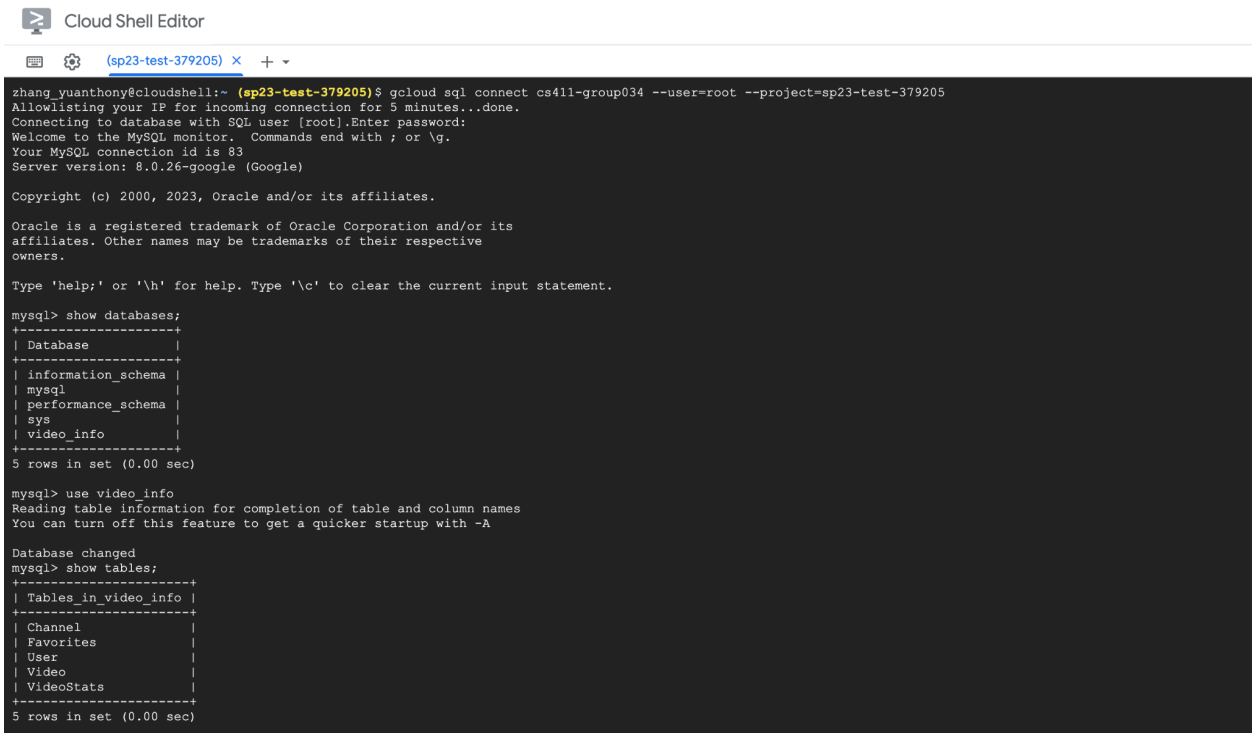
Implementation in GCP



DDL Commands

```
CREATE TABLE Video (
  VideoId CHAR(11) PRIMARY KEY,
  Title VARCHAR(100),
  PublishedDate VARCHAR(50),
  CategoryName VARCHAR(100),
  ChannelId CHAR(24),
  FOREIGN KEY (ChannelId) REFERENCES Channel(ChannelId)
);

CREATE TABLE VideoStats (
    VideoId CHAR(11) NOT NULL,
    TrendingDate VARCHAR(50) NOT NULL,
    ViewCount INT,
    Likes INT,
    Dislikes INT,
    PRIMARY KEY (VideoId, TrendingDate),
    FOREIGN KEY (VideoId) REFERENCES Video(VideoId)
```

```
);
/* These attributes are under VideoStats because each time a video is
trending it has different numbers for these in the dataset we are using */

CREATE TABLE Channel (
    ChannelId CHAR(24) PRIMARY KEY,
    ChannelTitle VARCHAR(255)
);

CREATE TABLE Favorites (
  WatchId CHAR(8) PRIMARY KEY,
  UserId CHAR(6),
  VideoId CHAR(11),
  VideoAddedDate VARCHAR(50)
);

CREATE TABLE  User (
  UserId CHAR(6) PRIMARY KEY,
  Username VARCHAR(100),
  Password VARCHAR(255),
  Email VARCHAR(255)
);
/* This is not one of the 4 main tables but we put it here as part of
setting up the database */
```

Table Counts

```
mysql> SELECT COUNT(*) FROM Channel;
+----------+
| COUNT(*) |
+----------+
|     1049 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT COUNT(*) FROM Favorites;
+----------+
| COUNT(*) |
+----------+
|     1001 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT COUNT(*) FROM Video;
+----------+
| COUNT(*) |
+----------+
|     1654 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT COUNT(*) FROM VideoStats;
+----------+
| COUNT(*) |
+----------+
|     7922 |
+----------+
1 row in set (0.01 sec)
```

Advanced Query

Query 1

This SQL query is used to count the number of favorite videos for each video category and sort the result by the number of favorite videos in descending order. As the dataset only contains 13 categories, we did not use the LIMIT clause to select the top 15 rows.

```sql
SELECT v.CategoryName, COUNT(*) AS num_favorites
FROM Video v JOIN Favorites f ON v.VideoId = f.VideoId
GROUP BY v.CategoryName
ORDER BY num_favorites DESC;
```

```
mysql> SELECT v.CategoryName, COUNT(*) AS num_favorites
    -> FROM Video v JOIN Favorites f ON v.VideoId = f.VideoId
    -> GROUP BY v.CategoryName
    -> ORDER BY num_favorites DESC;
+----------------------+---------------+
| CategoryName         | num_favorites |
+----------------------+---------------+
| Music                |           125 |
| Entertainment        |           121 |
| Sports               |            68 |
| Gaming               |            53 |
| People & Blogs       |            53 |
| Comedy               |            31 |
| News & Politics      |            30 |
| Howto & Style        |            29 |
| Science & Technology |            27 |
| Film & Animation     |            25 |
| Education            |             7 |
| Autos & Vehicles     |             6 |
| Travel & Events      |             4 |
+----------------------+---------------+
13 rows in set (0.10 sec)
```

Query 2

This query is finding the 15 most popular videos with at least 10,000 views that have a high proportion of likes relative to their total reactions based on the most recent record of each video information. Since there are many records for each video on different trending dates, this approach helps to ensure that the query results only contain the most recent information.

```
SELECT  v.Title, v.VideoId, v.ChannelId, n.Likes, n.Dislikes, n.ViewCount
FROM Video v JOIN (
      SELECT VideoId, ViewCount, Likes, Dislikes
      FROM VideoStats
      WHERE (VideoId, TrendingDate) IN (
         SELECT VideoId, MAX(TrendingDate)
         FROM VideoStats
         GROUP BY VideoId
      )
) AS n USING (VideoId)
WHERE n.Likes +n.Dislikes != 0 AND n.Likes/ (n.Likes +n.Dislikes) > 0.95
AND n.ViewCount >= 10000
ORDER BY n.ViewCount DESC
LIMIT 15;
```

```
mysql> SELECT  v.Title, v.VideoId, v.ChannelId, n.Likes, n.Dislikes, n.ViewCount
    -> FROM Video v JOIN (
    -> SELECT VideoId, ViewCount, Likes, Dislikes
    -> FROM VideoStats
    -> WHERE (VideoId, TrendingDate) IN (
    ->     SELECT VideoId, MAX(TrendingDate)
    ->     FROM VideoStats
    ->     GROUP BY VideoId
    -> )
    -> ) AS n USING (VideoId)
    -> WHERE n.Likes +n.Dislikes != 0 AND n.Likes/ (n.Likes +n.Dislikes) > 0.95 AND n.ViewCount >= 10000
    -> ORDER BY n.ViewCount DESC
    -> LIMIT 15;
```

Title	VideoId	ChannelId	Likes	Dislikes	ViewCount
BTS (Î≠©ÌÉÑÏÜåÎÖÑÎã®) 'Dynamite' Official MV	gdZLi9oWNZg	UC3IZKseVpdzPSBaWxBxundA	15735533	714194	232649205
BTS (Î≠©ÌÉÑÏÜåÎÖÑÎã®) 'Dynamite' Official Teaser	oxoWhyS9buA	UC3IZKseVpdzPSBaWxBxundA	6178664	158845	62496726
Ozuna x Karol G x Myke Towers - Caramelo Remix (Video Oficial)	KilybZma5vY	UCjIA3wwhi0QjSOXAZwOXbPA	926272	40744	45893190
BTS (Î≠©ÌÉÑÏÜåÎÖÑÎã®) 'Dynamite' Official MV (B-side)	BV2FdDmGiW0	UC3IZKseVpdzPSBaWxBxundA	5951286	97683	45596902
Drake - Laugh Now Cry Later (Official Music Video) ft. Lil Durk	JFm7YDVlqnI	UCQznUf1SjfDqx65hX3zRDiA	1656221	42515	45086708
DJ Khaled ft. Drake - POPSTAR (Official Music Video - Starring Justin Bieber)	3CxtK7-XtE0	UCrFB54bqp8sda4udJyNswlA	2109559	97497	43394819
Miley Cyrus - Midnight Sky (Official Video)	aS1no1myeTM	UCdI8evszfZvyAl2UVCypkTA	695129	35875	42667486
Stray Kids Back Door M/V	X-uJtV8ScYk	UCaO6TYtlC8U5ttz62hTrZgg	2077511	23985	39239499
Brawl Stars: Brawl Talk - Welcome to Starr Park! Gift Shop, Colette & More!	ME7qwm7oerQ	UCooVYzDxdwTtGYAkcPmOgOw	824753	28567	32114735
Last To Leave $800,000 Island Keeps It	NkE0AMGzpJY	UCX6OQ3DkcsbYNE6H8uQQuVA	2097381	35643	31801966
I Bought A Private Island	FbM1yi4mMMc	UCX6OQ3DkcsbYNE6H8uQQuVA	1801095	23693	30234876
ATEEZ(ÏóêÏù¥Ìã∞Ï¶à) - 'THANXX‚Äô Official MV	K7LY9Ta0eiY	UCQdq-lqPEq_yZ_wP_kuVB9Q	605389	5071	30059975
BTS Performs Dynamite | 2020 MTV VMAs	zJCdkOpU90g	UCxAICW_LdkfFYwTqTHHE0vg	2577092	58524	29442683
Better | Mamba Forever | Nike	C9I-W1eTCbk	UCUFgkRb0ZHc4Rpq15VRCICA	100490	4165	29422403
Sech, Daddy Yankee, J Balvin ft. Rosal√≠a, Farruko - Relaci√≥n Remix (Video Oficial)	XseIJg8Vyj0	UCte53PewI8_jYpBYn071MqQ	1081016	40222	28625426

```
15 rows in set (0.03 sec)
```

Indexing

Query 1

Initial result of EXPLAIN ANALYZE without using indexing

```
mysql> EXPLAIN ANALYZE SELECT v.CategoryName, COUNT(*) AS num_favorites FROM Video v JOIN Favorites f ON v.VideoId = f.VideoId GROUP BY v.CategoryName ORDER BY num_favorites DESC;
| EXPLAIN
| -> Sort: num_favorites DESC  (actual time=3.469..3.470 rows=13 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.002 rows=13 loops=1)
        -> Aggregate using temporary table  (actual time=3.437..3.439 rows=13 loops=1)
            -> Nested loop inner join  (cost=448.65 rows=992) (actual time=0.065..3.055 rows=579 loops=1)
                -> Filter: (f.VideoId is not null)  (cost=101.45 rows=992) (actual time=0.040..0.386 rows=1001 loops=1)
                    -> Table scan on f  (cost=101.45 rows=992) (actual time=0.039..0.304 rows=1001 loops=1)
                -> Single-row index lookup on v using PRIMARY (VideoId=f.VideoId)  (cost=0.25 rows=1) (actual time=0.002..0.003 rows=1 loops=1001)
 |
1 row in set (0.00 sec)
```

Index Design 1 - CREATE INDEX add_categoryname_idx ON Video(CategoryName)

After using the index, the query still performs a table scan on the temporary table and applies the same aggregate function to it. The performance improved slightly as it took a little less time but overall this change might be negligible. The reason this may not have helped is that we have very few categories to begin with so indexing on them doesn't really make a difference. Also we are not filtering any categories out.

```
mysql> EXPLAIN ANALYZE SELECT v.CategoryName, COUNT(*) AS num_favorites FROM Video v JOIN Favorites f ON v.VideoId = f.VideoId GROUP BY v.CategoryName ORDER BY num_favorites DESC;
| EXPLAIN
| -> Sort: num_favorites DESC  (actual time=3.374..3.375 rows=13 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.002 rows=13 loops=1)
        -> Aggregate using temporary table  (actual time=3.353..3.355 rows=13 loops=1)
            -> Nested loop inner join  (cost=448.65 rows=992) (actual time=0.068..2.935 rows=579 loops=1)
                -> Filter: (f.VideoId is not null)  (cost=101.45 rows=992) (actual time=0.041..0.369 rows=1001 loops=1)
                    -> Table scan on f  (cost=101.45 rows=992) (actual time=0.040..0.292 rows=1001 loops=1)
                -> Single-row index lookup on v using PRIMARY (VideoId=f.VideoId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1001)
 |
1 row in set (0.01 sec)
```

Index Design 2 - CREATE INDEX add_videoid_idx ON Video(VideoId)

After using the index, the query plan remained the same, but the overall query performance improved a little, as evidenced by the reduced actual time for the "Aggregate using temporary table”.

```
mysql> CREATE INDEX add_videoid_idx ON Video(VideoId);
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN ANALYZE SELECT v.CategoryName, COUNT(*) AS num_favorites FROM Video v JOIN Favorites f ON v.VideoId = f.VideoId GROUP BY v.CategoryName ORDER BY num_favorites DESC;
+----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------+
| EXPLAIN


                                                                    |
+----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------+
| -> Sort: num_favorites DESC  (actual time=3.412..3.412 rows=13 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.002 rows=13 loops=1)
        -> Aggregate using temporary table  (actual time=3.375..3.377 rows=13 loops=1)
            -> Nested loop inner join  (cost=448.65 rows=992) (actual time=0.068..2.985 rows=579 loops=1)
                -> Filter: (f.VideoId is not null)  (cost=101.45 rows=992) (actual time=0.041..0.382 rows=1001 loops=1)
                    -> Table scan on f  (cost=101.45 rows=992) (actual time=0.039..0.302 rows=1001 loops=1)
                -> Single-row index lookup on v using PRIMARY (VideoId=f.VideoId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1001)
 |
+----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------+
1 row in set (0.01 sec)
```

Index Design 3 - CREATE INDEX add_videoid_idx ON Favorites(VideoId)

After using the index, the query was able to utilize an index scan on the table being joined, then the query was able to use index scans to quickly locate the relevant data rows, which reduced the number of times the table needed to be scanned. This resulted in an improvement in query performance, as evidenced by the decrease in actual time from **3.469** to **2.827** milliseconds

```
mysql> CREATE INDEX add_videoid_idx ON Favorites(VideoId);
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN ANALYZE SELECT v.CategoryName, COUNT(*) AS num_favorites FROM Video v JOIN Favorites f ON v.VideoId = f.VideoId GROUP BY v.CategoryName ORDER BY num_favorites DESC;
+----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------+
| EXPLAIN


                                                                                           |
+----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------+
| -> Sort: num_favorites DESC  (actual time=2.827..2.827 rows=13 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.002 rows=13 loops=1)
        -> Aggregate using temporary table  (actual time=2.804..2.806 rows=13 loops=1)
            -> Nested loop inner join  (cost=448.65 rows=992) (actual time=0.057..2.452 rows=579 loops=1)
                -> Filter: (f.VideoId is not null)  (cost=101.45 rows=992) (actual time=0.037..0.331 rows=1001 loops=1)
                    -> Index scan on f using add_videoid_idx  (cost=101.45 rows=992) (actual time=0.036..0.257 rows=1001 loops=1)
                -> Single-row index lookup on v using PRIMARY (VideoId=f.VideoId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1001)
 |
+----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

Query 2

Initial result of EXPLAIN ANALYZE without using indexing

```
mysql> EXPLAIN ANALYZE SELECT  v.Title, v.VideoId, v.ChannelId, n.Likes, n.Dislikes, n.ViewCount FROM Video v JOIN ( SELECT VideoId, ViewCount, Likes, Dislikes FROM VideoStats WHERE (VideoId, TrendingDate) IN (     SE
LECT VideoId, MAX(TrendingDate)     FROM VideoStats     GROUP BY VideoId ) ) AS n USING (VideoId)  WHERE n.Likes +n.Dislikes != 0 AND n.Likes/ (n.Likes +n.Dislikes) > 0.95 AND n.ViewCount >= 10000 ORDER BY n.ViewCount
 DESC LIMIT 15;
+--------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                         |
+--------------------------------------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (cost=2108.67 rows=15) (actual time=32.731..32.765 rows=15 loops=1)
    -> Nested loop inner join  (cost=2108.67 rows=7357) (actual time=32.731..32.763 rows=15 loops=1)
        -> Sort: VideoStats.ViewCount DESC  (cost=759.95 rows=7357) (actual time=32.711..32.713 rows=15 loops=1)
            -> Filter: (((VideoStats.Likes + VideoStats.Dislikes) <> 0) and ((VideoStats.Likes / (VideoStats.Likes + VideoStats.Dislikes)) > 0.95) and (VideoStats.ViewCount >= 10000) and <in_optimizer>((VideoStats.Vid
eoId,VideoStats.TrendingDate),(VideoStats.VideoId,VideoStats.TrendingDate) in (select #3)))  (cost=759.95 rows=7357) (actual time=15.069..32.114 rows=1364 loops=1)
                -> Table scan on VideoStats  (cost=759.95 rows=7357) (actual time=0.049..2.950 rows=7922 loops=1)
                -> Select #3 (subquery in condition; run only once)
                    -> Filter: ((VideoStats.VideoId = `<materialized_subquery>`.VideoId) and (VideoStats.TrendingDate = `<materialized_subquery>`.`MAX(TrendingDate)`))  (actual time=0.001..0.001 rows=0 loops=6631)
                        -> Limit: 1 row(s)  (actual time=0.001..0.001 rows=0 loops=6631)
                            -> Index lookup on <materialized_subquery> using <auto_distinct_key> (VideoId=VideoStats.VideoId, MAX(TrendingDate)=VideoStats.TrendingDate)  (actual time=0.000..0.000 rows=0 loops=6631)
                                -> Materialize with deduplication  (cost=882.75..882.75 rows=1549) (actual time=21.387..21.387 rows=1654 loops=1)
                                    -> Index range scan on VideoStats using index_for_group_by(PRIMARY)  (cost=727.85 rows=1549) (actual time=0.019..13.844 rows=1654 loops=1)
        -> Single-row index lookup on v using PRIMARY (VideoId=VideoStats.VideoId)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=15)
 |
+--------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.04 sec)
```

Index Design 1 - CREATE INDEX add_viewcount_idx ON VideoStats(ViewCount)

We created an index on column **ViewCount** of table **VideoStats.** The original query performs a table scan on VideoStats and uses a nested loop join to join it with the Video table. After adding an index on the ViewCount column, the optimizer changes the query plan to use the index range scan instead of a table scan, which significantly reduces the number of rows being scanned. This results in a much lower cost and faster execution time. Additionally, the filter condition is optimized to use the index on ViewCount and is applied earlier in the query execution, resulting in a further reduction in the number of rows being processed.

```
mysql> CREATE INDEX add_viewcount_idx ON VideoStats(ViewCount);
Query OK, 0 rows affected (0.17 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN ANALYZE SELECT  v.Title, v.VideoId, v.ChannelId, n.Likes, n.Dislikes, n.ViewCount FROM Video v JOIN ( SELECT VideoId, ViewCount, Likes, Dislikes FROM VideoStats WHERE (VideoId, TrendingDate) IN (     SE
LECT VideoId, MAX(TrendingDate)     FROM VideoStats     GROUP BY VideoId ) ) AS n USING (VideoId)  WHERE n.Likes +n.Dislikes != 0 AND n.Likes/ (n.Likes +n.Dislikes) > 0.95 AND n.ViewCount >= 10000 ORDER BY n.ViewCount
 DESC LIMIT 15;
+--------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                         |
+--------------------------------------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (cost=2047.25 rows=15) (actual time=14.754..14.975 rows=15 loops=1)
    -> Nested loop inner join  (cost=2047.25 rows=3678) (actual time=14.752..14.971 rows=15 loops=1)
        -> Filter: (((VideoStats.Likes + VideoStats.Dislikes) <> 0) and ((VideoStats.Likes / (VideoStats.Likes + VideoStats.Dislikes)) > 0.95) and <in_optimizer>((VideoStats.VideoId,VideoStats.TrendingDate),(VideoStat
s.VideoId,VideoStats.TrendingDate) in (select #3)))  (cost=759.95 rows=3678) (actual time=14.734..14.911 rows=15 loops=1)
            -> Index range scan on VideoStats using add_viewcount_idx, with index condition: (VideoStats.ViewCount >= 10000)  (cost=759.95 rows=3678) (actual time=0.497..0.518 rows=68 loops=1)
            -> Select #3 (subquery in condition; run only once)
                -> Filter: ((VideoStats.VideoId = `<materialized_subquery>`.VideoId) and (VideoStats.TrendingDate = `<materialized_subquery>`.`MAX(TrendingDate)`))  (actual time=0.002..0.002 rows=0 loops=55)
                    -> Limit: 1 row(s)  (actual time=0.001..0.001 rows=0 loops=55)
                        -> Index lookup on <materialized_subquery> using <auto_distinct_key> (VideoId=VideoStats.VideoId, MAX(TrendingDate)=VideoStats.TrendingDate)  (actual time=0.001..0.001 rows=0 loops=55)
                            -> Materialize with deduplication  (cost=882.75..882.75 rows=1549) (actual time=14.296..14.296 rows=1654 loops=1)
                                -> Index range scan on VideoStats using index_for_group_by(PRIMARY)  (cost=727.85 rows=1549) (actual time=0.021..13.178 rows=1654 loops=1)
        -> Single-row index lookup on v using PRIMARY (VideoId=VideoStats.VideoId)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=15)
 |
+--------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

Index Design 2 - CREATE INDEX add_likes_idx ON VideoStats(Likes)

We then created an index on column **Likes** of table **VideoStats**. From the EXPLAIN ANALYZE output, we can see that adding the **add_likes_idx** did not have any noticeable impact on the cost or the actual execution time of the query. This may be because the query mostly looks for the videos and then checks their like to dislike ratio, the likes aren't what we are searching through though so adding an index on them doesn't make a difference.

```
mysql> CREATE INDEX add_likes_idx ON VideoStats(Likes);
Query OK, 0 rows affected (0.17 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN ANALYZE SELECT  v.Title, v.VideoId, v.ChannelId, n.Likes, n.Dislikes, n.ViewCount FROM Video v JOIN ( SELECT VideoId, ViewCount, Likes, Dislikes FROM VideoStats WHERE (VideoId, TrendingDate) IN (     SE
LECT VideoId, MAX(TrendingDate)     FROM VideoStats     GROUP BY VideoId ) ) AS n USING (VideoId)  WHERE n.Likes +n.Dislikes != 0 AND n.Likes/ (n.Likes +n.Dislikes) > 0.95 AND n.ViewCount >= 10000 ORDER BY n.ViewCount
 DESC LIMIT 15;
+--------------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                                     |
+--------------------------------------------------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (cost=2108.67 rows=15) (actual time=35.444..35.477 rows=15 loops=1)
    -> Nested loop inner join  (cost=2108.67 rows=7357) (actual time=35.443..35.475 rows=15 loops=1)
        -> Sort: VideoStats.ViewCount DESC  (cost=759.95 rows=7357) (actual time=35.423..35.424 rows=15 loops=1)
            -> Filter: (((VideoStats.Likes + VideoStats.Dislikes) <> 0) and ((VideoStats.Likes / (VideoStats.Likes + VideoStats.Dislikes)) > 0.95) and (VideoStats.ViewCount >= 10000) and <in_optimizer>((VideoStats.Vid
eoId,VideoStats.TrendingDate),(VideoStats.VideoId,VideoStats.TrendingDate) in (select #3)))  (cost=759.95 rows=7357) (actual time=16.816..34.801 rows=1364 loops=1)
                -> Table scan on VideoStats  (cost=759.95 rows=7357) (actual time=0.050..3.046 rows=7922 loops=1)
                -> Select #3 (subquery in condition; run only once)
                    -> Filter: ((VideoStats.VideoId = `<materialized_subquery>`.VideoId) and (VideoStats.TrendingDate = `<materialized_subquery>`.`MAX(TrendingDate)`))  (actual time=0.001..0.001 rows=0 loops=6631)
                        -> Limit: 1 row(s)  (actual time=0.001..0.001 rows=0 loops=6631)
                            -> Index lookup on <materialized_subquery> using <auto_distinct_key> (VideoId=VideoStats.VideoId, MAX(TrendingDate)=VideoStats.TrendingDate)  (actual time=0.001..0.001 rows=0 loops=6631)
                                -> Materialize with deduplication  (cost=882.75..882.75 rows=1549) (actual time=23.566..23.566 rows=1654 loops=1)
                                    -> Index range scan on VideoStats using index_for_group_by(PRIMARY)  (cost=727.85 rows=1549) (actual time=0.019..15.406 rows=1654 loops=1)
        -> Single-row index lookup on v using PRIMARY (VideoId=VideoStats.VideoId)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=15)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.04 sec)
```

Index Design 3 - CREATE INDEX add_dislikes_idx ON VideoStats(Dislikes)

After that, we tried to use **add_dislikes_idx.** Comparing the two EXPLAIN ANALYZE results for the query after and before adding this index on VideoStats.VideoId, there is no noticeable change in the execution plan, and the cost and actual time for each step remain almost the same. This is for the same reasons as Index Design 2.

```
mysql> CREATE INDEX add_dislikes_idx ON VideoStats(Dislikes);
Query OK, 0 rows affected (0.17 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN ANALYZE SELECT  v.Title, v.VideoId, v.ChannelId, n.Likes, n.Dislikes, n.ViewCount FROM Video v JOIN ( SELECT VideoId, ViewCount, Likes, Dislikes FROM VideoStats WHERE (VideoId, TrendingDate) IN (     SE
LECT VideoId, MAX(TrendingDate)     FROM VideoStats     GROUP BY VideoId ) ) AS n USING (VideoId)  WHERE n.Likes +n.Dislikes != 0 AND n.Likes/ (n.Likes +n.Dislikes) > 0.95 AND n.ViewCount >= 10000 ORDER BY n.ViewCount
 DESC LIMIT 15;
+--------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                        |
+--------------------------------------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (cost=2108.67 rows=15) (actual time=32.373..32.408 rows=15 loops=1)
    -> Nested loop inner join  (cost=2108.67 rows=7357) (actual time=32.372..32.406 rows=15 loops=1)
        -> Sort: VideoStats.ViewCount DESC  (cost=759.95 rows=7357) (actual time=32.350..32.351 rows=15 loops=1)
            -> Filter: (((VideoStats.Likes + VideoStats.Dislikes) <> 0) and ((VideoStats.Likes / (VideoStats.Likes + VideoStats.Dislikes)) > 0.95) and (VideoStats.ViewCount >= 10000) and <in_optimizer>((VideoStats.Vid
eoId,VideoStats.TrendingDate),(VideoStats.VideoId,VideoStats.TrendingDate) in (select #3)))  (cost=759.95 rows=7357) (actual time=14.503..31.752 rows=1364 loops=1)
                -> Table scan on VideoStats  (cost=759.95 rows=7357) (actual time=0.060..3.024 rows=7922 loops=1)
                -> Select #3 (subquery in condition; run only once)
                    -> Filter: ((VideoStats.VideoId = `<materialized_subquery>`.VideoId) and (VideoStats.TrendingDate = `<materialized_subquery>`.`MAX(TrendingDate)`))  (actual time=0.001..0.001 rows=0 loops=6631)
                        -> Limit: 1 row(s)  (actual time=0.001..0.001 rows=0 loops=6631)
                            -> Index lookup on <materialized_subquery> using <auto_distinct_key> (VideoId=VideoStats.VideoId, MAX(TrendingDate)=VideoStats.TrendingDate)  (actual time=0.001..0.001 rows=0 loops=6631)
                                -> Materialize with deduplication  (cost=882.75..882.75 rows=1549) (actual time=20.983..20.983 rows=1654 loops=1)
                                    -> Index range scan on VideoStats using index_for_group_by(PRIMARY)  (cost=727.85 rows=1549) (actual time=0.022..13.326 rows=1654 loops=1)
        -> Single-row index lookup on v using PRIMARY (VideoId=VideoStats.VideoId)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=15)
 |
+--------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.03 sec)
```

<u>Index Design 4 - CREATE INDEX add_videoid_idx ON Video(VideoId)</u>

Based on the result provided, it seems that adding this index did not help to speed up the query, and actually made it slower. This can be seen from the fact that the execution time increased from **32.763** to **40.926** milliseconds after adding the index. The longer time might be attributed to a lot of factors but the reason this index didn't help is because our query wasn't looking for a specific video id, so indexing makes no improvements because the id itself doesn't matter.

```
mysql> CREATE INDEX add_videoid_idx ON Video(VideoId);
Query OK, 0 rows affected (0.07 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN ANALYZE SELECT  v.Title, v.VideoId, v.ChannelId, n.Likes, n.Dislikes, n.ViewCount FROM Video v JOIN ( SELECT VideoId, ViewCount, Likes, Dislikes FROM VideoStats WHERE (VideoId, TrendingDate) IN (     SE
LECT VideoId, MAX(TrendingDate)     FROM VideoStats     GROUP BY VideoId ) ) AS n USING (VideoId)  WHERE n.Likes +n.Dislikes != 0 AND n.Likes/ (n.Likes +n.Dislikes) > 0.95 AND n.ViewCount >= 10000 ORDER BY n.ViewCount
 DESC LIMIT 15;
+--------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                        |
+--------------------------------------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (cost=2108.67 rows=15) (actual time=40.874..40.926 rows=15 loops=1)
    -> Nested loop inner join  (cost=2108.67 rows=7357) (actual time=40.873..40.924 rows=15 loops=1)
        -> Sort: VideoStats.ViewCount DESC  (cost=759.95 rows=7357) (actual time=40.841..40.843 rows=15 loops=1)
            -> Filter: (((VideoStats.Likes + VideoStats.Dislikes) <> 0) and ((VideoStats.Likes / (VideoStats.Likes + VideoStats.Dislikes)) > 0.95) and (VideoStats.ViewCount >= 10000) and <in_optimizer>((VideoStats.Vid
eoId,VideoStats.TrendingDate),(VideoStats.VideoId,VideoStats.TrendingDate) in (select #3)))  (cost=759.95 rows=7357) (actual time=16.491..40.071 rows=1364 loops=1)
                -> Table scan on VideoStats  (cost=759.95 rows=7357) (actual time=0.050..4.032 rows=7922 loops=1)
                -> Select #3 (subquery in condition; run only once)
                    -> Filter: ((VideoStats.VideoId = `<materialized_subquery>`.VideoId) and (VideoStats.TrendingDate = `<materialized_subquery>`.`MAX(TrendingDate)`))  (actual time=0.001..0.001 rows=0 loops=6631)
                        -> Limit: 1 row(s)  (actual time=0.001..0.001 rows=0 loops=6631)
                            -> Index lookup on <materialized_subquery> using <auto_distinct_key> (VideoId=VideoStats.VideoId, MAX(TrendingDate)=VideoStats.TrendingDate)  (actual time=0.001..0.001 rows=0 loops=6631)
                                -> Materialize with deduplication  (cost=882.75..882.75 rows=1549) (actual time=26.426..26.426 rows=1654 loops=1)
                                    -> Index range scan on VideoStats using index_for_group_by(PRIMARY)  (cost=727.85 rows=1549) (actual time=0.021..14.830 rows=1654 loops=1)
        -> Single-row index lookup on v using PRIMARY (VideoId=VideoStats.VideoId)  (cost=0.25 rows=1) (actual time=0.005..0.005 rows=1 loops=15)
 |
+--------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.04 sec)
```

Overall, we see that the overall query time is reduced and the query can return the results much faster after using **add_viewcount_idx**. This may be because this index could filter out rows that do not meet the condition (VideoStats.ViewCount >= 10000) before joining with other tables, which significantly reduces the number of rows being joined and thus improves the query performance.