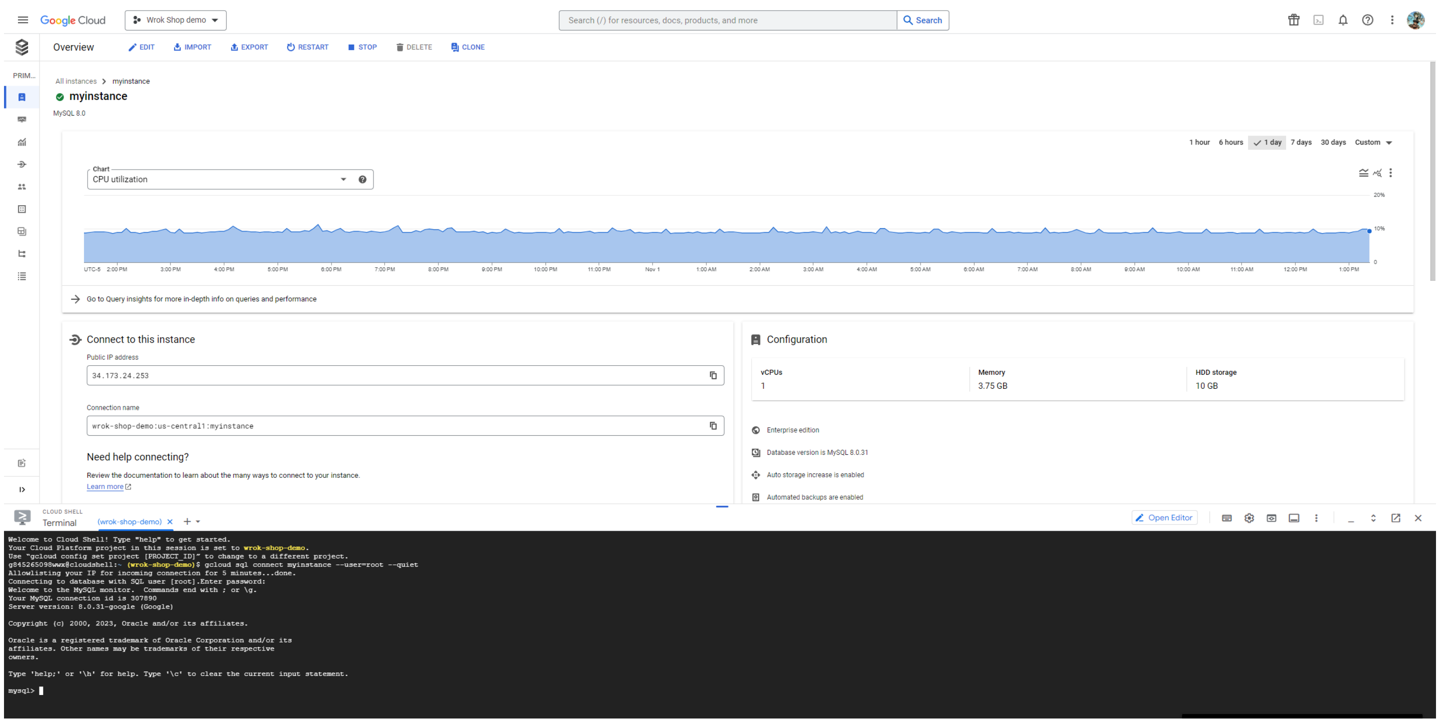
GCP Connection



Public IP address: 34.173.24.253

Connection name: wrok-shop-demo:us-central1:myinstance

vCPUs	Memory	HDD storage
1	3.75 GB	10 GB

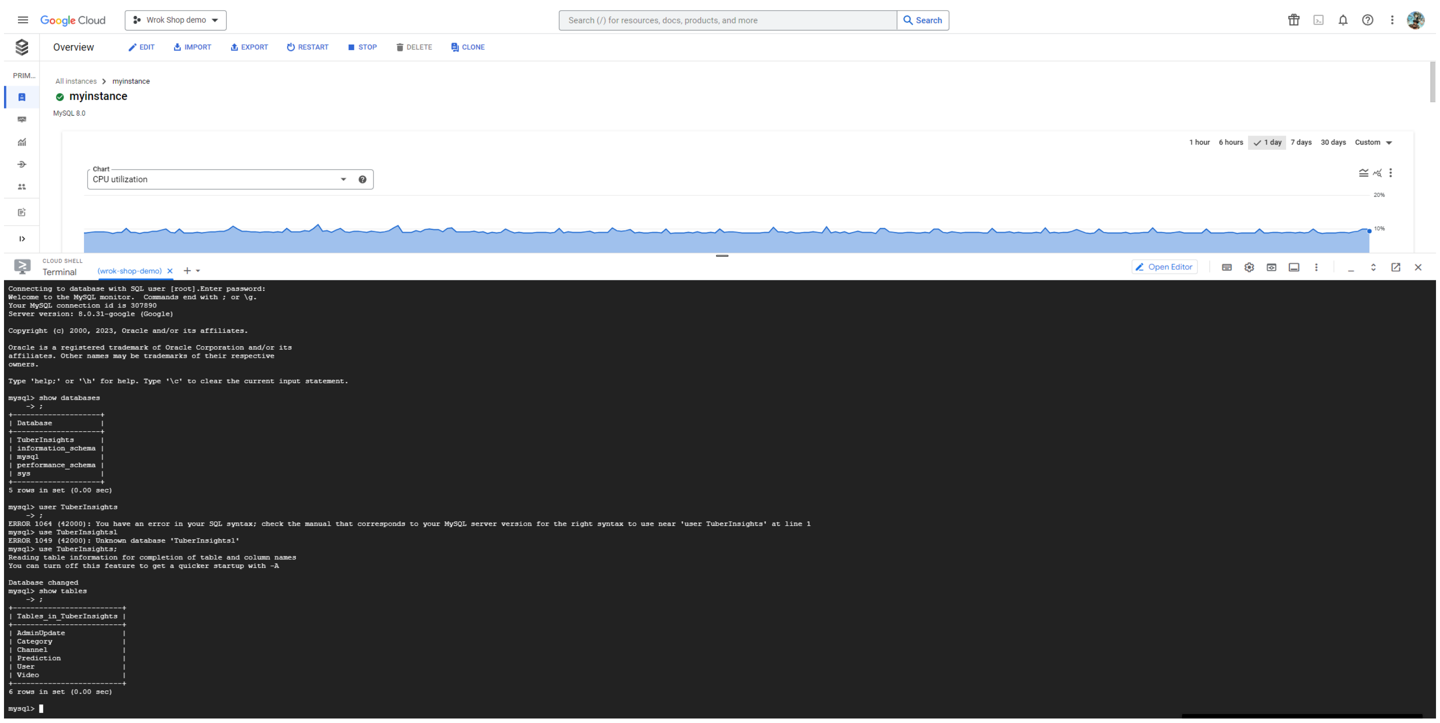
Enterprise edition

Database version is MySQL 8.0.31

Auto storage increase is enabled

Automated backups are enabled

```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to wrok-shop-demo.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
g845265098wwx@cloudshell:~ (wrok-shop-demo)$ gcloud sql connect myinstance --user=root --quiet
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 307890
Server version: 8.0.31-google (Google)

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

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

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

mysql> 
```



```
Connecting to database with SQL user [root].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 307890
Server version: 8.0.31-google (Google)

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

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

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

mysql> show databases
    -> ;
+--------------------+
| Database           |
+--------------------+
| TuberInsights      |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)

mysql> user TuberInsights
    -> ;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'user TuberInsights' at line 1
mysql> use TuberInsights1
ERROR 1049 (42000): Unknown database 'TuberInsights1'
mysql> use TuberInsights;
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_TuberInsights |
+-------------------------+
| AdminUpdate             |
| Category                |
| Channel                 |
| Prediction              |
| User                    |
| Video                   |
+-------------------------+
6 rows in set (0.00 sec)

mysql> 
```

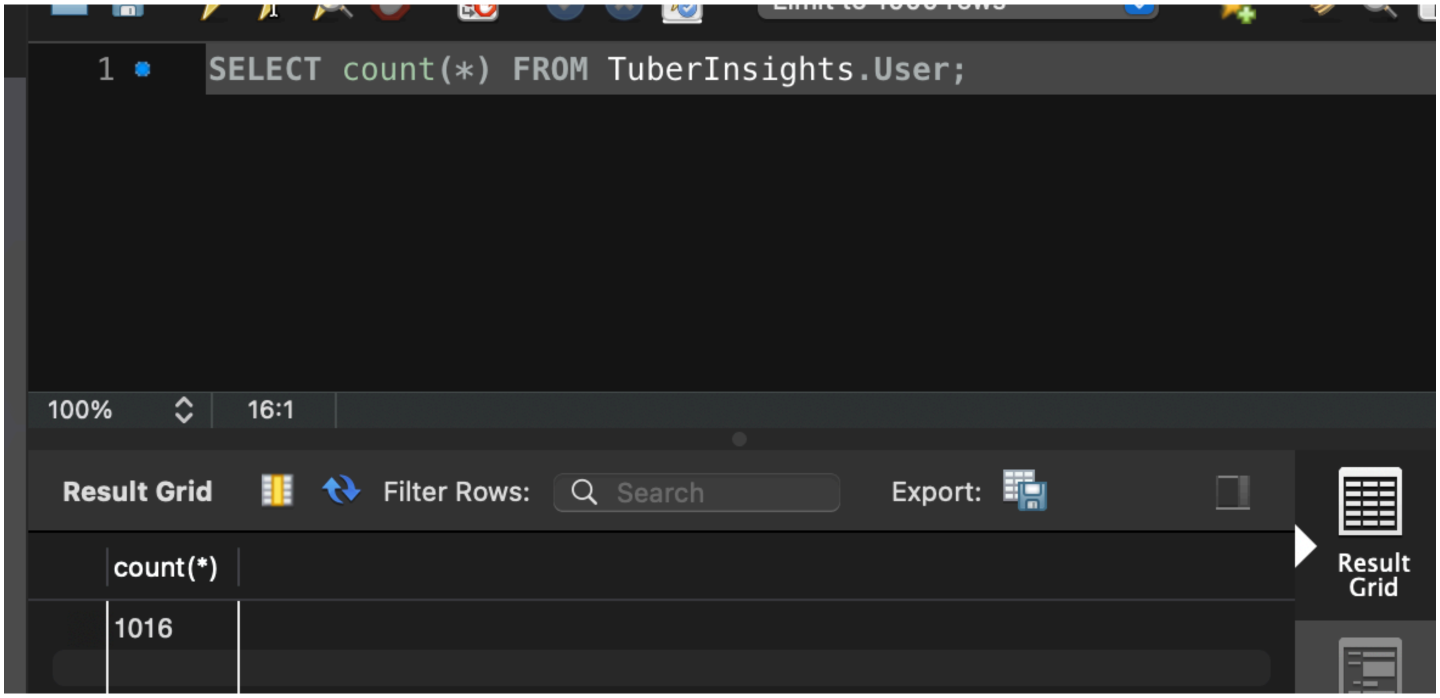
DDL command

```
CREATE DATABASE IF NOT EXISTS TuberInsights;
```

```sql
USE TuberInsights;

CREATE TABLE User (
    userID INT PRIMARY KEY,
    userName VARCHAR(30),
    password VARCHAR(40),
    userType ENUM('admin', 'default')
);

CREATE TABLE Prediction (
    predictID INT PRIMARY KEY,
    userID INT,
    model ENUM('tag_generation', 'trending_prediction'),
    input VARCHAR(100),
    result VARCHAR(100),
    FOREIGN KEY (userID) REFERENCES User(userID) ON DELETE CASCADE ON UPDATE SET NULL

);

CREATE TABLE Channel (
    channelID VARCHAR(35) PRIMARY KEY,
    channelTitle VARCHAR(30)
);

CREATE TABLE Category (
    categoryID VARCHAR(35) PRIMARY KEY,
    categoryName VARCHAR(30)
);

CREATE TABLE Video (
    videoID VARCHAR(35) PRIMARY KEY,
    channelID VARCHAR(35),
    categoryID VARCHAR(35),
    dislikes INT,
    likes INT,
    description TEXT,
    tags VARCHAR(255),
    trending_date DATE,
    title VARCHAR(255),
    comment_count INT,
    view_count BIGINT,
    published_at DATE,
    region VARCHAR(255),
    FOREIGN KEY (channelID) REFERENCES Channel(channelID) ON DELETE CASCADE ON UPDATE SET
NULL,
    FOREIGN KEY (categoryID) REFERENCES Category(categoryID) ON DELETE CASCADE ON UPDATE SET
NULL
);
```

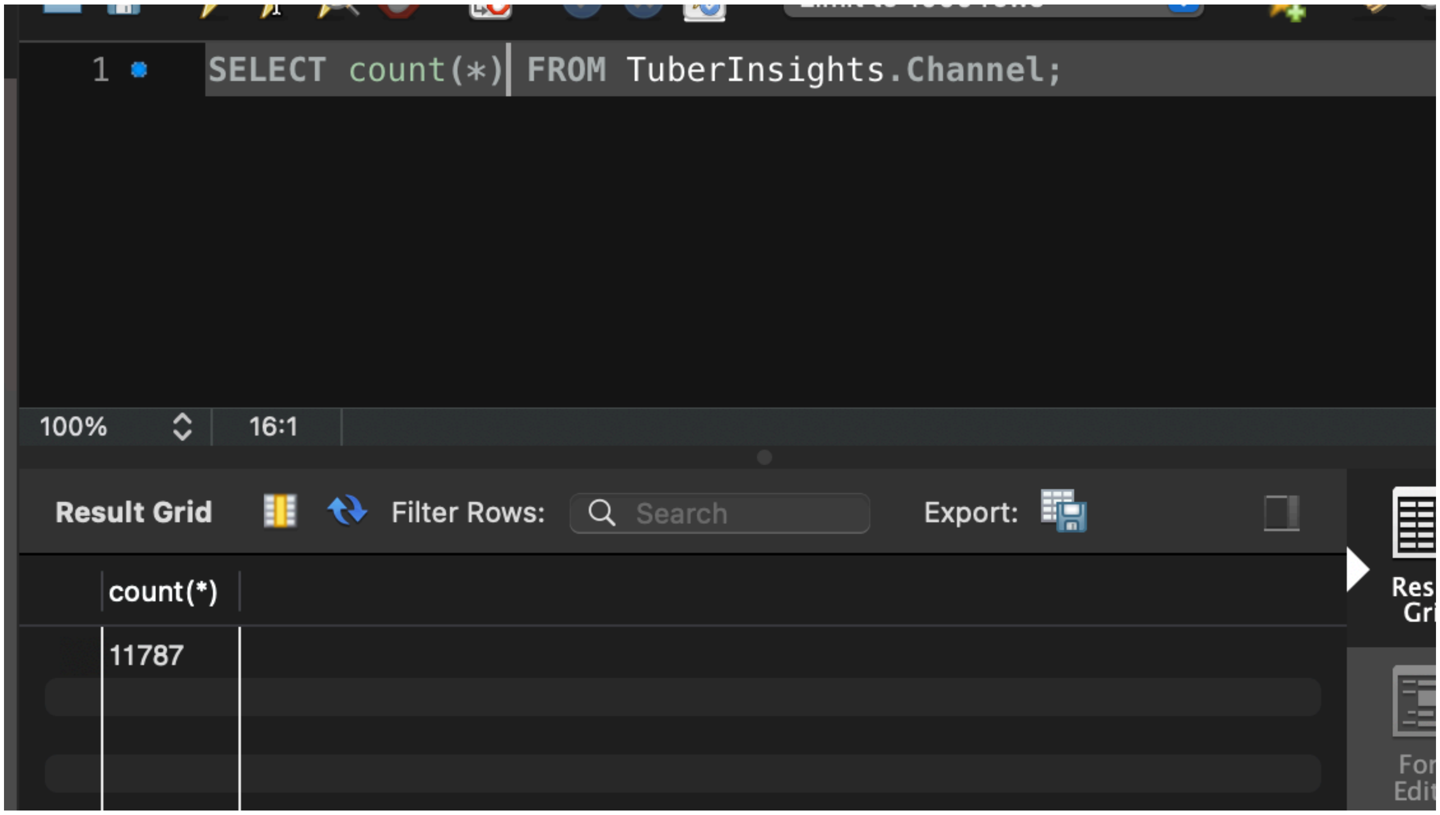
```
CREATE TABLE AdminUpdate (
    videoID VARCHAR(35),
    userID INT,
    PRIMARY KEY (videoID, userID),
    FOREIGN KEY (videoID) REFERENCES Video(videoID) ON DELETE CASCADE,
    FOREIGN KEY (userID) REFERENCES User(userID) ON DELETE CASCADE
);
```

Table Records



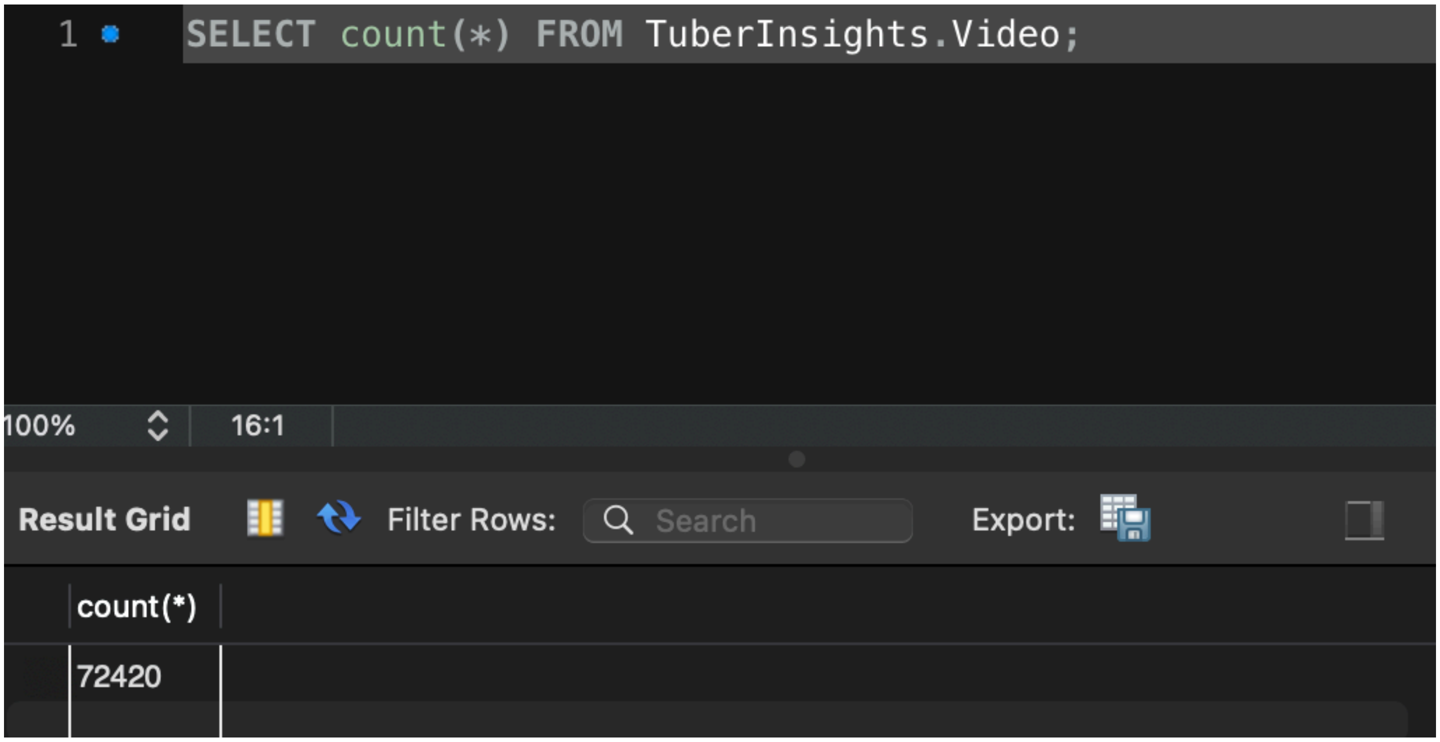
```
SELECT count(*) FROM TuberInsights.User;
```

```
count(*)
1016
```



```
SELECT count(*) FROM TuberInsights.Channel;
```

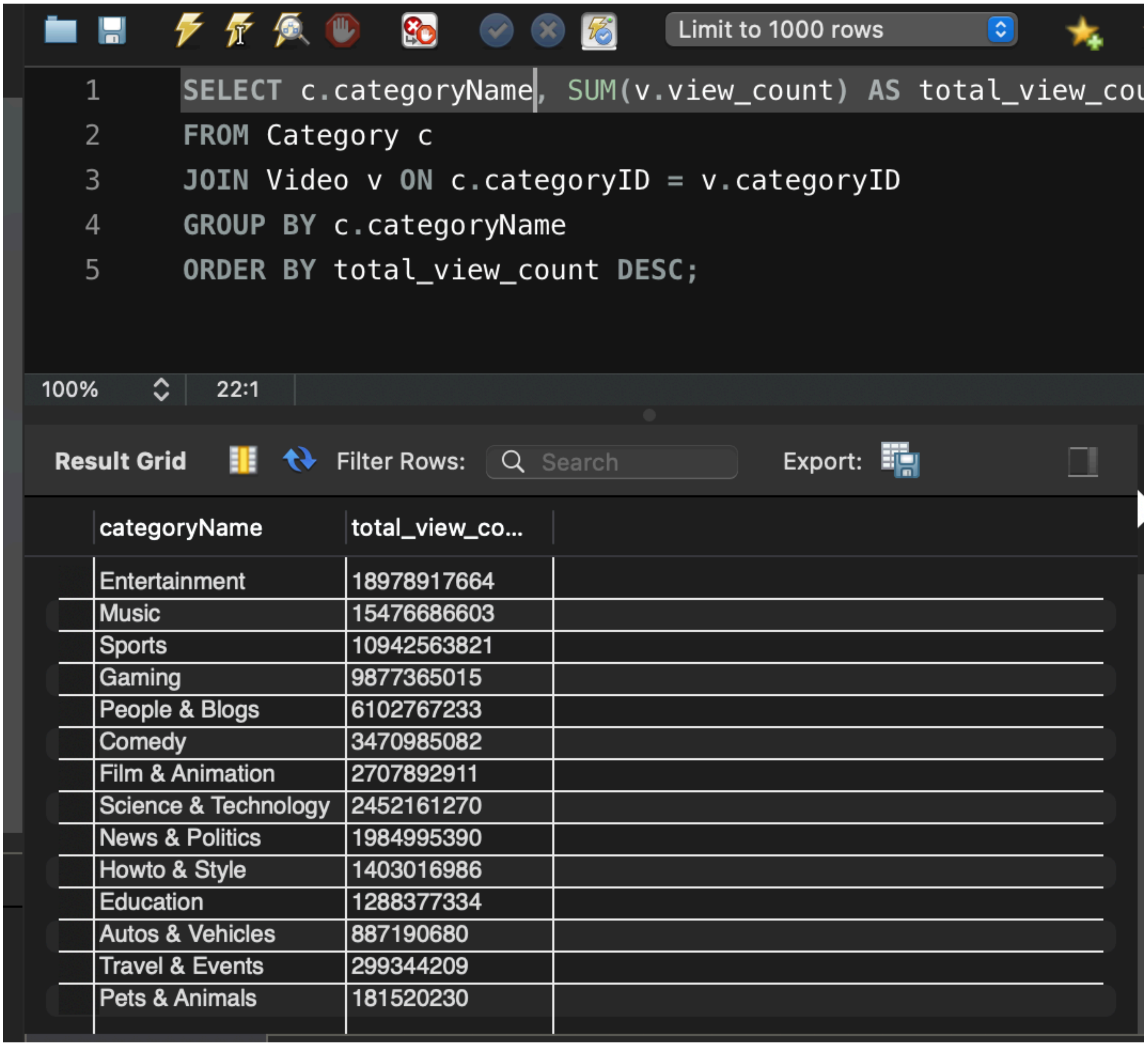
```
count(*)
11787
```



```
SELECT count(*) FROM TuberInsights.Video;
```

```
count(*)
72420
```

Advanced Query



Limit to 1000 rows

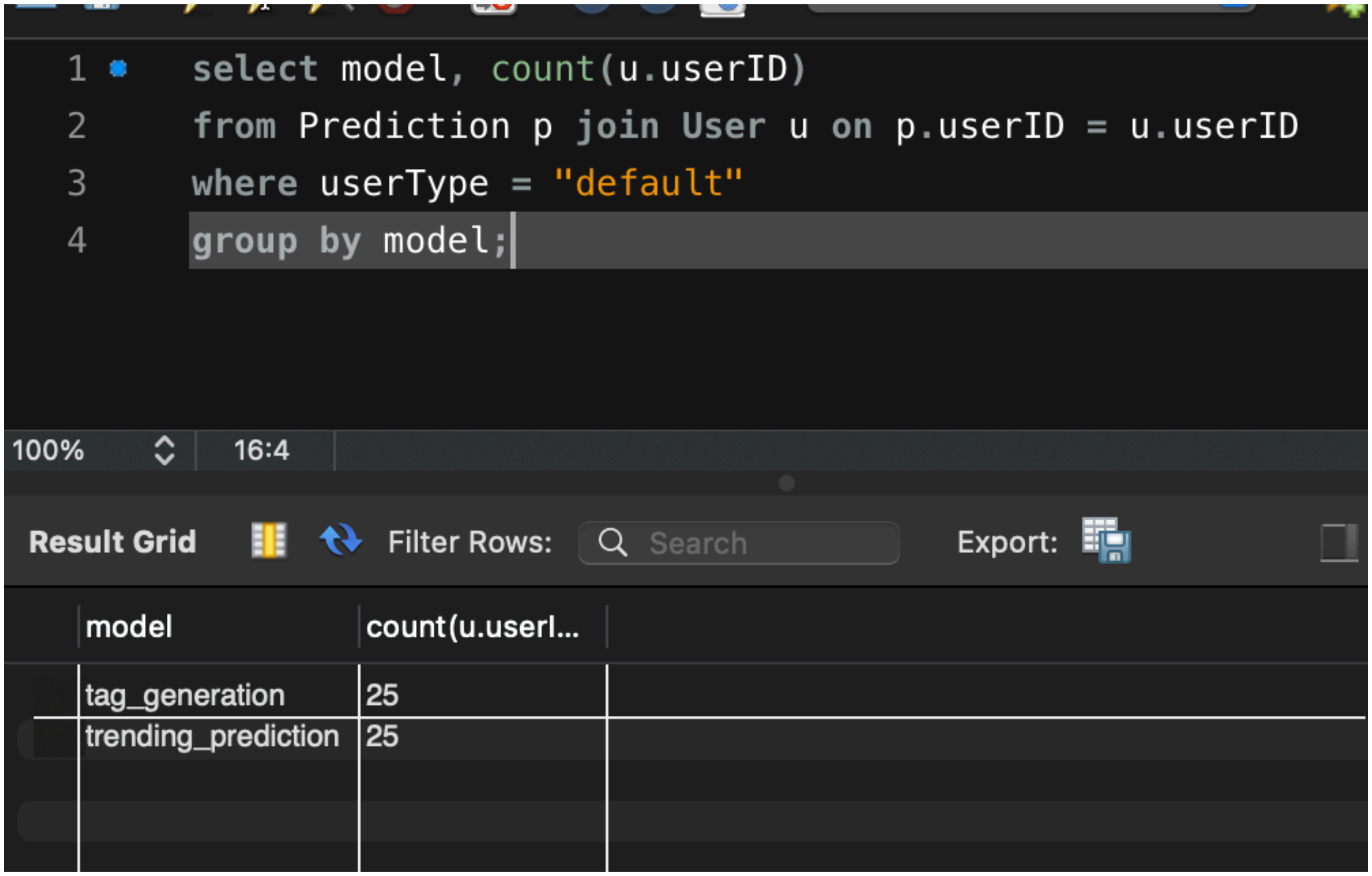
```
SELECT c.categoryName, SUM(v.view_count) AS total_view_cou
FROM Category c
JOIN Video v ON c.categoryID = v.categoryID
GROUP BY c.categoryName
ORDER BY total_view_count DESC;
```

100% 22:1

Result Grid Filter Rows: Search Export:

categoryName	total_view_co...
Entertainment	18978917664
Music	15476686603
Sports	10942563821
Gaming	9877365015
People & Blogs	6102767233
Comedy	3470985082
Film & Animation	2707892911
Science & Technology	2452161270
News & Politics	1984995390
Howto & Style	1403016986
Education	1288377334
Autos & Vehicles	887190680
Travel & Events	299344209
Pets & Animals	181520230

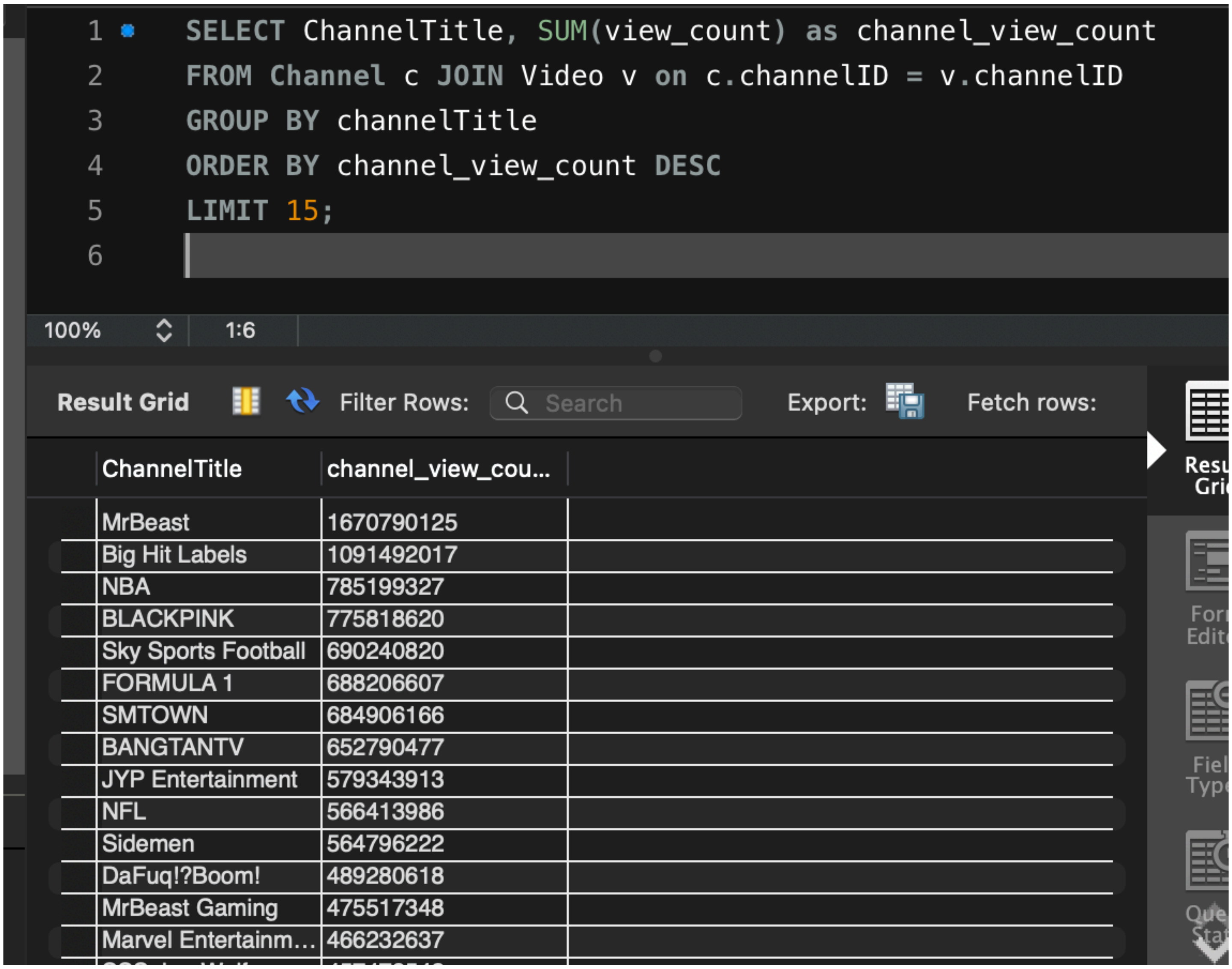
Only **14** rows returned from the first advanced query



```
select model, count(u.userID)
from Prediction p join User u on p.userID = u.userID
where userType = "default"
group by model;
```

model	count(u.userI...
tag_generation	25
trending_prediction	25

Only **2** rows generated from this advanced query



```
SELECT ChannelTitle, SUM(view_count) as channel_view_count
FROM Channel c JOIN Video v on c.channelID = v.channelID
GROUP BY channelTitle
ORDER BY channel_view_count DESC
LIMIT 15;

```

ChannelTitle	channel_view_cou...
MrBeast	1670790125
Big Hit Labels	1091492017
NBA	785199327
BLACKPINK	775818620
Sky Sports Football	690240820
FORMULA 1	688206607
SMTOWN	684906166
BANGTANTV	652790477
JYP Entertainment	579343913
NFL	566413986
Sidemen	564796222
DaFuq!?Boom!	489280618
MrBeast Gaming	475517348
Marvel Entertainm...	466232637

Indexing & Explaination

Query 1

```
EXPLAIN ANALYZE
SELECT ChannelTitle, SUM(view_count) as channel_view_count
FROM Channel c JOIN Video v on c.channelID = v.channelID
GROUP BY channelTitle
ORDER BY channel_view_count DESC
```

1. **Without indexing**

```
# EXPLAIN
-> Sort: channel_view_count DESC  (actual time=363.306..364.992 rows=11668 loops=1)
    -> Table scan on <temporary>  (actual time=351.150..354.019 rows=11668 loops=1)
        -> Aggregate using temporary table  (actual time=351.148..351.148 rows=11668
loops=1)
            -> Nested loop inner join  (cost=24268.42 rows=65780) (actual
time=0.099..278.095 rows=72420 loops=1)
                -> Table scan on c  (cost=1237.85 rows=12136) (actual time=0.052..4.855
rows=11787 loops=1)
                -> Index lookup on v using Video_ibfk_1 (channelID=c.channelID)  (cost=1.36
rows=5) (actual time=0.012..0.022 rows=6 loops=11787)
```

2. **B-tree** Indexing

```
-- indexing on channelID --
CREATE INDEX channelID_index ON Channel (channelID);

# EXPLAIN
-> Sort: channel_view_count DESC  (actual time=422.572..424.176 rows=11668 loops=1)
    -> Table scan on <temporary>  (actual time=410.976..413.844 rows=11668 loops=1)
        -> Aggregate using temporary table  (actual time=410.974..410.974 rows=11668
loops=1)
            -> Nested loop inner join  (cost=26478.59 rows=72116) (actual
time=0.091..340.839 rows=72420 loops=1)
                -> Table scan on c  (cost=1237.85 rows=12136) (actual time=0.036..5.124
rows=11787 loops=1)
                -> Index lookup on v using channelID (channelID=c.channelID)  (cost=1.49
rows=6) (actual time=0.015..0.028 rows=6 loops=11787)

-- Indexing on videoID

# EXPLAIN
-> Sort: channel_view_count DESC  (actual time=360.526..362.105 rows=11668 loops=1)
    -> Table scan on <temporary>  (actual time=348.782..351.605 rows=11668 loops=1)
        -> Aggregate using temporary table  (actual time=348.779..348.779 rows=11668
loops=1)
            -> Nested loop inner join  (cost=24268.42 rows=65780) (actual
time=0.104..279.465 rows=72420 loops=1)
                -> Table scan on c  (cost=1237.85 rows=12136) (actual time=0.060..4.797
rows=11787 loops=1)
                -> Index lookup on v using Video_ibfk_1 (channelID=c.channelID)  (cost=1.36
rows=5) (actual time=0.012..0.022 rows=6 loops=11787)


-- Indexing on channelID and viedoID
```

```
CREATE INDEX channelID_index ON Channel (channelID);
CREATE INDEX viewID_index ON Video (videoID);

# EXPLAIN
-> Sort: channel_view_count DESC  (actual time=409.990..411.651 rows=11668 loops=1)
    -> Table scan on <temporary>  (actual time=398.809..401.662 rows=11668 loops=1)
        -> Aggregate using temporary table  (actual time=398.807..398.807 rows=11668
loops=1)
            -> Nested loop inner join  (cost=26478.59 rows=72116) (actual
time=0.094..325.162 rows=72420 loops=1)
                -> Table scan on c  (cost=1237.85 rows=12136) (actual time=0.051..4.884
rows=11787 loops=1)
                -> Index lookup on v using channelID (channelID=c.channelID)  (cost=1.49
rows=6) (actual time=0.014..0.026 rows=6 loops=11787)

-- indexing on channelTitle --
CREATE INDEX channelTitle_index ON Channel (channelTitle);
# EXPLAIN
# EXPLAIN
-> Sort: channel_view_count DESC  (actual time=172.495..174.221 rows=11668 loops=1)
    -> Stream results  (cost=27473.57 rows=61848) (actual time=0.104..163.248 rows=11668
loops=1)
        -> Group aggregate: sum(v.view_count)  (cost=27473.57 rows=61848) (actual
time=0.100..157.353 rows=11668 loops=1)
            -> Nested loop inner join  (cost=21288.75 rows=61848) (actual
time=0.082..126.184 rows=72420 loops=1)
                -> Covering index scan on c using channelTitle_index  (cost=1237.85
rows=12136) (actual time=0.053..4.096 rows=11787 loops=1)
                -> Covering index lookup on v using video_covered_index
(channelID=c.channelID)  (cost=1.14 rows=5) (actual time=0.006..0.010 rows=6 loops=11787)

-- indexing on view_count
CREATE INDEX video_view_count_index ON Video (view_count);
# EXPLAIN
-> Sort: channel_view_count DESC  (actual time=354.908..356.522 rows=11668 loops=1)
    -> Table scan on <temporary>  (actual time=343.963..346.774 rows=11668 loops=1)
        -> Aggregate using temporary table  (actual time=343.961..343.961 rows=11668
loops=1)
            -> Nested loop inner join  (cost=24268.42 rows=65780) (actual
time=0.126..275.601 rows=72420 loops=1)
                -> Table scan on c  (cost=1237.85 rows=12136) (actual time=0.080..4.930
rows=11787 loops=1)
                -> Index lookup on v using Video_ibfk_1 (channelID=c.channelID)  (cost=1.36
rows=5) (actual time=0.011..0.022 rows=6 loops=11787)

-- indexing on channelID and view_count
CREATE INDEX video_covered_index ON Video (channelID, view_count);
```

```
-> Sort: channel_view_count DESC  (actual time=311.037..312.676 rows=11668 loops=1)
    -> Table scan on <temporary>  (actual time=300.070..302.844 rows=11668 loops=1)
        -> Aggregate using temporary table  (actual time=300.068..300.068 rows=11668 
loops=1)
            -> Nested loop inner join  (cost=21334.11 rows=61848) (actual 
time=0.085..229.292 rows=72420 loops=1)
                -> Table scan on c  (cost=1237.85 rows=12136) (actual time=0.050..5.388 
rows=11787 loops=1)
                -> Covering index lookup on v using video_covered_index 
(channelID=c.channelID)  (cost=1.15 rows=5) (actual time=0.008..0.018 rows=6 loops=11787)
```

For this advanced query, we compare the effect of using Channel(channelID), Channel(channelTitle),Video(videoID), Video(view_count), and Channel(channelID), Video(videoID) together. The cost of each query is shown in table below.

Indexing	Nested loop inner join cost	Table scan cost	Index lookup	Index lookup cost
channelID	26478.59	1237.85	channelID	1.49
Channel(channelTitle)	30846.40	24268.42	channelTitle_index	1237.85
videoID	24268.42	1237.85	Video_ibfk_1	1.36
Video(view_count)	26478.59	1237.85	Video_ibfk_1	1.36
channelID & videoID	26478.59	1237.85	channelID	1.49
Video(channelID & view_count)	***21334.11***	***1237.85***	video_covered_index	**1.15**
Default	24268.42	1237.85	Video_ibfk_1	1.36

There are two noticeable indexing methods are worth discussing after conducting the experiement. Firstly, indexing on channelTitle attribute from Channel Table has significantly negative effect on the query performance. We could see compared to default query, the cost of table scan is 20 times larger. Based on my understanding, One reason may be that creating index on GROUP BY attributes may not as efficient as that are used for filtering or joining. The GROUP BY sort by channelTitle which is not a primary key may also slow down the query process. Secondly, indexing on channelID and view_count on Video table improves the query performance. Here, I used covered indexes which contain all of the columns required for a query, so the query can be satisfied entirely by the index without having to access the underlying table.

Query 2

```
select model, count(u.userID)
from Prediction p join User u on p.userID = u.userID
where userType = "default"
group by model;
```

1. **Without indexing**

```
-> Table scan on <temporary>  (actual time=0.234..0.235 rows=2 loops=1)
    -> Aggregate using temporary table  (actual time=0.234..0.234 rows=2 loops=1)
        -> Nested loop inner join  (cost=22.75 rows=25) (actual time=0.051..0.160 rows=50 
loops=1)
            -> Filter: (p.userID is not null)  (cost=5.25 rows=50) (actual time=0.036..0.052 
rows=50 loops=1)
                -> Table scan on p  (cost=5.25 rows=50) (actual time=0.035..0.044 rows=50 
loops=1)
            -> Filter: (u.userType = 'default')  (cost=0.25 rows=0.5) (actual 
time=0.002..0.002 rows=1 loops=50)
                -> Single-row index lookup on u using PRIMARY (userID=p.userID)  (cost=0.25 
rows=1) (actual time=0.001..0.001 rows=1 loops=50)
```

2. **B-tree** Indexing

```
-- index on Prediction(model, userID)
CREATE INDEX coveredPredictionIndex ON Prediction (model, userID);
-> Table scan on <temporary>  (actual time=0.202..0.202 rows=2 loops=1)
    -> Aggregate using temporary table  (actual time=0.200..0.200 rows=2 loops=1)
        -> Nested loop inner join  (cost=22.75 rows=25) (actual time=0.048..0.130 rows=50 
loops=1)
            -> Filter: (p.userID is not null)  (cost=5.25 rows=50) (actual time=0.031..0.047 
rows=50 loops=1)
                -> Covering index scan on p using coveredPredictionIndex  (cost=5.25 
rows=50) (actual time=0.030..0.040 rows=50 loops=1)
            -> Filter: (u.userType = 'default')  (cost=0.25 rows=0.5) (actual 
time=0.001..0.001 rows=1 loops=50)
                -> Single-row index lookup on u using PRIMARY (userID=p.userID)  (cost=0.25 
rows=1) (actual time=0.001..0.001 rows=1 loops=50)

-- index on User(userID, userType)
CREATE INDEX coveredUserIndex ON User (userID, userType);
-> Table scan on <temporary>  (actual time=0.196..0.197 rows=2 loops=1)
    -> Aggregate using temporary table  (actual time=0.194..0.194 rows=2 loops=1)
        -> Nested loop inner join  (cost=22.75 rows=25) (actual time=0.038..0.120 rows=50 
loops=1)
            -> Filter: (p.userID is not null)  (cost=5.25 rows=50) (actual time=0.024..0.040 
rows=50 loops=1)
                -> Covering index scan on p using coveredUserIndex  (cost=5.25 rows=50) 
(actual time=0.030..0.040 rows=50 loops=1)
            -> Filter: (u.userType = 'default')  (cost=0.25 rows=0.5) (actual 
time=0.001..0.001 rows=1 loops=50)
                -> Single-row index lookup on u using PRIMARY (userID=p.userID)  (cost=0.25 
rows=1) (actual time=0.001..0.001 rows=1 loops=50)

-- index on Prediction(predictID, model)
CREATE INDEX coverdModelIndex ON Prediction (predictID, model);
```

```
-> Table scan on <temporary>  (actual time=0.228..0.228 rows=2 loops=1)
    -> Aggregate using temporary table  (actual time=0.227..0.227 rows=2 loops=1)
        -> Nested loop inner join  (cost=22.75 rows=25) (actual time=0.054..0.136 rows=50
loops=1)
            -> Filter: (p.userID is not null)  (cost=5.25 rows=50) (actual time=0.038..0.054
rows=50 loops=1)
                -> Table scan on p  (cost=5.25 rows=50) (actual time=0.035..0.044 rows=50
loops=1)
            -> Filter: (u.userType = 'default')  (cost=0.25 rows=0.5) (actual
time=0.001..0.001 rows=1 loops=50)
                -> Single-row index lookup on u using PRIMARY (userID=p.userID)  (cost=0.25
rows=1) (actual time=0.001..0.001 rows=1 loops=50)
```

For this advanced query, we compare the effect of using Prediction(model, userID), User(userID, userType),Prediction(predictID, model). The cost of each query is shown in table below.

Indexing	Nested loop inner join cost	Filter: (p.userID is not null) cost	Filter: (u.userType = 'default') cost
model, userID	22.75	5.25	0.25
userID, userType	22.75	5.25	0.25
predictID, model	22.75	5.25	0.25
default	22.75	5.25	0.25

So there are acturally not difference between the default indexing and the rest three added indexing. However, the explain analyze output does imply that MySQL is using different indexing strategies to execute query when different indexing were added to the tables. After a considerable analysis on this result, we believe that too few data might be the main cause of no improvement after adding indexing. Besides, since Prediction is a weak entity depending on User, the perforamce of join and consequently affect the perforamce of indexing. As a result, adding indexing has no effect on query perforamce.

Query 3

```
EXPLAIN ANALYZE
SELECT c.categoryName, SUM(v.view_count) AS total_view_count
FROM Category c
JOIN Video v ON c.categoryID = v.categoryID
GROUP BY c.categoryName
ORDER BY total_view_count DESC;
```

1. **Without indexing**

```
# EXPLAIN
-> Sort: total_view_count DESC  (actual time=215.152..215.154 rows=14 loops=1)
    -> Table scan on <temporary>  (actual time=215.118..215.121 rows=14 loops=1)
        -> Aggregate using temporary table  (actual time=215.115..215.115 rows=14 loops=1)
            -> Nested loop inner join  (cost=30612.71 rows=64376) (actual
time=0.130..153.223 rows=72420 loops=1)
                -> Filter: (v.categoryID is not null)  (cost=8081.11 rows=64376) (actual
time=0.103..59.762 rows=72420 loops=1)
                    -> Table scan on v  (cost=8081.11 rows=64376) (actual time=0.102..52.262
rows=72420 loops=1)
                -> Single-row index lookup on c using PRIMARY (categoryID=v.categoryID)
 (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=72420)
```

2. **B-tree** Indexing

```
-- indexing on Category(categoryName) --
CREATE INDEX categoryName_index ON Category (categoryName);

-> Sort: total_view_count DESC  (actual time=206.141..206.143 rows=14 loops=1)
    -> Table scan on <temporary>  (actual time=206.115..206.118 rows=14 loops=1)
        -> Aggregate using temporary table  (actual time=206.113..206.113 rows=14 loops=1)
            -> Nested loop inner join  (cost=30612.71 rows=64376) (actual
time=0.081..145.102 rows=72420 loops=1)
                -> Filter: (v.categoryID is not null)  (cost=8081.11 rows=64376) (actual
time=0.063..52.126 rows=72420 loops=1)
                    -> Table scan on v  (cost=8081.11 rows=64376) (actual time=0.062..45.087
rows=72420 loops=1)
                -> Single-row index lookup on c using PRIMARY (categoryID=v.categoryID)
 (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=72420)

-- Covered index for the Video table --
CREATE INDEX video_index ON Video (categoryID, view_count);

# EXPLAIN
-> Sort: total_view_count DESC  (actual time=166.687..166.689 rows=14 loops=1)
    -> Table scan on <temporary>  (actual time=166.656..166.663 rows=14 loops=1)
        -> Aggregate using temporary table  (actual time=166.654..166.654 rows=14 loops=1)
            -> Nested loop inner join  (cost=20867.01 rows=153512) (actual
time=0.770..111.423 rows=72420 loops=1)
                -> Table scan on c  (cost=3.35 rows=31) (actual time=0.025..0.065 rows=31
loops=1)
                -> Covering index lookup on v using video_index (categoryID=c.categoryID)
 (cost=193.80 rows=4952) (actual time=0.042..3.343 rows=2336 loops=31)
```

```
-- Covered index for the Video table and Category table--
CREATE INDEX category_index ON Category (categoryID, categoryName);
CREATE INDEX video_index ON Video (categoryID, view_count);

# EXPLAIN
-> Sort: total_view_count DESC  (actual time=161.173..161.175 rows=14 loops=1)
    -> Table scan on <temporary>  (actual time=161.139..161.146 rows=14 loops=1)
        -> Aggregate using temporary table  (actual time=161.136..161.136 rows=14 loops=1)
            -> Nested loop inner join  (cost=20835.69 rows=153512) (actual
time=0.089..66.437 rows=72420 loops=1)
                -> Table scan on c  (cost=3.35 rows=31) (actual time=0.031..0.085 rows=31
loops=1)
                -> Covering index lookup on v using video_index (categoryID=c.categoryID)
 (cost=192.79 rows=4952) (actual time=0.024..1.815 rows=2336 loops=31)
```

In this performance analysis, we aimed to evaluate the impact of different indexing strategies on a complex SQL query. Specifically, we compared the default query execution to three indexing methods applied to the Video and Category tables. The following observations and conclusions can be drawn from the results:

Indexing	**Nested Loop Inner Join Cost**	**Table Scan Cost**	**Index Lookup**	**Index Lookup Cost**
No Indexing (Default)	30612.71	8081.11	PRIMARY (categoryID)	0.25
Category (categoryName)	30612.71	8081.11	PRIMARY	0.25
Video (categoryID, view_count)	20867.01	3.35	video_index	193.80
Category (categoryID, categoryName) Video (categoryID, view_count)	**20835.69**	**3.35**	video_index	192.79

Without indexing, the default query had high costs of 30612.71. Creating a covered index on the Video table for categoryID and view_count reduced cost to 20867 by allowing direct data access. Introducing combined indexes on both Category and Video tables further improved performance, with an execution cost 20835.69. Our analysis revealed the significance of proper indexing in optimizing query performance. Creating covered indexes on the attributes used for filtering and joining can notably enhance query execution speed. In this case, the covered index on categoryID and view_count in the Video table demonstrated the most efficient indexing strategy, resulting in the lowest query execution cost.