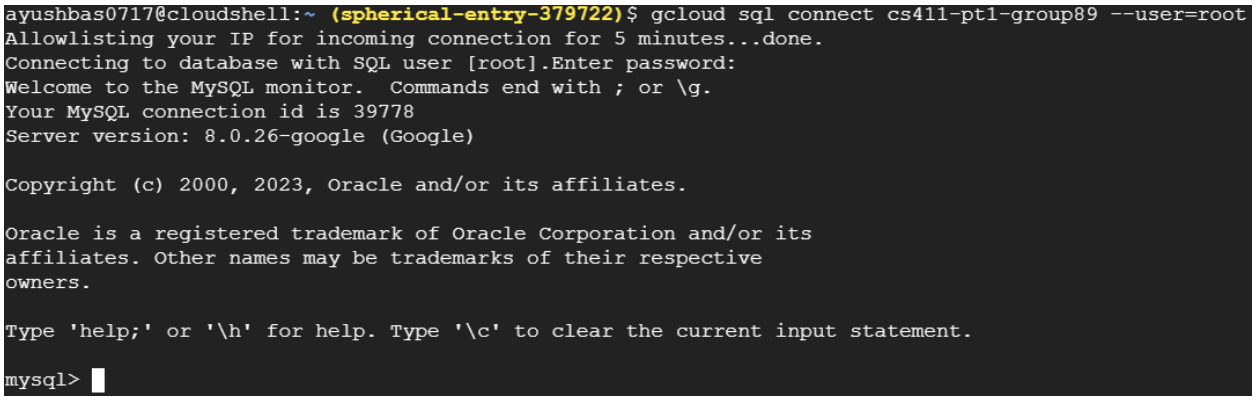
Stage 3, Group 89: Ayush, Riya, Sritha, and Yue

Connection to GCP



DDL COMMANDS

```
CREATE DATABASE YoutubeTrending

CREATE TABLE AllData(
        video_id VARCHAR(50),
        title VARCHAR(100),
        publishedAt TEXT,
        channelId VARCHAR(50),
        channelTitle VARCHAR(100),
        categoryId INT,
        trending_date TEXT,
        tags TEXT,
        view_count INT,
        likes INT,
        dislikes INT,
        comment_count INT,
        thumbnail_link VARCHAR(200),
        comments_disabled BOOLEAN,
        ratings_disabled BOOLEAN,
        description TEXT
);
```

This is a temporary table created to import all the data from the downloaded csv file from [here](#) which has been cleaned in Python.

```python
import pandas as pd
#load the downloaded dataset into a Pandas dataframe
df = pd.read_csv('US_youtube_trending_data.csv')
#drop any duplicate videos keeping the entry with the highest view_count
#this should be the most recent entry which is most useful to our project
dropped = df.sort_values('view_count', ascending=False).drop_duplicates('video_id')
#covert to cleaned dataset to a csv file which we can import into GCP
dropped.to_csv('fixed_data.csv', index=False)
```

✓ 6.9s

VideoInfo(video_id:VARCHAR(50) [PK], title:VARCHAR(100), publishedAt: TEXT, tags:TEXT, description:TEXT, channelId:VARCHAR(50), categoryId:INT, view_count: INT, likes: INT, dislikes: INT, comment_count: INT)

```
CREATE TABLE VideoInfo(
        video_id VARCHAR(50),
        title VARCHAR(100),
        publishedAt TEXT,
        tags TEXT,
        description TEXT,
        channelId VARCHAR(50),
        categoryId INT,
        view_count INT,
        likes INT,
        dislikes INT,
        comment_count INT,
        PRIMARY KEY(video_id)
);

INSERT INTO VideoInfo
SELECT video_id, title, publishedAt, tags, description, channelId, categoryId, view_count, likes,
dislikes, comment_count
FROM AllData;
```

```
mysql> select count(*) from VideoInfo;
+----------+
| count(*) |
+----------+
|     4613 |
+----------+
```

Creators(channelId:VARCHAR(50) [PK], channelTitle:VARCHAR(100), categoryId:INT, video_id:VARCHAR(50) [FK to VideoInfo.video_id])

```
CREATE TABLE Creators (
        channelId VARCHAR(50),
        channelTitle VARCHAR(100),
        categoryId INT,
        video_id VARCHAR(50),
        PRIMARY KEY(channelId),
        FOREIGN KEY(video_id) REFERENCES VideoInfo(video_id)
        ON DELETE CASCADE
);

INSERT IGNORE INTO Creators
SELECT DISTINCT channelId, channelTitle, categoryId, video_id
FROM AllData;
```

```
mysql> select count(*) from Creators;
+----------+
| count(*) |
+----------+
|     2434 |
+----------+
```

CategoryInfo(categoryId:INT [PK] [FK to TrendingKeywords.categoryId], totalPublished:INT, totalLiked:INT, totalChannels:INT, totalViews:INT)

```
CREATE INDEX category_index ON VideoInfo(categoryId);

CREATE TABLE CategoryInfo (
        categoryId INT,
        totalPublished INT,
        totalLiked BIGINT,
        totalChannels INT,
        totalViews BIGINT,
        PRIMARY KEY(categoryId),
        FOREIGN KEY(categoryId) REFERENCES VideoInfo(categoryId)
        ON DELETE CASCADE
);

INSERT INTO CategoryInfo(categoryId, totalChannels, totalViews, totalLiked, totalPublished)
SELECT t1.categoryId, t1.totalChannels, t2.totalViews, t2.totalLiked, t2.totalPublished
FROM
   (SELECT categoryId, COUNT(channelId) AS totalChannels
    FROM AllData
    GROUP BY categoryId) AS t1
JOIN
   (SELECT categoryId, SUM(view_count) AS totalViews, SUM(likes) AS totalLiked,
COUNT(video_id) AS totalPublished
    FROM AllData
    GROUP BY categoryId) AS t2
ON t1.categoryId = t2.categoryId;
```

```
mysql> select count(*) from CategoryInfo;
+----------+
| count(*) |
+----------+
|       16 |
+----------+
```

TrendingKeywords(keywords: VARCHAR(50) [PK], categoryId: INT, use_count: INT)

```
CREATE TABLE TrendingKeywords (
        keywords VARCHAR(50),
        use_count INT,
        categoryId INT,
        PRIMARY KEY(keywords)
);
```

This code is a temporary implementation to generate keywords using SQL. We will change this to be generated through Python packages in later stages.

```
 CREATE TEMPORARY TABLE temp_words AS
 SELECT SUBSTRING_INDEX(SUBSTRING_INDEX(t.title, ' ', n.n), ' ', -1) AS word
 FROM VideoInfo t
 CROSS JOIN (
   SELECT a.N + b.N * 10 + 1 AS n
   FROM (SELECT 0 AS N UNION SELECT 1 UNION SELECT 2 UNION SELECT 3 UNION
SELECT 4 UNION SELECT 5 UNION SELECT 6 UNION SELECT 7 UNION SELECT 8 UNION
SELECT 9) a
   CROSS JOIN (SELECT 0 AS N UNION SELECT 1 UNION SELECT 2 UNION SELECT 3
UNION SELECT 4 UNION SELECT 5 UNION SELECT 6 UNION SELECT 7 UNION SELECT 8
UNION SELECT 9) b
   ORDER BY n
 ) n
 WHERE n.n <= 1 + (CHAR_LENGTH(t.title) - CHAR_LENGTH(REPLACE(t.title, ' ', '')))
 AND t.title IS NOT NULL;

DELETE FROM temp_words WHERE word IN ('the', 'and', 'or');

INSERT IGNORE INTO TrendingKeywords (keywords, use_count)
SELECT word, COUNT(*) AS count
FROM temp_words
GROUP BY word
ORDER BY count DESC;

DROP TEMPORARY TABLE temp_words;
```

```
mysql> select count(*) from TrendingKeywords;
+----------+
| count(*) |
+----------+
|    11676 |
+----------+
```

WebsiteUsers(user_id: INT [PK], username: VARCHAR(30), password: VARCHAR(30), email: VARCHAR(50), channelId: VARCHAR(50) [FK to Creators.channelId])

```
CREATE TABLE WebsiteUsers (
        user_id INT,
        username VARCHAR(30),
        password VARCHAR(30),
        email VARCHAR(50),
        channelId VARCHAR(50),
        PRIMARY KEY(user_id),
        FOREIGN KEY(channelId) REFERENCES Creators(channelId) ON DELETE CASCADE
);
```

```
CREATE INDEX category_index ON TrendingKeywords(categoryId);

CREATE TABLE KeywordToVideo (
  video_id VARCHAR(50),
  categoryId INT,
  FOREIGN KEY (video_id) REFERENCES VideoInfo(video_id),
  FOREIGN KEY (categoryId) REFERENCES TrendingKeywords(categoryId)
);
```

Deleting Temporary Table used to import data

```
DROP TABLE AllData;
```

ADVANCED SQL QUERIES

SQL Query #1: retrieves the video titles and the view counts for the top 10 most viewed videos that were uploaded by creators who have one other video with over 1 million views

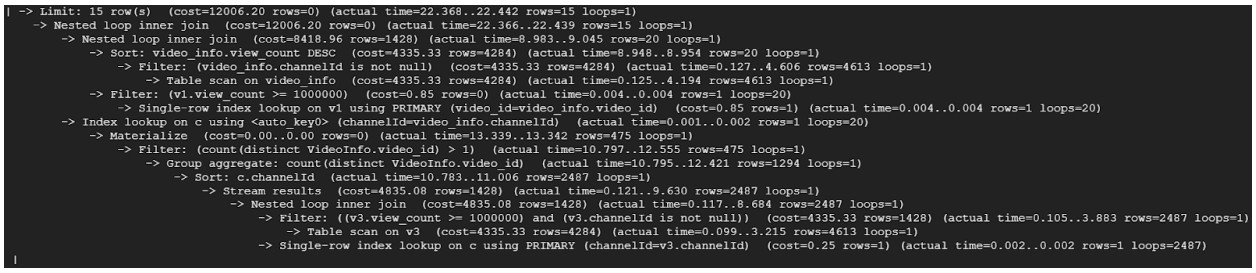
```
SELECT v.title, v.view_count
FROM VideoInfo v
JOIN
(
  SELECT v1.video_id
  FROM VideoInfo v1
  WHERE v1.view_count >= 1000000
) v1
ON v.video_id = v1.video_id
JOIN
(
  SELECT DISTINCT c.channelId
  FROM Creators c
  JOIN VideoInfo v2 ON c.channelId = v2.channelId
  WHERE v2.view_count >= 1000000
  GROUP BY c.channelId
  HAVING COUNT(DISTINCT v2.video_id) > 1
) c
ON v.channelId = c.channelId
ORDER BY v.view_count DESC
LIMIT 15;
```

Screenshot showing top 15 rows with advanced query results:

```
+-------------------------------------------------------------------------------------+------------+
| title                                                                               | view_count |
+-------------------------------------------------------------------------------------+------------+
| BLACKPINK - 'Ice Cream (with Selena Gomez)' M/V                                     |  184778248 |
| Dice Stacks from $1 to $100                                                         |  103564168 |
| BTS (방탄소년단) 'Yet To Come (The Most Beautiful Moment)' Official MV              |   93952431 |
| Bella Poarch - Build a B*tch (Official Music Video)                                 |   84063330 |
| so long nerds                                                                       |   68010978 |
| Last To Take Hand Off Jet, Keeps It!                                                |   52206793 |
| DJ Snake, Ozuna, Megan Thee Stallion, LISA of BLACKPINK - SG (Official Music Video) |   44849154 |
| Coldplay X BTS - My Universe (Official Video)                                       |   43856062 |
| BTS (방탄소년단) 'Butter (Hotter Remix)' Official MV                                |   43163081 |
| ITZY "Cheshire" M/V @ITZY                                                           |   42406775 |
| Christian Nodal, Ángela Aguilar - Dime Cómo Quieres (Video Oficial)                 |   39713484 |
| The biggest news from the Apple Event | Apple                                       |   39700876 |
| Nicky Jam || BZRP Music Sessions #41                                                |   39571352 |
| SOMI (전소미) - 'DUMB DUMB' M/V                                                     |   39414712 |
| hi, I'm Dream.                                                                      |   39252484 |
+-------------------------------------------------------------------------------------+------------+
15 rows in set (5.27 sec)
```

Indexing

EXPLAIN ANALYZE:



```
| -> Limit: 15 row(s)  (cost=12006.20 rows=0) (actual time=22.368..22.442 rows=15 loops=1)
    -> Nested loop inner join  (cost=12006.20 rows=0) (actual time=22.366..22.439 rows=15 loops=1)
      -> Nested loop inner join  (cost=8418.96 rows=1428) (actual time=8.983..9.045 rows=20 loops=1)
        -> Sort: video_info.view_count DESC  (cost=4335.33 rows=4284) (actual time=8.948..8.954 rows=20 loops=1)
          -> Filter: (video_info.channelId is not null)  (cost=4335.33 rows=4284) (actual time=0.127..4.606 rows=4613 loops=1)
            -> Table scan on video_info  (cost=4335.33 rows=4284) (actual time=0.125..4.194 rows=4613 loops=1)
        -> Filter: (v1.view_count >= 1000000)  (cost=0.85 rows=0) (actual time=0.004..0.004 rows=1 loops=20)
          -> Single-row index lookup on v1 using PRIMARY (video_id=video_info.video_id)  (cost=0.85 rows=1) (actual time=0.004..0.004 rows=1 loops=20)
      -> Index lookup on c using <auto_key0> (channelId=video_info.channelId)  (actual time=0.001..0.002 rows=1 loops=20)
        -> Materialize  (cost=0.00..0.00 rows=0) (actual time=13.339..13.342 rows=475 loops=1)
          -> Filter: (count(distinct VideoInfo.video_id) > 1)  (actual time=10.797..12.555 rows=475 loops=1)
            -> Group aggregate: count(distinct VideoInfo.video_id)  (actual time=10.795..12.421 rows=1294 loops=1)
              -> Sort: c.channelId  (actual time=10.783..11.006 rows=2487 loops=1)
                -> Stream results  (cost=4835.08 rows=1428) (actual time=0.121..9.630 rows=2487 loops=1)
                  -> Nested loop inner join  (cost=4835.08 rows=1428) (actual time=0.117..8.684 rows=2487 loops=1)
                    -> Filter: ((v3.view_count >= 1000000) and (v3.channelId is not null))  (cost=4335.33 rows=1428) (actual time=0.105..3.883 rows=2487 loops=1)
                      -> Table scan on v3  (cost=4335.33 rows=4284) (actual time=0.099..3.215 rows=4613 loops=1)
                    -> Single-row index lookup on c using PRIMARY (channelId=v3.channelId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=2487)
|
```

COST = 12006.20

CREATE INDEX channelId_index on VideoInfo(channelId);

EXPLAIN ANALYZE:

```
| -> Limit: 15 row(s)  (actual time=122.706..122.709 rows=15 loops=1)
    -> Sort: v.view_count DESC, limit input to 15 row(s) per chunk  (actual time=122.706..122.707 rows=15 loops=1)
        -> Stream results  (cost=9424.02 rows=838) (actual time=111.984..122.439 rows=1668 loops=1)
            -> Nested loop inner join  (cost=9424.02 rows=838) (actual time=111.982..121.934 rows=1668 loops=1)
                -> Nested loop inner join  (cost=7029.73 rows=2513) (actual time=111.974..119.022 rows=1985 loops=1
)
                    -> Table scan on c  (cost=0.01..20.34 rows=1428) (actual time=0.002..0.102 rows=475 loops=1)
                        -> Materialize  (cost=4615.11..4635.44 rows=1428) (actual time=111.957..112.089 rows=475 lo
ops=1)
                            -> Filter: (count(distinct v2.video_id) > 1)  (cost=4472.31 rows=1428) (actual time=0.1
72..111.710 rows=475 loops=1)
                                -> Group aggregate: count(distinct v2.video_id)  (cost=4472.31 rows=1428) (actual t
ime=0.171..111.457 rows=1294 loops=1)
                                    -> Nested loop inner join  (cost=4329.53 rows=1428) (actual time=0.153..109.613
 rows=2487 loops=1)
                                        -> Index scan on c using PRIMARY  (cost=245.90 rows=2434) (actual time=0.04
0..0.868 rows=2434 loops=1)
                                        -> Filter: (v2.view_count >= 1000000)  (cost=1.50 rows=1) (actual time=0.04
3..0.044 rows=1 loops=2434)
                                            -> Index lookup on v2 using channelId_index (channelId=c.channelId)  (c
ost=1.50 rows=2) (actual time=0.042..0.044 rows=2 loops=2434)
                -> Index lookup on v using channelId_index (channelId=c.channelId)  (cost=1.50 rows=2) (actualt
ime=0.009..0.014 rows=4 loops=475)
            -> Filter: (v1.view_count >= 1000000)  (cost=0.85 rows=0) (actual time=0.001..0.001 rows=1 loops=19
85)
```

Cost = 9424.02

Justification:
As seen in the above images, indexing on channelId reduced the cost from 12006.20 to 9424.02, which is a marked improvement in optimization. We chose to index on channelId as our query performs a join operation of the VideoInfo and Creators on channelId - when we are performing a join operation, the database needs to match the values in the join columns of the tables being joined. Indexing these columns reduces the amount of data that needs to be scanned which allows the data to be retrieved more efficiently. In terms of our overall project, several of our future queries will be using this same join command, so indexing on channelId will also help in optimizing any future queries we write with these same two columns joined on channelId.

DROP INDEX channelId_index on VideoInfo;

CREATE INDEX view_count_index on VideoInfo(view_count);

EXPLAIN ANALYZE:

```
| -> Limit: 15 row(s)  (cost=9892.30 rows=0) (actual time=17.278..17.433 rows=15 loops=1)
    -> Nested loop inner join  (cost=9892.30 rows=0) (actual time=17.278..17.430 rows=15 loops=1)
        -> Nested loop inner join  (cost=3657.25 rows=1) (actual time=0.054..0.202 rows=20 loops=1)
            -> Filter: (v.channelId is not null)  (cost=1.82 rows=2) (actual time=0.046..0.138 rows=20 loops=1)
                -> Index scan on v using view_count_index (reverse)  (cost=1.82 rows=2) (actual time=0.045..0.135 r
ows=20 loops=1)
            -> Filter: (v1.view_count >= 1000000)  (cost=0.85 rows=1) (actual time=0.003..0.003 rows=1 loops=20)
                -> Single-row index lookup on v1 using PRIMARY (video_id=v.video_id)  (cost=0.85 rows=1) (actual ti
me=0.003..0.003 rows=1 loops=20)
        -> Index lookup on c using <auto_key0> (channelId=v.channelId)  (actual time=0.002..0.002 rows=1 loops=20)
            -> Materialize  (cost=0.00..0.00 rows=0) (actual time=17.217..17.221 rows=475 loops=1)
                -> Filter: (count(distinct VideoInfo.video_id) > 1)  (actual time=14.566..16.504 rows=475 loops=1)
                    -> Group aggregate: count(distinct VideoInfo.video_id)  (actual time=14.564..16.377 rows=1294 l
oops=1)
                        -> Sort: c.channelId  (actual time=14.551..14.919 rows=2487 loops=1)
                            -> Stream results  (cost=3490.69 rows=2487) (actual time=0.378..13.424 rows=2487 loops=
1)
                                -> Nested loop inner join  (cost=3490.69 rows=2487) (actual time=0.376..12.572 rows
=2487 loops=1)
                                    -> Filter: (v2.channelId is not null)  (cost=2620.24 rows=2487) (actual time=0.
367..8.253 rows=2487 loops=1)
                                        -> Index range scan on v2 using view_count_index, with index condition: (v2
.view_count >= 1000000)  (cost=2620.24 rows=2487) (actual time=0.366..8.032 rows=2487 loops=1)
                                    -> Single-row index lookup on c using PRIMARY (channelId=v2.channelId)  (cost=0
.25 rows=1) (actual time=0.002..0.002 rows=1 loops=2487)
 |
```

Cost = 9892.30

Justification:

As seen in the above images, indexing on VideoInfo reduced the cost from 12006.20 to 9892.30, which is a marked improvement in optimization, but is a smaller decrease than what we saw when we indexed on channelId. We chose to index on VideoInfo as our query performs a join operation of the VideoInfo and Creators - when we are performing a join operation, the database needs to match the values in the join columns of the tables being joined. Indexing these columns reduces the amount of data that needs to be scanned which allows the data to be retrieved more efficiently. In terms of our overall project, several of our future queries will be using this same join command, so indexing on VideoInfo will also help in optimizing any future queries we write with these same two columns joined.

DROP INDEX view_count_index on VideoInfo;

CREATE INDEX title_index on VideoInfo(title);

EXPLAIN ANALYZE:

```
| -> Limit: 15 row(s)  (cost=12006.20 rows=0) (actual time=25.157..25.246 rows=15 loops=1)
    -> Nested loop inner join  (cost=12006.20 rows=0) (actual time=25.156..25.243 rows=15 loops=1)
        -> Nested loop inner join  (cost=8418.96 rows=1428) (actual time=10.731..10.814 rows=20 loops=1)
            -> Sort: v.view_count DESC  (cost=4335.33 rows=4284) (actual time=10.689..10.695 rows=20 loops=1)
                -> Filter: (v.channelId is not null)  (cost=4335.33 rows=4284) (actual time=0.122..5.721 rows=4613
loops=1)
                    -> Table scan on v  (cost=4335.33 rows=4284) (actual time=0.120..5.111 rows=4613 loops=1)
            -> Filter: (v1.view_count >= 1000000)  (cost=0.85 rows=0) (actual time=0.006..0.006 rows=1 loops=20)
                -> Single-row index lookup on v1 using PRIMARY (video_id=v.video_id)  (cost=0.85 rows=1) (actual ti
me=0.005..0.005 rows=1 loops=20)
        -> Index lookup on c using <auto_key0> (channelId=v.channelId)  (actual time=0.002..0.003 rows=1 loops=20)
            -> Materialize  (cost=0.00..0.00 rows=0) (actual time=14.417..14.421 rows=475 loops=1)
                -> Filter: (count(distinct VideoInfo.video_id) > 1)  (actual time=11.990..13.683 rows=475 loops=1)
                    -> Group aggregate: count(distinct VideoInfo.video_id)  (actual time=11.987..13.570 rows=1294 l
oops=1)
                        -> Sort: c.channelId  (actual time=11.893..12.139 rows=2487 loops=1)
                            -> Stream results  (cost=4835.08 rows=1428) (actual time=0.190..10.521 rows=2487 loops=
1)
                                -> Nested loop inner join  (cost=4835.08 rows=1428) (actual time=0.187..9.564 rows=
2487 loops=1)
                                    -> Filter: ((v2.view_count >= 1000000) and (v2.channelId is not null))  (cost=4
335.33 rows=1428) (actual time=0.172..4.511 rows=2487 loops=1)
                                        -> Table scan on v2  (cost=4335.33 rows=4284) (actual time=0.166..3.938 row
s=4613 loops=1)
                                    -> Single-row index lookup on c using PRIMARY (channelId=v2.channelId)  (cost=0
.25 rows=1) (actual time=0.002..0.002 rows=1 loops=2487)
 |
```

Cost = 12006.20

Justification:

As seen in the above images, indexing on title did not reduce the cost from 12006.20. We chose to index on VideoInfo as our query selects the title, and because it was the only other attribute in our query. This did not make much of a difference in our optimization as this attribute is only used in our select command. Although we may be selecting title in future queries, indexing on it as an attribute did not show a marked improvement in cost. Indexing on channelId and VideoInfo as shown above are better options for indexing.

DROP INDEX view_count_index on VideoInfo;

CREATE INDEX view_count_index on VideoInfo(view_count);
CREATE INDEX channelId_index on VideoInfo(channelId);

EXPLAIN ANALYZE:

```
| -> Limit: 15 row(s)  (actual time=40.840..40.842 rows=15 loops=1)
    -> Sort: v.view_count DESC, limit input to 15 row(s) per chunk  (actual time=40.839..40.840 rows=15 loops=1)
        -> Stream results  (cost=7472.14 rows=0) (actual time=16.665..40.303 rows=1668 loops=1)
            -> Nested loop inner join  (cost=7472.14 rows=0) (actual time=16.661..39.691 rows=1668 loops=1)
                -> Nested loop inner join  (cost=3737.32 rows=0) (actual time=16.650..36.019 rows=1985 loops=1)
                    -> Table scan on c  (cost=2.50..2.50 rows=0) (actual time=0.003..0.421 rows=475 loops=1)
                        -> Materialize  (cost=2.50..2.50 rows=0) (actual time=16.596..17.054 rows=475 loops=1)
                            -> Filter: (count(distinct VideoInfo.video_id) > 1)  (actual time=14.499..16.461 rows=4
75 loops=1)
                                -> Group aggregate: count(distinct VideoInfo.video_id)  (actual time=14.497..16.343
 rows=1294 loops=1)
                                    -> Sort: c.channelId  (actual time=14.485..14.969 rows=2487 loops=1)
                                        -> Stream results  (cost=3490.69 rows=2487) (actual time=0.369..13.332 rows
=2487 loops=1)
                                            -> Nested loop inner join  (cost=3490.69 rows=2487) (actual time=0.367.
.12.446 rows=2487 loops=1)
                                                -> Filter: (v2.channelId is not null)  (cost=2620.24 rows=2487) (ac
tual time=0.357..8.032 rows=2487 loops=1)
                                                    -> Index range scan on v2 using view_count_index, with index co
ndition: (v2.view_count >= 1000000)  (cost=2620.24 rows=2487) (actual time=0.355..7.824 rows=2487 loops=1)
                                                -> Single-row index lookup on c using PRIMARY (channelId=v2.channel
Id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=2487)
                    -> Index lookup on v using channelId_index (channelId=c.channelId)  (cost=1.50 rows=2) (actual
time=0.033..0.039 rows=4 loops=475)
                -> Filter: (v1.view_count >= 1000000)  (cost=0.85 rows=1) (actual time=0.002..0.002 rows=1 loops=19
85)
                    -> Single-row index lookup on v1 using PRIMARY (video_id=v.video_id)  (cost=0.85 rows=1) (actua
l time=0.001..0.001 rows=1 loops=1985)
 |
```

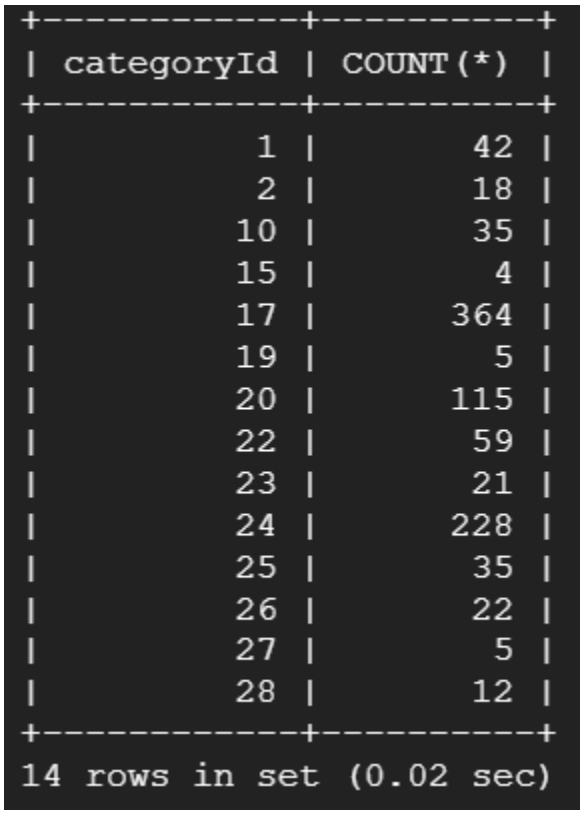
Cost = 7472.14

As we noted previously, indexing on channelId and view_count together significantly improves the performance of this query. Furthermore, this should not degrade the performance of other queries overall since we will be using the relationships between VideoInfo and Creators tables through channelId often as well as frequently using VideoInfo.view_count to sort our results.

SQL Query #2: counts the number of videos for each category which has a title that contains the most commonly used keyword

```
SELECT categoryId, COUNT(*) FROM VideoInfo
WHERE title LIKE
CONCAT('%',
(
  SELECT keywords
  FROM TrendingKeywords
  ORDER BY use_count DESC
  LIMIT 1
),
 '%')
GROUP BY categoryId;
```

Screenshot showing top 15 rows with advanced query results:



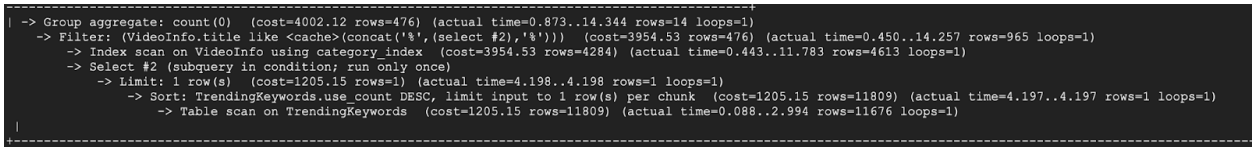
categoryId	COUNT(*)
1	42
2	18
10	35
15	4
17	364
19	5
20	115
22	59
23	21
24	228
25	35
26	22
27	5
28	12

14 rows in set (0.02 sec)

There are only 14 categoryIds so the output only has 14 rows.

INDEXING

DROP INDEX ‘PRIMARY’ on TrendingKeywords
Result of EXPLAIN ANALYZE with no indexes:

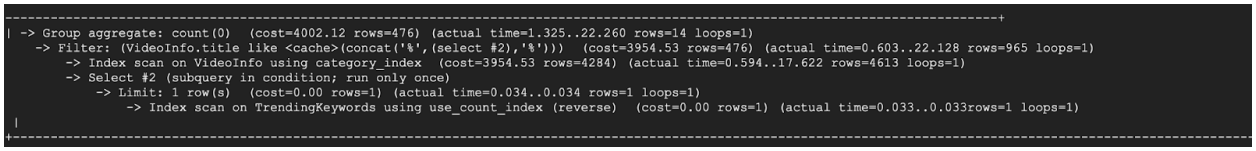


```
| -> Group aggregate: count(0)  (cost=4002.12 rows=476) (actual time=0.873..14.344 rows=14 loops=1)
    -> Filter: (VideoInfo.title like <cache>(concat('%',(select #2),'%')))  (cost=3954.53 rows=476) (actual time=0.450..14.257 rows=965 loops=1)
        -> Index scan on VideoInfo using category_index  (cost=3954.53 rows=4284) (actual time=0.443..11.783 rows=4613 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Limit: 1 row(s)  (cost=1205.15 rows=1) (actual time=4.198..4.198 rows=1 loops=1)
                -> Sort: TrendingKeywords.use_count DESC, limit input to 1 row(s) per chunk  (cost=1205.15 rows=11809) (actual time=4.197..4.197 rows=1 loops=1)
                    -> Table scan on TrendingKeywords  (cost=1205.15 rows=11809) (actual time=0.088..2.994 rows=11676 loops=1)
 |
```

Cost = 4002.12

CREATE INDEX keywords_index ON TrendingKeywords (keywords);
CREATE INDEX use_count_index ON TrendingKeywords(use_count);

EXPLAIN ANALYZE:



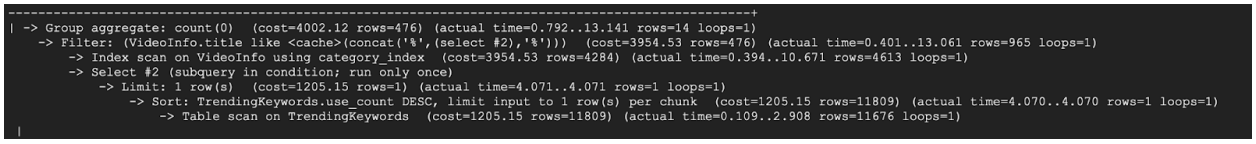
```
| -> Group aggregate: count(0)  (cost=4002.12 rows=476) (actual time=1.325..22.260 rows=14 loops=1)
    -> Filter: (VideoInfo.title like <cache>(concat('%',(select #2),'%')))  (cost=3954.53 rows=476) (actual time=0.603..22.128 rows=965 loops=1)
        -> Index scan on VideoInfo using category_index  (cost=3954.53 rows=4284) (actual time=0.594..17.622 rows=4613 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Limit: 1 row(s)  (cost=0.00 rows=1) (actual time=0.034..0.034 rows=1 loops=1)
                -> Index scan on TrendingKeywords using use_count_index (reverse)  (cost=0.00 rows=1) (actual time=0.033..0.033rows=1 loops=1)
 |
```

Cost = 4002.12

DROP INDEX keywords_index ON TrendingKeywords;
DROP INDEX use_count_index ON TrendingKeywords;

CREATE INDEX keywords_hash_index USING HASH ON TrendingKeywords (keywords);
CREATE INDEX use_count_hash_index USING HASH ON TrendingKeywords (use_count);

EXPLAIN ANALYZE:



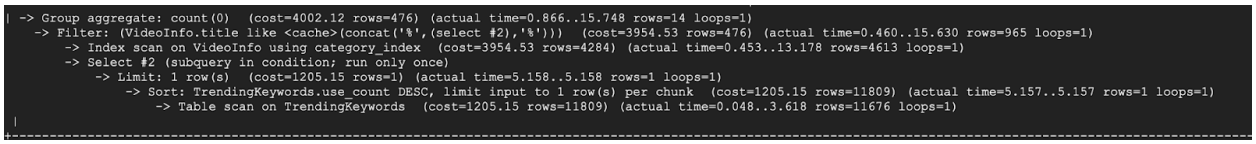
```
| -> Group aggregate: count(0)  (cost=4002.12 rows=476) (actual time=0.792..13.141 rows=14 loops=1)
    -> Filter: (VideoInfo.title like <cache>(concat('%',(select #2),'%')))  (cost=3954.53 rows=476) (actual time=0.401..13.061 rows=965 loops=1)
        -> Index scan on VideoInfo using category_index  (cost=3954.53 rows=4284) (actual time=0.394..10.671 rows=4613 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Limit: 1 row(s)  (cost=1205.15 rows=1) (actual time=4.071..4.071 rows=1 loops=1)
                -> Sort: TrendingKeywords.use_count DESC, limit input to 1 row(s) per chunk  (cost=1205.15 rows=11809) (actual time=4.070..4.070 rows=1 loops=1)
                    -> Table scan on TrendingKeywords  (cost=1205.15 rows=11809) (actual time=0.109..2.908 rows=11676 loops=1)
 |
```

Cost = 4002.12

DROP INDEX keywords_hash_index ON TrendingKeywords;
DROP INDEX use_count_hash_index ON TrendingKeywords;

CREATE UNIQUE INDEX keywords_unique_index on TrendingKeywords(keywords);
CREATE UNIQUE INDEX use_count_unique_index on TrendingKeywords(use_count);

EXPLAIN ANALYZE:



Cost = 4002.12

Justification:

As seen in the above images, indexing elements of this query did not affect or improve the query cost. We attempted to index the key elements in the query, “keywords” and “use_count”. By using the SHOW INDEX command, we found that “keywords” had already been indexed because it was the primary key for the TrendingKeywords tables. We dropped this to see what the default cost of the query would be, and it was 4002.12. After adding indexes on “keywords” and “use_count” the cost did not change. We dropped these indexes and added hash indexes. We found that we could not add a hash index of “keywords” because it has type VARCHAR. Adding a hash index on use_count did not change anything. Adding unique indexes also did not affect the cost because the column “keywords” already contained unique values.

We theorized that this query’s cost does not improve because of one main reason. The query mainly utilizes the “keywords” column. Indexing this column does not help too much because it cannot truly be organized in an ascending or descending order so it must be traversed in its entirety. Other table columns do not have as significant of a role in this query because they are only being used in the ORDER BY. Hence, this query’s cost cannot be optimized.