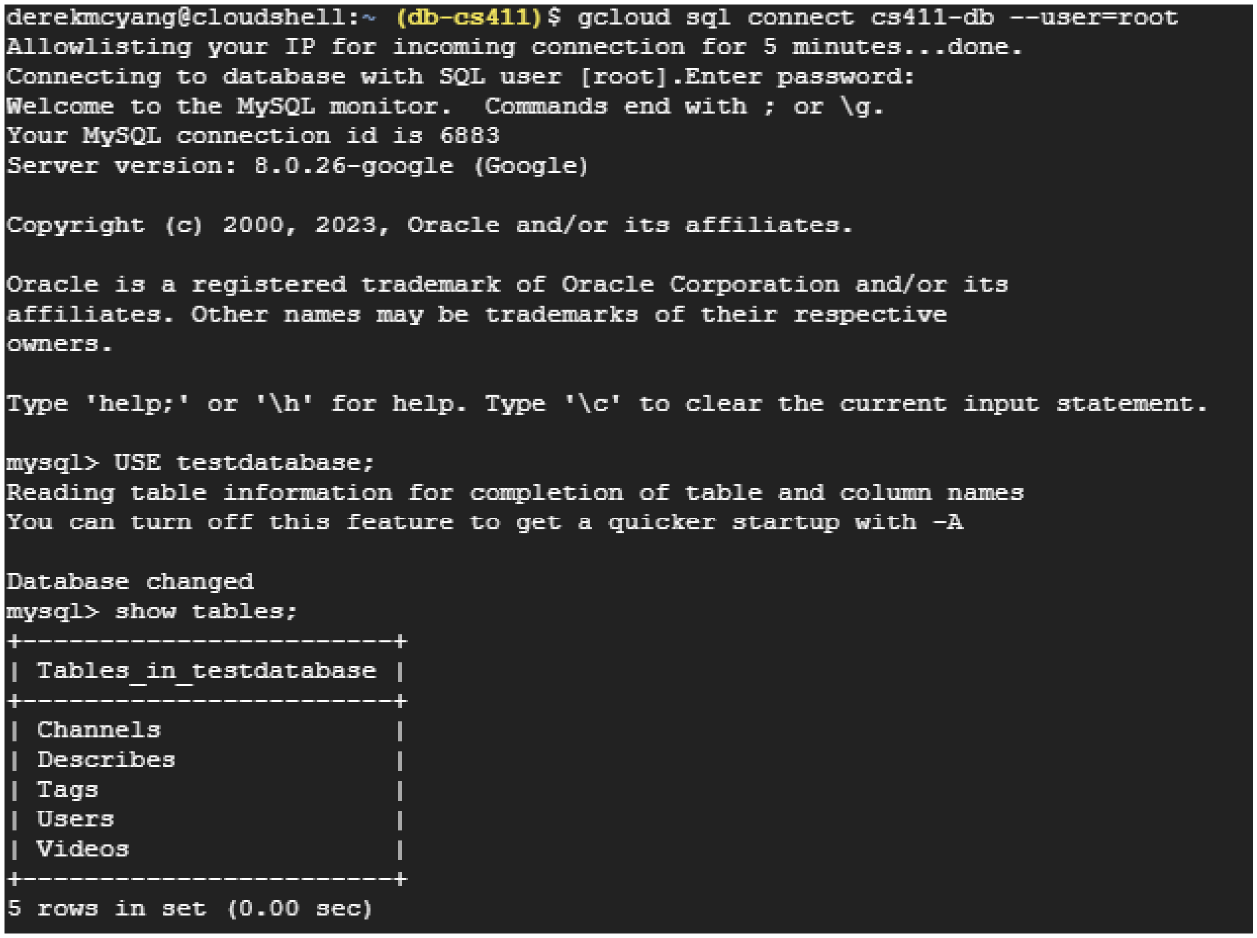
Design

We implemented the database on Google Cloud Platform, below shows the screen shot that we logged in with Gcloud and accessed the tables within that database.



```
derekmcyang@cloudshell:~ (db-cs411)$ gcloud sql connect cs411-db --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 6883
Server version: 8.0.26-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 testdatabase;
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_testdatabase |
+------------------------+
| Channels               |
| Describes              |
| Tags                   |
| Users                  |
| Videos                 |
+------------------------+
5 rows in set (0.00 sec)
```

Below shows our DDL commands that we used to create the tables:

- Videos

```
CREATE TABLE Videos(
    VideoID CHAR(64) NOT NULL,
    VideoTitle CHAR(128) NOT NULL,
    Region CHAR(64),
    Length FLOAT(4),
    Views BIGINT,
    Dislikes BIGINT,
    Likes BIGINT,
    PublishDate DATE,
    NumComments BIGINT,
    ChannelID CHAR(64),
    UserID CHAR(64),
    PRIMARY KEY (VideoID, ChannelID, UserID),
    FOREIGN KEY (ChannelID) REFERENCES Channels(ChannelID), FOREIGN KEY
(UserID) REFERENCES Users(UserID) ON DELETE CASCADE
);
```

- Users

```
CREATE TABLE Users(
    `UserID` CHAR(64) NOT NULL,
    `Nickname` CHAR(128),
    `AccountCreateDate` datetime,
    primary key(`UserID`)
);
```

- Tags

```
CREATE TABLE Tags(
    `TagsID` CHAR(64),
    `Name` TEXT NOT NULL,
    `NumVid` BIGINT,
    `NumChannel` BIGINT,
    primary key(`TagsID`)
);
```

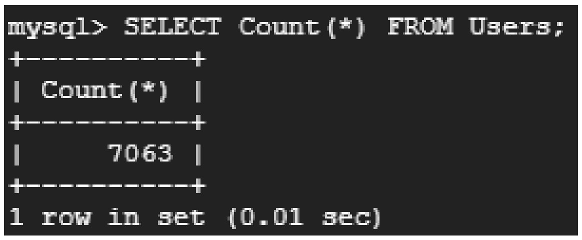
- Describes

```
CREATE TABLE Describes(
    TagsID CHAR(64) NOT NULL,
    VideoID CHAR(64) NOT NULL,
    ChannelID CHAR(64) NOT NULL,
    UserID CHAR(64) NOT NULL,
    PRIMARY KEY (VideoID, ChannelID, UserID, TagsID),
    FOREIGN KEY (ChannelID) REFERENCES Channels(ChannelID) ON DELETE
CASCADE,
    FOREIGN KEY (UserID) REFERENCES Users(UserID) ON DELETE CASCADE,
    FOREIGN KEY (VideoID) REFERENCES Videos(VideoID) ON DELETE CASCADE,
    FOREIGN KEY (TagsID) REFERENCES Tags(TagsID) ON DELETE CASCADE
);
```

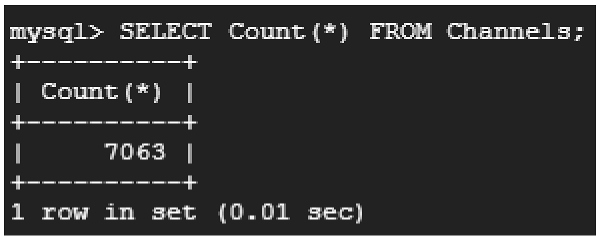
- Channels

```
CREATE TABLE Channels(
    `ChannelID` char(64) NOT NULL,
    `ChannelName` CHAR(128) NOT NULL,
    `Region` CHAR(64),
    `CreationDate` DATE,
    `UserID` CHAR(64) NOT NULL,
    PRIMARY KEY (ChannelID, UserID),
    FOREIGN KEY (UserID) REFERENCES Users(UserID) ON DELETE CASCADE
);
```

We inserted 7063 rows into ***Channels*** and ***Users***

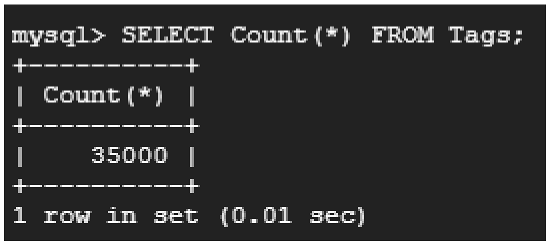


```
mysql> SELECT Count(*) FROM Users;
+----------+
| Count(*) |
+----------+
|     7063 |
+----------+
1 row in set (0.01 sec)
```



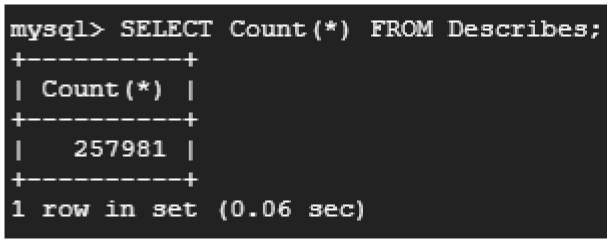
```
mysql> SELECT Count(*) FROM Channels;
+----------+
| Count(*) |
+----------+
|     7063 |
+----------+
1 row in set (0.01 sec)
```

We inserted 35000 rows into ***Tags***



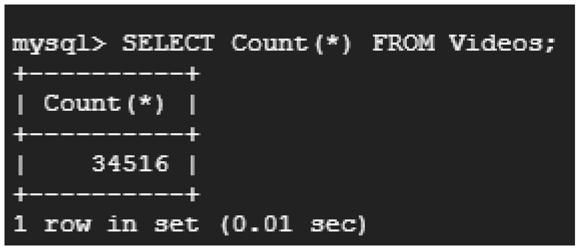
```
mysql> SELECT Count(*) FROM Tags;
+----------+
| Count(*) |
+----------+
|    35000 |
+----------+
1 row in set (0.01 sec)
```

We inserted 257981 rows into ***Describes***



```
mysql> SELECT Count(*) FROM Describes;
+----------+
| Count(*) |
+----------+
|   257981 |
+----------+
1 row in set (0.06 sec)
```

We inserted 34516 rows into ***Videos***

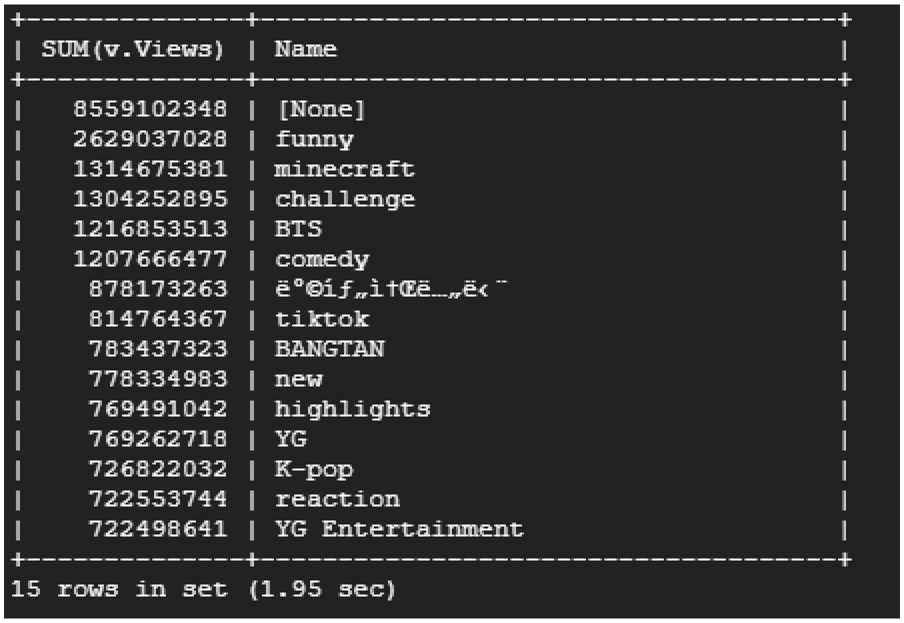


```
mysql> SELECT Count(*) FROM Videos;
+----------+
| Count(*) |
+----------+
|    34516 |
+----------+
1 row in set (0.01 sec)
```

Advanced Queries

1. **SELECT SUM(v.Views),t.Name**
   FROM Describes d JOIN Videos v ON(d.VideoID=v.VideoID) JOIN Tags t ON
   (d.TagsID=t.TagsID)
   GROUP BY d.TagsID,t.Name
   ORDER BY SUM(v.Views) DESC
   LIMIT 15;

This query returns the total number of views on videos under each tag. We do this by first joining our Describes table with our Videos table and Tags table, simply on the condition that the TagsID and VideoID are present in Describes. This is because it implies that the video possesses the corresponding tag. Then, we group by tags, and then sum up all views of videos in each group. This ultimately leaves us with the total number of views for each tag.
Below shows the result of said query limited to a length of 15.

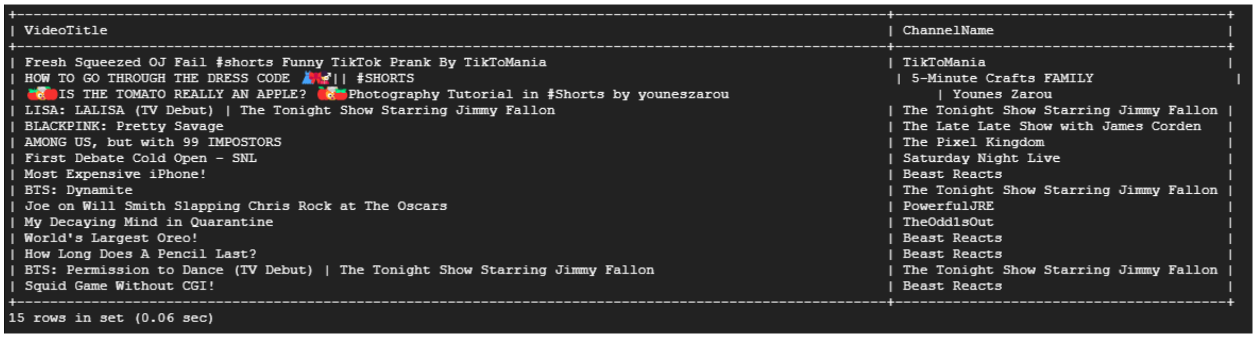


```
+--------------+--------------------------------------+
| SUM(v.Views) | Name                                 |
+--------------+--------------------------------------+
|   8559102348 | [None]                               |
|   2629037028 | funny                                |
|   1314675381 | minecraft                            |
|   1304252895 | challenge                            |
|   1216853513 | BTS                                  |
|   1207666477 | comedy                               |
|    878173263 | ë°©íƒ„ì†Œë…„ë‹¨                         |
|    814764367 | tiktok                               |
|    783437323 | BANGTAN                              |
|    778334983 | new                                  |
|    769491042 | highlights                           |
|    769262718 | YG                                   |
|    726822032 | K-pop                                |
|    722553744 | reaction                             |
|    722498641 | YG Entertainment                     |
+--------------+--------------------------------------+
15 rows in set (1.95 sec)
```

2. **SELECT v.VideoTitle, c.ChannelName**
   FROM Videos v JOIN Channels c ON (c.ChannelID = v.ChannelID)
   WHERE v.VideoID IN (SELECT VideoID FROM Tags t JOIN Describes d ON (d.TagsID=t.TagsID) WHERE t.Name LIKE "funny") AND v.Likes > 100
   ORDER BY v.Views DESC
   LIMIT 15;

This query returns the most popular videos by view count along with their corresponding channel name. Additionally, the videos must possess a “funny” tag as well as a like count of over 100. We did this by joining videos with their corresponding channel from the Videos and Channels tables, and then adding a WHERE condition checking that the video possesses a funny tag (which is done by checking for VideoID in the Describe table with corresponding TagID where the tag has the name “funny”). We then added the additional condition that the video’s likes must be greater than 100, and ordered the rows in descending number of views.

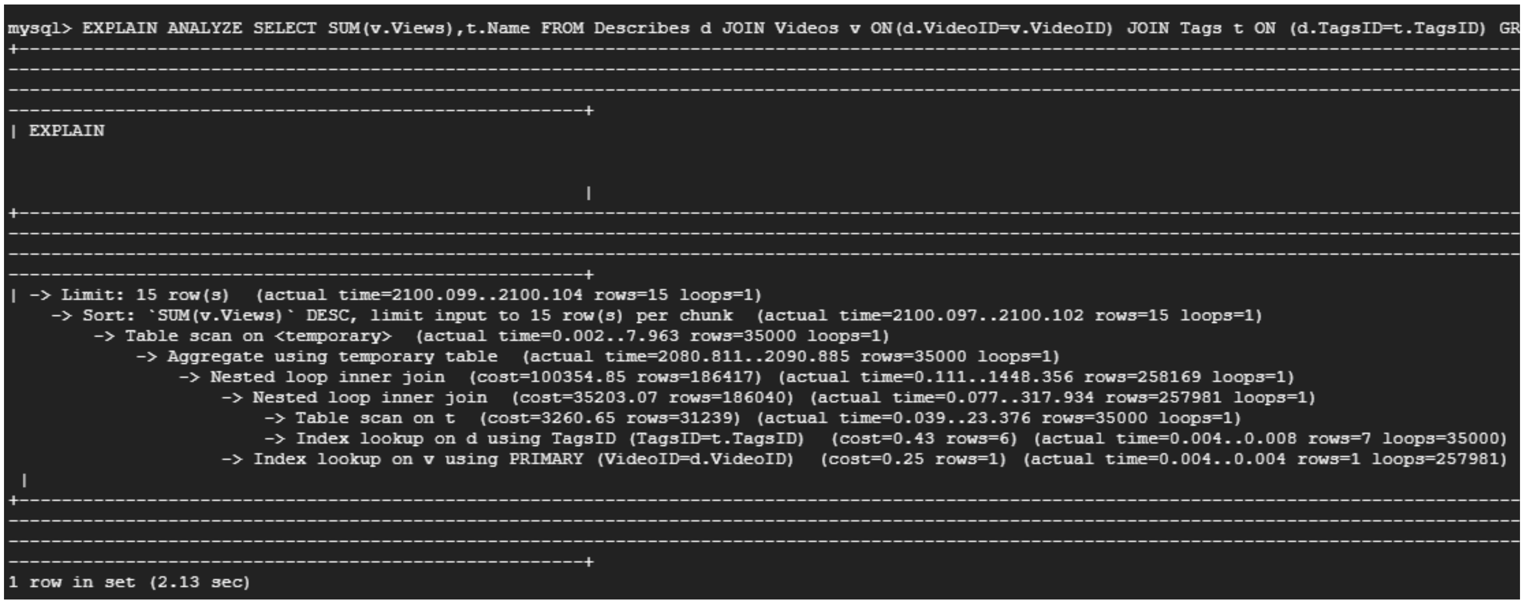
Below shows the result of said query limited to a length of 15.



```
+------------------------------------------------------------------------------------------------+-------------------------------------+
| VideoTitle                                                                                     | ChannelName                         |
+------------------------------------------------------------------------------------------------+-------------------------------------+
| Fresh Squeezed OJ Fail #shorts Funny TikTok Prank By TikToMania                                | TikToMania                          |
| HOW TO GO THROUGH THE DRESS CODE 👗🎀|| #SHORTS                                                 | 5-Minute Crafts FAMILY              |
| 🍅🍅IS THE TOMATO REALLY AN APPLE? 🍅🍅Photography Tutorial in #Shorts by youneszarou              | Younes Zarou                        |
| LISA: LALISA (TV Debut) | The Tonight Show Starring Jimmy Fallon                               | The Tonight Show Starring Jimmy Fallon |
| BLACKPINK: Pretty Savage                                                                       | The Late Late Show with James Corden |
| AMONG US, but with 99 IMPOSTORS                                                                | The Pixel Kingdom                   |
| First Debate Cold Open - SNL                                                                   | Saturday Night Live                 |
| Most Expensive iPhone!                                                                         | Beast Reacts                        |
| BTS: Dynamite                                                                                  | The Tonight Show Starring Jimmy Fallon |
| Joe on Will Smith Slapping Chris Rock at The Oscars                                            | PowerfulJRE                         |
| My Decaying Mind in Quarantine                                                                 | TheOdd1sOut                         |
| World's Largest Oreo!                                                                          | Beast Reacts                        |
| How Long Does A Pencil Last?                                                                   | Beast Reacts                        |
| BTS: Permission to Dance (TV Debut) | The Tonight Show Starring Jimmy Fallon                   | The Tonight Show Starring Jimmy Fallon |
| Squid Game Without CGI!                                                                        | Beast Reacts                        |
+------------------------------------------------------------------------------------------------+-------------------------------------+
15 rows in set (0.06 sec)
```

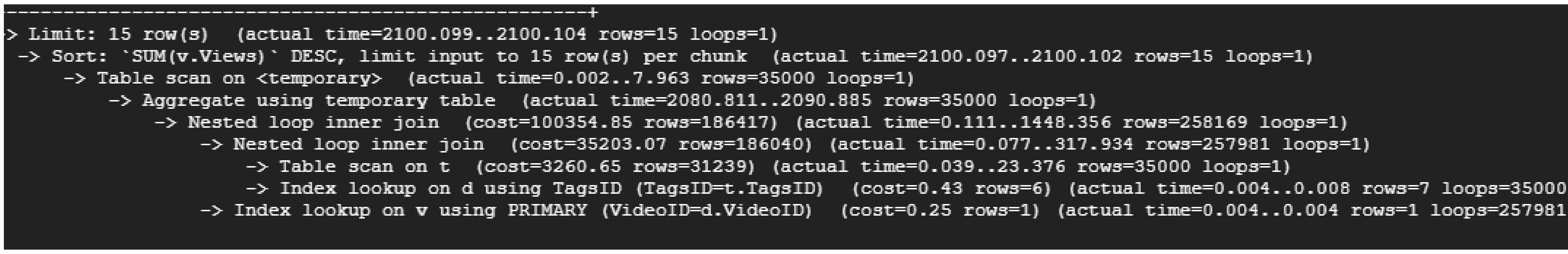
Indexing

We ran explain analyze on the first query with the below result.



```
mysql> EXPLAIN ANALYZE SELECT SUM(v.Views),t.Name FROM Describes d JOIN Videos v ON(d.VideoID=v.VideoID) JOIN Tags t ON (d.TagsID=t.TagsID) GR
+----------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------+
| EXPLAIN


                                                  |
+----------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=2100.099..2100.104 rows=15 loops=1)
    -> Sort: `SUM(v.Views)` DESC, limit input to 15 row(s) per chunk  (actual time=2100.097..2100.102 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=0.002..7.963 rows=35000 loops=1)
            -> Aggregate using temporary table  (actual time=2080.811..2090.885 rows=35000 loops=1)
                -> Nested loop inner join  (cost=100354.85 rows=186417) (actual time=0.111..1448.356 rows=258169 loops=1)
                    -> Nested loop inner join  (cost=35203.07 rows=186040) (actual time=0.077..317.934 rows=257981 loops=1)
                        -> Table scan on t  (cost=3260.65 rows=31239) (actual time=0.039..23.376 rows=35000 loops=1)
                        -> Index lookup on d using TagsID (TagsID=t.TagsID)  (cost=0.43 rows=6) (actual time=0.004..0.008 rows=7 loops=35000)
                    -> Index lookup on v using PRIMARY (VideoID=d.VideoID)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=257981)
 |
+----------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------+
1 row in set (2.13 sec)
```

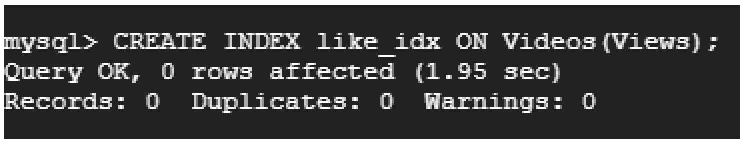
Below is a larger screen shot focused on the analysis



```
--------------------------------------------------+
> Limit: 15 row(s)  (actual time=2100.099..2100.104 rows=15 loops=1)
 -> Sort: `SUM(v.Views)` DESC, limit input to 15 row(s) per chunk  (actual time=2100.097..2100.102 rows=15 loops=1)
     -> Table scan on <temporary>  (actual time=0.002..7.963 rows=35000 loops=1)
         -> Aggregate using temporary table  (actual time=2080.811..2090.885 rows=35000 loops=1)
             -> Nested loop inner join  (cost=100354.85 rows=186417) (actual time=0.111..1448.356 rows=258169 loops=1)
                 -> Nested loop inner join  (cost=35203.07 rows=186040) (actual time=0.077..317.934 rows=257981 loops=1)
                     -> Table scan on t  (cost=3260.65 rows=31239) (actual time=0.039..23.376 rows=35000 loops=1)
                     -> Index lookup on d using TagsID (TagsID=t.TagsID)  (cost=0.43 rows=6) (actual time=0.004..0.008 rows=7 loops=35000
                 -> Index lookup on v using PRIMARY (VideoID=d.VideoID)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=257981
```

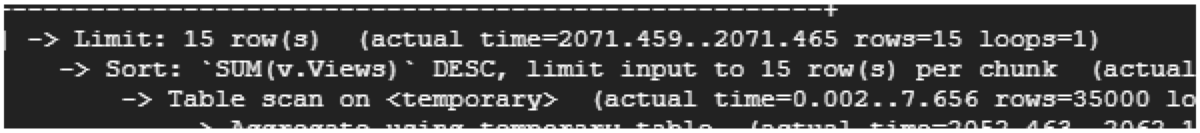
Below we show 3 attempts on creating an index in attempts to improve on performances.

1. Here we created an index for Videos.Views in attempts to lower the cost of Sum(v.Views) to see if indexing would have an effect on the end results of Sum.



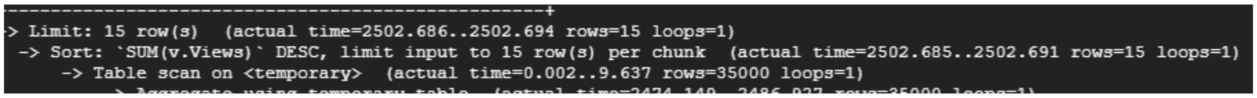
```
mysql> CREATE INDEX like_idx ON Videos(Views);
Query OK, 0 rows affected (1.95 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Before :



```
---------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=2071.459..2071.465 rows=15 loops=1)
    -> Sort: `SUM(v.Views)` DESC, limit input to 15 row(s) per chunk  (actual
        -> Table scan on <temporary>  (actual time=0.002..7.656 rows=35000 lo
            -> Aggregate using temporary table  (actual time=2052.463..2062.1
```

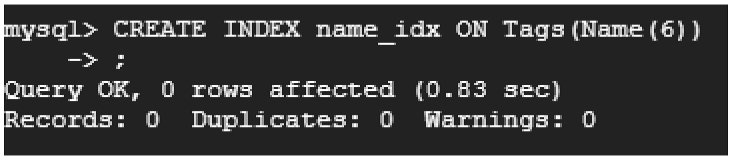
After:



```
------------------------------------------------+
-> Limit: 15 row(s)  (actual time=2502.686..2502.694 rows=15 loops=1)
 -> Sort: `SUM(v.Views)` DESC, limit input to 15 row(s) per chunk  (actual time=2502.685..2502.691 rows=15 loops=1)
    -> Table scan on <temporary>  (actual time=0.002..9.637 rows=35000 loops=1)
        -> Aggregate using temporary table  (actual time=2474.149..2486.927 rows=35000 loops=1)
```

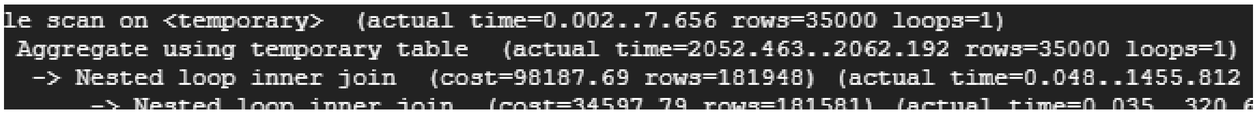
The indexing made no difference on the cost and rows but somehow made the query took more time. The time increase we suspect the reason being due to latency issues. And the results show that the performance of the things in SELECT clause will not be affected by indexing.

2. Here we created an index for Tags.Name to see if group by clause would be affected by indexing.



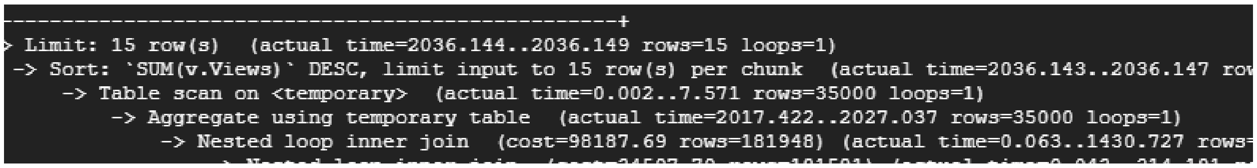
```
mysql> CREATE INDEX name_idx ON Tags(Name(6))
    -> ;
Query OK, 0 rows affected (0.83 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Before:



```
le scan on <temporary>  (actual time=0.002..7.656 rows=35000 loops=1)
 Aggregate using temporary table  (actual time=2052.463..2062.192 rows=35000 loops=1)
  -> Nested loop inner join  (cost=98187.69 rows=181948) (actual time=0.048..1455.812
      -> Nested loop inner join  (cost=34597.79 rows=181581) (actual time=0.035..320.6
```

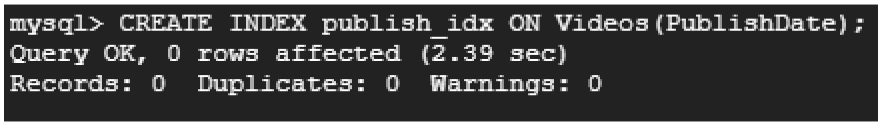
After:



```
------------------------------------------------+
> Limit: 15 row(s)  (actual time=2036.144..2036.149 rows=15 loops=1)
 -> Sort: `SUM(v.Views)` DESC, limit input to 15 row(s) per chunk  (actual time=2036.143..2036.147 row
    -> Table scan on <temporary>  (actual time=0.002..7.571 rows=35000 loops=1)
        -> Aggregate using temporary table  (actual time=2017.422..2027.037 rows=35000 loops=1)
            -> Nested loop inner join  (cost=98187.69 rows=181948) (actual time=0.063..1430.727 rows=
                -> Nested loop inner join  (cost=34597.79 rows=181581) (actual time=0.043..314.181 r
```

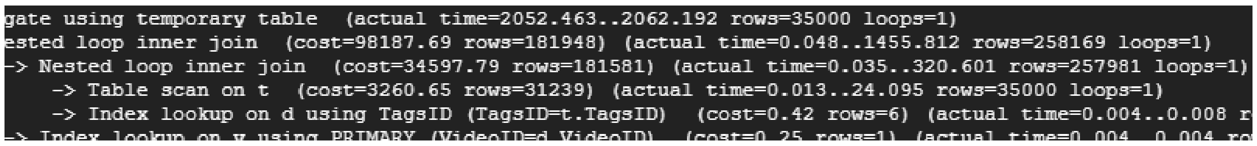
The results show no big difference on time and rows checked. Our assumption is that ***Group by*** clause does not use any indexing to speed up performance.

3. Created an index for Videos.PublishDate in attempts to see if any indexing of the contents within the joined table would affect the cost of nest joins



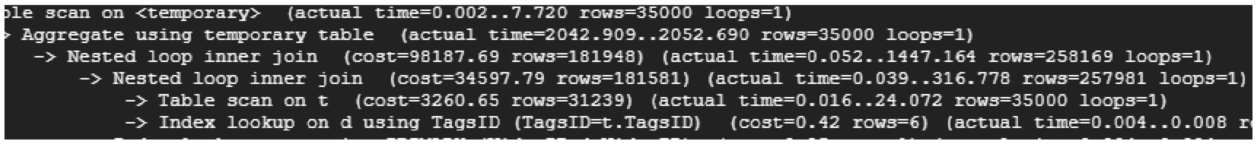
```
mysql> CREATE INDEX publish_idx ON Videos(PublishDate);
Query OK, 0 rows affected (2.39 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Before:



```
gate using temporary table  (actual time=2052.463..2062.192 rows=35000 loops=1)
ested loop inner join  (cost=98187.69 rows=181948) (actual time=0.048..1455.812 rows=258169 loops=1)
-> Nested loop inner join  (cost=34597.79 rows=181581) (actual time=0.035..320.601 rows=257981 loops=1)
    -> Table scan on t  (cost=3260.65 rows=31239) (actual time=0.013..24.095 rows=35000 loops=1)
    -> Index lookup on d using TagsID (TagsID=t.TagsID)  (cost=0.42 rows=6) (actual time=0.004..0.008 r
-> Index lookup on v using PRIMARY (VideoID=d.VideoID)  (cost=0.25 rows=1) (actual time=0.004..0.004 ro
```

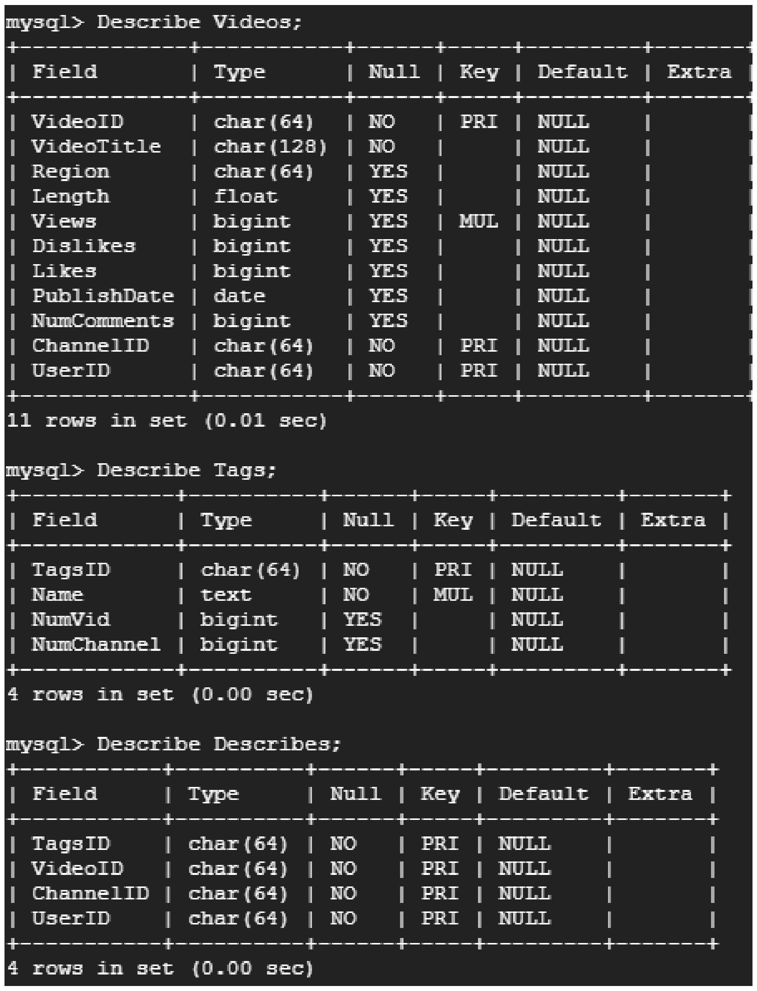
After:



```
ble scan on <temporary>  (actual time=0.002..7.720 rows=35000 loops=1)
> Aggregate using temporary table  (actual time=2042.909..2052.690 rows=35000 loops=1)
   -> Nested loop inner join  (cost=98187.69 rows=181948) (actual time=0.052..1447.164 rows=258169 loops=1)
       -> Nested loop inner join  (cost=34597.79 rows=181581) (actual time=0.039..316.778 rows=257981 loops=1)
           -> Table scan on t  (cost=3260.65 rows=31239) (actual time=0.016..24.072 rows=35000 loops=1)
           -> Index lookup on d using TagsID (TagsID=t.TagsID)  (cost=0.42 rows=6) (actual time=0.004..0.008 r
```

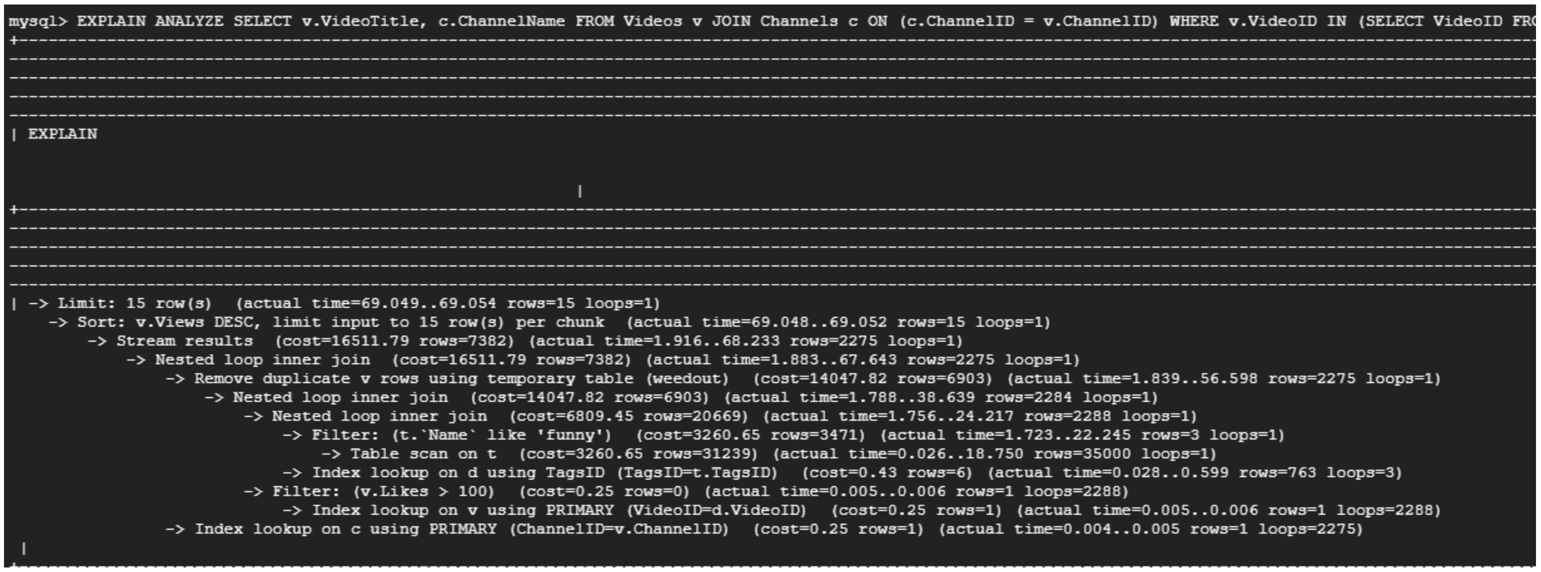
The results show no difference with the cost or rows checked. We assume that the index of joined tables' contents does not affect the cost of nested loop inner joins.

Conclusion: Since this query mainly uses Primary keys to do any searches and joins, we don't have much things that we can optimize for.



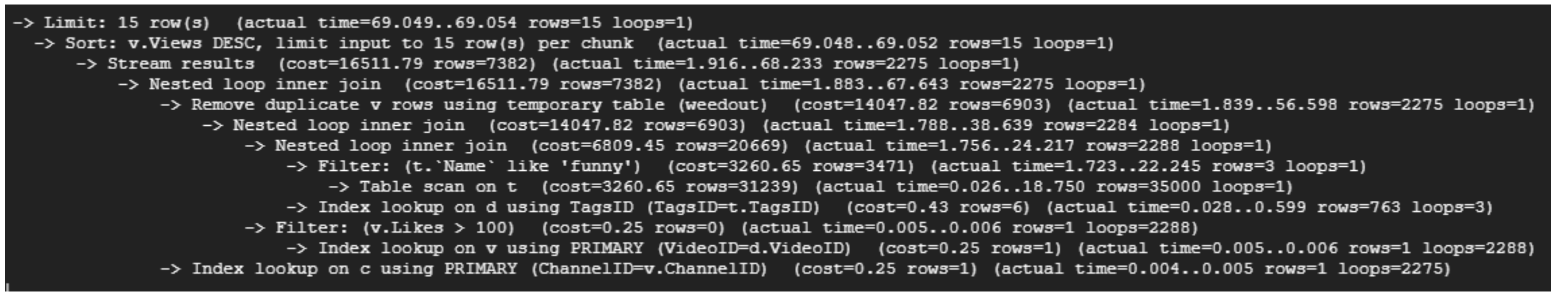
```
mysql> Describe Videos;
+-------------+-----------+------+-----+---------+-------
| Field       | Type      | Null | Key | Default | Extra |
+-------------+-----------+------+-----+---------+-------
| VideoID     | char(64)  | NO   | PRI | NULL    |
| VideoTitle  | char(128) | NO   |     | NULL    |
| Region      | char(64)  | YES  |     | NULL    |
| Length      | float     | YES  |     | NULL    |
| Views       | bigint    | YES  | MUL | NULL    |
| Dislikes    | bigint    | YES  |     | NULL    |
| Likes       | bigint    | YES  |     | NULL    |
| PublishDate | date      | YES  |     | NULL    |
| NumComments | bigint    | YES  |     | NULL    |
| ChannelID   | char(64)  | NO   | PRI | NULL    |
| UserID      | char(64)  | NO   | PRI | NULL    |
+-------------+-----------+------+-----+---------+-------
11 rows in set (0.01 sec)

mysql> Describe Tags;
+------------+----------+------+-----+---------+-------+
| Field      | Type     | Null | Key | Default | Extra |
+------------+----------+------+-----+---------+-------+
| TagsID     | char(64) | NO   | PRI | NULL    |       |
| Name       | text     | NO   | MUL | NULL    |       |
| NumVid     | bigint   | YES  |     | NULL    |       |
| NumChannel | bigint   | YES  |     | NULL    |       |
+------------+----------+------+-----+---------+-------+
4 rows in set (0.00 sec)

mysql> Describe Describes;
+-----------+----------+------+-----+---------+-------+
| Field     | Type     | Null | Key | Default | Extra |
+-----------+----------+------+-----+---------+-------+
| TagsID    | char(64) | NO   | PRI | NULL    |       |
| VideoID   | char(64) | NO   | PRI | NULL    |       |
| ChannelID | char(64) | NO   | PRI | NULL    |       |
| UserID    | char(64) | NO   | PRI | NULL    |       |
+-----------+----------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```

We ran explain analyze on the second query with the below result.



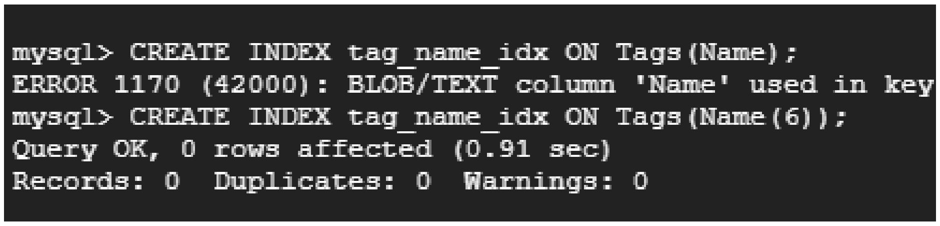
```
mysql> EXPLAIN ANALYZE SELECT v.VideoTitle, c.ChannelName FROM Videos v JOIN Channels c ON (c.ChannelID = v.ChannelID) WHERE v.VideoID IN (SELECT VideoID FRO
+-----------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------
| EXPLAIN


                                                         |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------
| -> Limit: 15 row(s)  (actual time=69.049..69.054 rows=15 loops=1)
    -> Sort: v.Views DESC, limit input to 15 row(s) per chunk  (actual time=69.048..69.052 rows=15 loops=1)
        -> Stream results  (cost=16511.79 rows=7382) (actual time=1.916..68.233 rows=2275 loops=1)
            -> Nested loop inner join  (cost=16511.79 rows=7382) (actual time=1.883..67.643 rows=2275 loops=1)
                -> Remove duplicate v rows using temporary table (weedout)  (cost=14047.82 rows=6903) (actual time=1.839..56.598 rows=2275 loops=1)
                    -> Nested loop inner join  (cost=14047.82 rows=6903) (actual time=1.788..38.639 rows=2284 loops=1)
                        -> Nested loop inner join  (cost=6809.45 rows=20669) (actual time=1.756..24.217 rows=2288 loops=1)
                            -> Filter: (t.`Name` like 'funny')  (cost=3260.65 rows=3471) (actual time=1.723..22.245 rows=3 loops=1)
                                -> Table scan on t  (cost=3260.65 rows=31239) (actual time=0.026..18.750 rows=35000 loops=1)
                            -> Index lookup on d using TagsID (TagsID=t.TagsID)  (cost=0.43 rows=6) (actual time=0.028..0.599 rows=763 loops=3)
                        -> Filter: (v.Likes > 100)  (cost=0.25 rows=0) (actual time=0.005..0.006 rows=1 loops=2288)
                            -> Index lookup on v using PRIMARY (VideoID=d.VideoID)  (cost=0.25 rows=1) (actual time=0.005..0.006 rows=1 loops=2288)
                -> Index lookup on c using PRIMARY (ChannelID=v.ChannelID)  (cost=0.25 rows=1) (actual time=0.004..0.005 rows=1 loops=2275)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------
```

Close up shot:



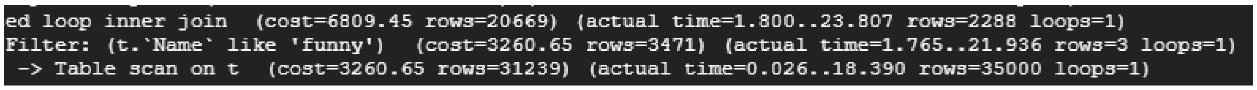
```
-> Limit: 15 row(s)  (actual time=69.049..69.054 rows=15 loops=1)
  -> Sort: v.Views DESC, limit input to 15 row(s) per chunk  (actual time=69.048..69.052 rows=15 loops=1)
      -> Stream results  (cost=16511.79 rows=7382) (actual time=1.916..68.233 rows=2275 loops=1)
          -> Nested loop inner join  (cost=16511.79 rows=7382) (actual time=1.883..67.643 rows=2275 loops=1)
              -> Remove duplicate v rows using temporary table (weedout)  (cost=14047.82 rows=6903) (actual time=1.839..56.598 rows=2275 loops=1)
                  -> Nested loop inner join  (cost=14047.82 rows=6903) (actual time=1.788..38.639 rows=2284 loops=1)
                      -> Nested loop inner join  (cost=6809.45 rows=20669) (actual time=1.756..24.217 rows=2288 loops=1)
                          -> Filter: (t.`Name` like 'funny')  (cost=3260.65 rows=3471) (actual time=1.723..22.245 rows=3 loops=1)
                              -> Table scan on t  (cost=3260.65 rows=31239) (actual time=0.026..18.750 rows=35000 loops=1)
                          -> Index lookup on d using TagsID (TagsID=t.TagsID)  (cost=0.43 rows=6) (actual time=0.028..0.599 rows=763 loops=3)
                      -> Filter: (v.Likes > 100)  (cost=0.25 rows=0) (actual time=0.005..0.006 rows=1 loops=2288)
                          -> Index lookup on v using PRIMARY (VideoID=d.VideoID)  (cost=0.25 rows=1) (actual time=0.005..0.006 rows=1 loops=2288)
              -> Index lookup on c using PRIMARY (ChannelID=v.ChannelID)  (cost=0.25 rows=1) (actual time=0.004..0.005 rows=1 loops=2275)
```

1. Created an index for Tags.Name to make the filter easier with the name hashed with 6 char.



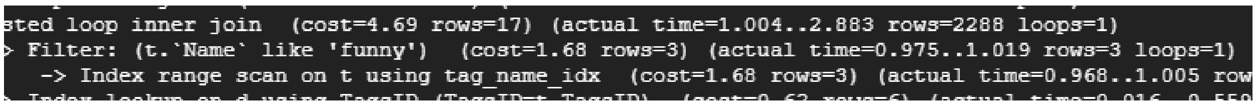
```
mysql> CREATE INDEX tag_name_idx ON Tags(Name);
ERROR 1170 (42000): BLOB/TEXT column 'Name' used in key
mysql> CREATE INDEX tag_name_idx ON Tags(Name(6));
Query OK, 0 rows affected (0.91 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Before:



```
ed loop inner join  (cost=6809.45 rows=20669) (actual time=1.800..23.807 rows=2288 loops=1)
Filter: (t.`Name` like 'funny')  (cost=3260.65 rows=3471) (actual time=1.765..21.936 rows=3 loops=1)
 -> Table scan on t  (cost=3260.65 rows=31239) (actual time=0.026..18.390 rows=35000 loops=1)
```

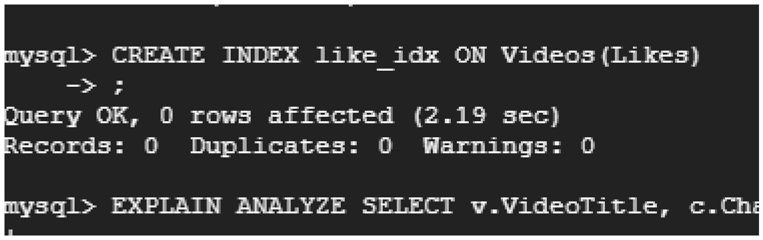
After:



```
sted loop inner join  (cost=4.69 rows=17) (actual time=1.004..2.883 rows=2288 loops=1)
> Filter: (t.`Name` like 'funny')  (cost=1.68 rows=3) (actual time=0.975..1.019 rows=3 loops=1)
   -> Index range scan on t using tag_name_idx  (cost=1.68 rows=3) (actual time=0.968..1.005 row
```

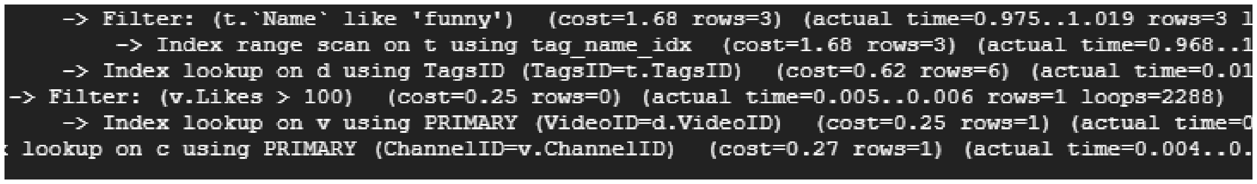
After the indexing, our cost lowered from 3260.65 and checked 3471 rows to only 1.68 on cost and 3 rows checked. Execution time decreased from 0.06 seconds to 0.04.

2. Created an index for Videos.Likes in attempts to lower the filtering range



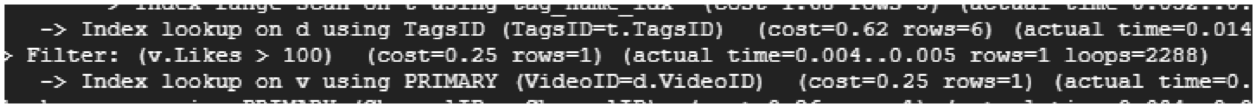
```
mysql> CREATE INDEX like_idx ON Videos(Likes)
    -> ;
Query OK, 0 rows affected (2.19 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN ANALYZE SELECT v.VideoTitle, c.Cha
```

Before:



```
    -> Filter: (t.`Name` like 'funny')  (cost=1.68 rows=3) (actual time=0.975..1.019 rows=3 l
        -> Index range scan on t using tag_name_idx  (cost=1.68 rows=3) (actual time=0.968..1
    -> Index lookup on d using TagsID (TagsID=t.TagsID)  (cost=0.62 rows=6) (actual time=0.01
-> Filter: (v.Likes > 100)  (cost=0.25 rows=0) (actual time=0.005..0.006 rows=1 loops=2288)
    -> Index lookup on v using PRIMARY (VideoID=d.VideoID)  (cost=0.25 rows=1) (actual time=0
 lookup on c using PRIMARY (ChannelID=v.ChannelID)  (cost=0.27 rows=1) (actual time=0.004..0.
```

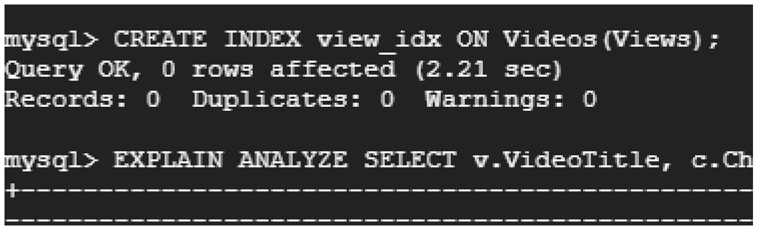
After:



```
  -> Index lookup on d using TagsID (TagsID=t.TagsID)  (cost=0.62 rows=6) (actual time=0.014
 Filter: (v.Likes > 100)  (cost=0.25 rows=1) (actual time=0.004..0.005 rows=1 loops=2288)
  -> Index lookup on v using PRIMARY (VideoID=d.VideoID)  (cost=0.25 rows=1) (actual time=0.
```

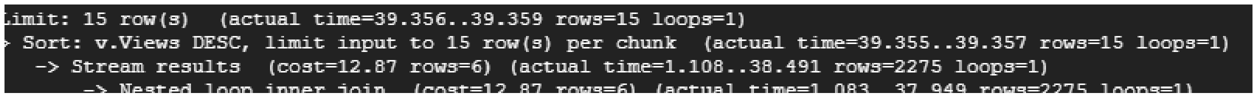
After the indexing, the cost remains and somehow added one row to check.Our prediction is that maybe >100 is too small of a boundary to make a difference using that index or the index was never used in the first place.

3. Created an index for Videos.Views to see if the sorting at the end would make a difference with the index



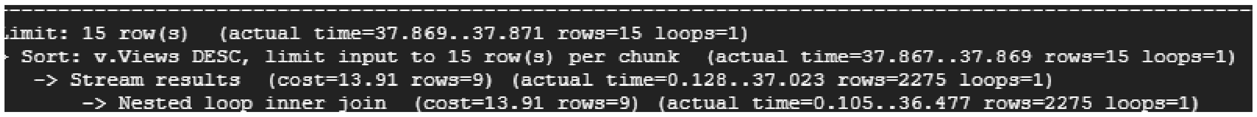
```
mysql> CREATE INDEX view_idx ON Videos(Views);
Query OK, 0 rows affected (2.21 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN ANALYZE SELECT v.VideoTitle, c.Ch
+-----------------------------------------------
------------------------------------------------
```

Before:



```
Limit: 15 row(s)  (actual time=39.356..39.359 rows=15 loops=1)
 Sort: v.Views DESC, limit input to 15 row(s) per chunk  (actual time=39.355..39.357 rows=15 loops=1)
  -> Stream results  (cost=12.87 rows=6) (actual time=1.108..38.491 rows=2275 loops=1)
      -> Nested loop inner join  (cost=12.87 rows=6) (actual time=1.083..37.949 rows=2275 loops=1)
```

After:



```
----------------------------------------------------------------------------------------------------
Limit: 15 row(s)  (actual time=37.869..37.871 rows=15 loops=1)
 Sort: v.Views DESC, limit input to 15 row(s) per chunk  (actual time=37.867..37.869 rows=15 loops=1)
  -> Stream results  (cost=13.91 rows=9) (actual time=0.128..37.023 rows=2275 loops=1)
      -> Nested loop inner join  (cost=13.91 rows=9) (actual time=0.105..36.477 rows=2275 loops=1)
```

After indexing the cost and rows checked remains the same. Which means the index might not help the sorting algorithm or the fact that limit 15 had something to do with the sorting and the indexing is not used under that circumstances.

Conclusion:

By indexing Tags.Name we successfully lowered the filtering part from a cost of 3260.65 to 1.68 and only checked 3 rows instead of checking 3471 rows. Making the final time used of this query decreased from 0.06 seconds to 0.04 seconds.