVEVO	43394819.0000
The Pixel Kingdom	41005385.5000
J97	37422074.0000
HarryStylesVEVO	32844931.0000
KQ ENTERTAINMENT	30059975.0000
Bad Bunny	27669680.3333
CamiloVEVO	26515286.0000
Ozuna	26229136.5000
DrakeVEVO	23829361.0000
MrBeast	23816410.8462
Apple	23513581.8333

Inserted at least 1000 rows in the tables

Below is a screenshot of each count query:

```
SELECT COUNT(*)
FROM Video
```

COUNT(*)
20859

```
SELECT COUNT(*)
FROM Statistic
```

COUNT(*)
20859

```
SELECT COUNT(*)
FROM YouTuber
```

```
mysql> SELECT COUNT(*) FROM YouTuber;
+----------+
| COUNT(*) |
+----------+
|     1709 |
+----------+
```

We cannot have 1000 categories. We only have the categories listed below:

```
SELECT *
FROM Category
```

```
mysql> SELECT * FROM Category;
+------------+----------------------+
| categoryId | category_name        |
+------------+----------------------+
|          1 | Film & Animation     |
|          2 | Autos & Vehicles     |
|         10 | Music                |
|         15 | Pets & Animals       |
|         17 | Sports               |
|         18 | Short Movies         |
|         19 | Travel & Events      |
|         20 | Gaming               |
|         21 | Videoblogging        |
|         22 | People & Blogs       |
|         23 | Comedy               |
|         24 | Entertainment        |
|         25 | News & Politics      |
|         26 | Howto & Style        |
|         27 | Education            |
|         28 | Science & Technology |
|         29 | Nonprofits & Activism |
|         30 | Movies               |
|         31 | Anime/Animation      |
|         32 | Action/Adventure     |
|         33 | Classics             |
|         34 | Comedy               |
|         35 | Documentary          |
|         36 | Drama                |
|         37 | Family               |
|         38 | Foreign              |
+------------+----------------------+
26 rows in set (0.00 sec)
```

We cannot have 1000 regions. We only have the regions listed below:

```
SELECT *
FROM Category
```

region_id	region_name
1	United States
2	India
3	Great Britain
4	Germany
5	Canada
6	France
7	Russia
8	Brazil
9	Mexico
10	South Korea
11	Japan

Indexing designs & reports

0. No Index

```
| -> Filter: (avg_views > (select #3))  (actual time=204.279..205.104 rows=345 loops=1)
    -> Table scan on <temporary>  (actual time=197.202..197.627 rows=1701 loops=1)
        -> Aggregate using temporary table  (actual time=197.201..197.201 rows=1701 loops=1)
            -> Nested loop inner join  (cost=15994.65 rows=19923) (actual time=0.150..190.374 rows=3561 loops=1)
                -> Nested loop inner join  (cost=9021.60 rows=19923) (actual time=0.140..182.615 rows=3561 loops=1)
                    -> Filter: ((s.trending_date = (select #2)) and (s.video_id is not null))  (cost=2048.55 rows=19923) (actual time=0.128..166.487 rows=3561 loops=1)
                        -> Table scan on s  (cost=2048.55 rows=19923) (actual time=0.048..8.402 rows=17812 loops=1)
                        -> Select #2 (subquery in condition; dependent)
                            -> Aggregate: max(s2.trending_date)  (cost=3.24 rows=1) (actual time=0.008..0.008 rows=1 loops=17812)
                                -> Covering index lookup on s2 using new_index_name (video_id=s.video_id)  (cost=2.68 rows=6) (actual time=0.004..0.006 rows=8 loops=17812)
                    -> Filter: (v.channelId is not null)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=3561)
                        -> Single-row index lookup on v using PRIMARY (video_id=s.video_id)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=3561)
                -> Single-row index lookup on y using PRIMARY (channelId=v.channelId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=3561)
    -> Select #3 (subquery in condition; run only once)
        -> Aggregate: avg(Statistics.view_count)  (cost=4040.85 rows=1) (actual time=7.037..7.037 rows=1 loops=1)
            -> Table scan on Statistics  (cost=2048.55 rows=19923) (actual time=0.034..5.248 rows=17812 loops=1)
 |
```

| -> Filter: (avg_views > (select #3)) (actual time=204.279..205.104 rows=345 loops=1)
   -> Table scan on <temporary> (actual time=197.202..197.627 rows=1701 loops=1)
      -> Aggregate using temporary table (actual time=197.201..197.201 rows=1701 loops=1)
         -> Nested loop inner join (cost=15994.65 rows=19923) (actual time=0.150..190.374 rows=3561 loops=1)
            -> Nested loop inner join (cost=9021.60 rows=19923) (actual time=0.140..182.615 rows=3561 loops=1)
               -> Filter: ((s.trending_date = (select #2)) and (s.video_id is not null)) (cost=2048.55 rows=19923) (actual time=0.128..166.487 rows=3561 loops=1)
                  -> Table scan on s (cost=2048.55 rows=19923) (actual time=0.048..8.402 rows=17812 loops=1)
                  -> Select #2 (subquery in condition; dependent)

-> Aggregate: max(s2.trending_date) (cost=3.24 rows=1) (actual time=0.008..0.008 rows=1 loops=17812)
-> Covering index lookup on s2 using new_index_name (video_id=s.video_id) (cost=2.68 rows=6) (actual time=0.004..0.006 rows=8 loops=17812)
-> Filter: (v.channelId is not null) (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=3561)
-> Single-row index lookup on v using PRIMARY (video_id=s.video_id) (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=3561)
-> Single-row index lookup on y using PRIMARY (channelId=v.channelId) (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=3561)
-> Select #3 (subquery in condition; run only once)
-> Aggregate: avg(Statistics.view_count) (cost=4040.85 rows=1) (actual time=7.037..7.037 rows=1 loops=1)
-> Table scan on Statistics (cost=2048.55 rows=19923) (actual time=0.034..5.248 rows=17812 loops=1)
|

1. CREATE INDEX idx_stats_video_trending ON Statistics (video_id, trending_date);

```
| -> Filter: (avg_views > (select #3))  (actual time=179.604..180.369 rows=345 loops=1)
    -> Table scan on <temporary>  (actual time=173.028..173.418 rows=1701 loops=1)
        -> Aggregate using temporary table  (actual time=173.027..173.027 rows=1701 loops=1)
            -> Nested loop inner join  (cost=15994.65 rows=19923) (actual time=0.150..167.237 rows=3561 loops=1)
                -> Nested loop inner join  (cost=9021.60 rows=19923) (actual time=0.143..160.428 rows=3561 loops=1)
                    -> Filter: ((s.trending_date = (select #2)) and (s.video_id is not null))  (cost=2048.55 rows=19923) (actual time=0.133..146.443 rows=3561 loops=1)
                        -> Table scan on s  (cost=2048.55 rows=19923) (actual time=0.064..7.949 rows=17812 loops=1)
                        -> Select #2 (subquery in condition; dependent)
                            -> Aggregate: max(s2.trending_date)  (cost=3.24 rows=1) (actual time=0.007..0.007 rows=1 loops=17812)
                                -> Covering index lookup on s2 using new_index_name (video_id=s.video_id)  (cost=2.68 rows=6) (actual time=0.003..0.005 rows=8 loops=17812)
                    -> Filter: (v.channelId is not null)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=3561)
                        -> Single-row index lookup on v using PRIMARY (video_id=s.video_id)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=3561)
                -> Single-row index lookup on y using PRIMARY (channelId=v.channelId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=3561)
    -> Select #3 (subquery in condition; run only once)
        -> Aggregate: avg(Statistics.view_count)  (cost=4040.85 rows=1) (actual time=6.525..6.526 rows=1 loops=1)
            -> Table scan on Statistics  (cost=2048.55 rows=19923) (actual time=0.029..4.782 rows=17812 loops=1)
 |
```

| -> Filter: (avg_views > (select #3)) (actual time=179.604..180.369 rows=345 loops=1)
-> Table scan on <temporary> (actual time=173.028..173.418 rows=1701 loops=1)
-> Aggregate using temporary table (actual time=173.027..173.027 rows=1701 loops=1)
-> Nested loop inner join (cost=15994.65 rows=19923) (actual time=0.150..167.237 rows=3561 loops=1)
-> Nested loop inner join (cost=9021.60 rows=19923) (actual time=0.143..160.428 rows=3561 loops=1)
-> Filter: ((s.trending_date = (select #2)) and (s.video_id is not null)) (cost=2048.55 rows=19923) (actual time=0.133..146.443 rows=3561 loops=1)
-> Table scan on s (cost=2048.55 rows=19923) (actual time=0.064..7.949 rows=17812 loops=1)
-> Select #2 (subquery in condition; dependent)
-> Aggregate: max(s2.trending_date) (cost=3.24 rows=1) (actual time=0.007..0.007 rows=1 loops=17812)
-> Covering index lookup on s2 using new_index_name (video_id=s.video_id) (cost=2.68 rows=6) (actual time=0.003..0.005 rows=8 loops=17812)

-> Filter: (v.channelId is not null) (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=3561)
-> Single-row index lookup on v using PRIMARY (video_id=s.video_id) (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=3561)
-> Single-row index lookup on y using PRIMARY (channelId=v.channelId) (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=3561)
-> Select #3 (subquery in condition; run only once)
-> Aggregate: avg(Statistics.view_count) (cost=4040.85 rows=1) (actual time=6.525..6.526 rows=1 loops=1)
-> Table scan on Statistics (cost=2048.55 rows=19923) (actual time=0.029..4.782 rows=17812 loops=1)
|

This index includes video_id and trending_date, which are likely to be used together in queries that analyze trends over time for a specific video. For example, a query might want to retrieve the trend of views for a video over a period. Using this index, the database can quickly filter rows by video_id and then immediately have the rows in order of trending_date, which can make retrieval much faster.

This index appears to provide relatively fast query performance, with an actual time of 179.604 to 180.369. The speed is attributed to the index being created on the columns video_id and trending_date, which are mostly used in the WHERE clause of your query, particularly in the subquery's correlated condition. The index improves filtering based on the conditions in your WHERE clause and the correlated subquery, resulting in faster access to the required rows in the Statistics table.

Then we drop the index with

```
DROP INDEX idx_stats_video_trending ON Statistics;
```

2. CREATE INDEX idx_stats_view_count ON Statistics (view_count);

```
| -> Filter: (avg_views > (select #3))  (actual time=179.604..180.369 rows=345 loops=1)
    -> Table scan on <temporary>  (actual time=173.028..173.418 rows=1701 loops=1)
        -> Aggregate using temporary table  (actual time=173.027..173.027 rows=1701 loops=1)
            -> Nested loop inner join  (cost=15994.65 rows=19923) (actual time=0.150..167.237 rows=3561 loops=1)
                -> Nested loop inner join  (cost=9021.60 rows=19923) (actual time=0.143..160.428 rows=3561 loops=1)
                    -> Filter: ((s.trending_date = (select #2)) and (s.video_id is not null))  (cost=2048.55 rows=19923) (actual time=0.133..146.443 rows=3561 loops=1)
                        -> Table scan on s  (cost=2048.55 rows=19923) (actual time=0.064..7.949 rows=17812 loops=1)
                        -> Select #2 (subquery in condition; dependent)
                            -> Aggregate: max(s2.trending_date)  (cost=3.24 rows=1) (actual time=0.007..0.007 rows=1 loops=17812)
                                -> Covering index lookup on s2 using new_index_name (video_id=s.video_id)  (cost=2.68 rows=6) (actual time=0.003..0.005 rows=8 loops=17812)
                    -> Filter: (v.channelId is not null)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=3561)
                        -> Single-row index lookup on v using PRIMARY (video_id=s.video_id)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=3561)
                -> Single-row index lookup on y using PRIMARY (channelId=v.channelId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=3561)
    -> Select #3 (subquery in condition; run only once)
        -> Aggregate: avg(Statistics.view_count)  (cost=4040.85 rows=1) (actual time=6.525..6.526 rows=1 loops=1)
            -> Table scan on Statistics  (cost=2048.55 rows=19923) (actual time=0.029..4.782 rows=17812 loops=1)
 |
```

| -> Filter: (avg_views > (select #3)) (actual time=188.730..189.528 rows=345 loops=1)
-> Table scan on <temporary> (actual time=183.015..183.436 rows=1701 loops=1)
-> Aggregate using temporary table (actual time=183.014..183.014 rows=1701 loops=1)

```
          -> Nested loop inner join  (cost=15994.65 rows=19923) (actual time=0.186..176.641
rows=3561 loops=1)
              -> Nested loop inner join  (cost=9021.60 rows=19923) (actual time=0.177..169.422
rows=3561 loops=1)
                  -> Filter: ((s.trending_date = (select #2)) and (s.video_id is not null))  (cost=2048.55
rows=19923) (actual time=0.164..153.234 rows=3561 loops=1)
                      -> Table scan on s  (cost=2048.55 rows=19923) (actual time=0.053..8.477
rows=17812 loops=1)
                      -> Select #2 (subquery in condition; dependent)
                          -> Aggregate: max(s2.trending_date)  (cost=3.24 rows=1) (actual
time=0.007..0.007 rows=1 loops=17812)
                              -> Covering index lookup on s2 using new_index_name
(video_id=s.video_id)  (cost=2.68 rows=6) (actual time=0.003..0.005 rows=8 loops=17812)
                  -> Filter: (v.channelId is not null)  (cost=0.25 rows=1) (actual time=0.004..0.004
rows=1 loops=3561)
                      -> Single-row index lookup on v using PRIMARY (video_id=s.video_id)
(cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=3561)
              -> Single-row index lookup on y using PRIMARY (channelId=v.channelId)  (cost=0.25
rows=1) (actual time=0.002..0.002 rows=1 loops=3561)
    -> Select #3 (subquery in condition; run only once)
        -> Aggregate: avg(Statistics.view_count)  (cost=4040.85 rows=1) (actual time=5.677..5.678
rows=1 loops=1)
            -> Covering index scan on Statistics using idx_stats_view_count  (cost=2048.55
rows=19923) (actual time=0.036..3.912 rows=17812 loops=1)
 |
```

This index is on view_count, which would be useful for queries that involve sorting or filtering videos based on the number of views. For instance, a common operation might be to find the most or least viewed videos in a certain timeframe. The index would speed up these kinds of queries by allowing the database to quickly locate rows within a certain range of view_count without having to scan the entire table.

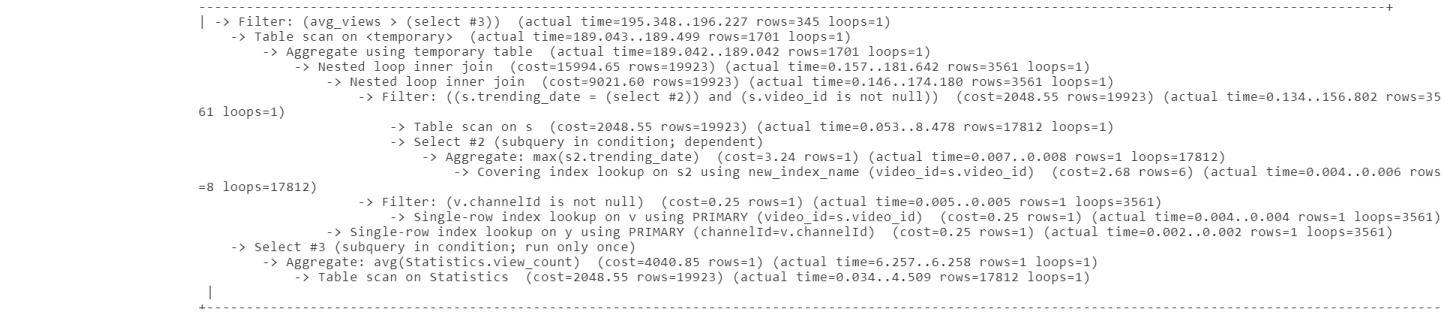
This index is also relatively fast, with an actual time of 188.730 to 189.528. It is created on the view_count column, which is used in the AVG function. Since the index allows for quicker access to view_count, it speeds up the aggregation process when calculating the average view count in your query. However, the overall performance improvement might be slightly slower compared to idx_stats_video_trending due to the nature of the query.

Index: idx_youtuber_id_title

Then we drop the index with

```
DROP INDEX idx_stats_view_count ON Statistics;
```

3. CREATE INDEX idx_youtuber_id_title ON YouTuber (channelId, channelTitle);



```
| -> Filter: (avg_views > (select #3))  (actual time=195.348..196.227 rows=345 loops=1)
    -> Table scan on <temporary>  (actual time=189.043..189.499 rows=1701 loops=1)
        -> Aggregate using temporary table  (actual time=189.042..189.042 rows=1701 loops=1)
            -> Nested loop inner join  (cost=15994.65 rows=19923) (actual time=0.157..181.642 rows=3561 loops=1)
                -> Nested loop inner join  (cost=9021.60 rows=19923) (actual time=0.146..174.180 rows=3561 loops=1)
                    -> Filter: ((s.trending_date = (select #2)) and (s.video_id is not null))  (cost=2048.55 rows=19923) (actual time=0.134..156.802 rows=3561 loops=1)
                        -> Table scan on s  (cost=2048.55 rows=19923) (actual time=0.053..8.478 rows=17812 loops=1)
                        -> Select #2 (subquery in condition; dependent)
                            -> Aggregate: max(s2.trending_date)  (cost=3.24 rows=1) (actual time=0.007..0.008 rows=1 loops=17812)
                                -> Covering index lookup on s2 using new_index_name (video_id=s.video_id)  (cost=2.68 rows=6) (actual time=0.004..0.006 rows=8 loops=17812)
                    -> Filter: (v.channelId is not null)  (cost=0.25 rows=1) (actual time=0.005..0.005 rows=1 loops=3561)
                        -> Single-row index lookup on v using PRIMARY (video_id=s.video_id)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=3561)
                -> Single-row index lookup on y using PRIMARY (channelId=v.channelId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=3561)
    -> Select #3 (subquery in condition; run only once)
        -> Aggregate: avg(Statistics.view_count)  (cost=4040.85 rows=1) (actual time=6.257..6.258 rows=1 loops=1)
            -> Table scan on Statistics  (cost=2048.55 rows=19923) (actual time=0.034..4.509 rows=17812 loops=1)
 |
```

| -> Filter: (avg_views > (select #3)) (actual time=195.348..196.227 rows=345 loops=1)
-> Table scan on <temporary> (actual time=189.043..189.499 rows=1701 loops=1)
-> Aggregate using temporary table (actual time=189.042..189.042 rows=1701 loops=1)
-> Nested loop inner join (cost=15994.65 rows=19923) (actual time=0.157..181.642 rows=3561 loops=1)
-> Nested loop inner join (cost=9021.60 rows=19923) (actual time=0.146..174.180 rows=3561 loops=1)
-> Filter: ((s.trending_date = (select #2)) and (s.video_id is not null)) (cost=2048.55 rows=19923) (actual time=0.134..156.802 rows=3561 loops=1)
-> Table scan on s (cost=2048.55 rows=19923) (actual time=0.053..8.478 rows=17812 loops=1)
-> Select #2 (subquery in condition; dependent)
-> Aggregate: max(s2.trending_date) (cost=3.24 rows=1) (actual time=0.007..0.008 rows=1 loops=17812)
-> Covering index lookup on s2 using new_index_name (video_id=s.video_id) (cost=2.68 rows=6) (actual time=0.004..0.006 rows=8 loops=17812)
-> Filter: (v.channelId is not null) (cost=0.25 rows=1) (actual time=0.005..0.005 rows=1 loops=3561)
-> Single-row index lookup on v using PRIMARY (video_id=s.video_id) (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=3561)
-> Single-row index lookup on y using PRIMARY (channelId=v.channelId) (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=3561)
-> Select #3 (subquery in condition; run only once)
-> Aggregate: avg(Statistics.view_count) (cost=4040.85 rows=1) (actual time=6.257..6.258 rows=1 loops=1)
-> Table scan on Statistics (cost=2048.55 rows=19923) (actual time=0.034..4.509 rows=17812 loops=1)
|

This index could be justified in a general situation where queries frequently need to filter or sort by both channelId and channelTitle. This index might facilitate quicker joins between the YouTuber table and other tables using channelId, while also accelerating searches that involve the channelTitle. However, because channelId is a primary key and already indexed, the

additional benefit of this composite index would be more pronounced in scenarios where channelTitle is also a common query parameter.

This index is also relatively fast, with an actual time of 195.348 to 196.227. It is created on the channelId and channelTitle columns in the YouTuber table. This index speeds up the JOIN operation between YouTuber and Statistics tables by optimizing access to the join columns. The relatively fast performance is due to the efficient handling of the JOIN operations.

Then we drop the index with

```
DROP INDEX idx_youtuber_id_title ON YouTuber;
```

Each Index ran this code:

```
EXPLAIN ANALYZE
SELECT
    y.channelTitle,
    AVG(s.view_count) AS avg_views
FROM Video v
JOIN YouTuber y ON v.channelId = y.channelId
JOIN Statistics s ON v.video_id = s.video_id
WHERE s.trending_date = (
    SELECT MAX(trending_date)
    FROM Statistics s2
    WHERE s2.video_id = s.video_id
)
GROUP BY y.channelTitle
HAVING avg_views > (SELECT AVG(view_count) FROM Statistics);
```