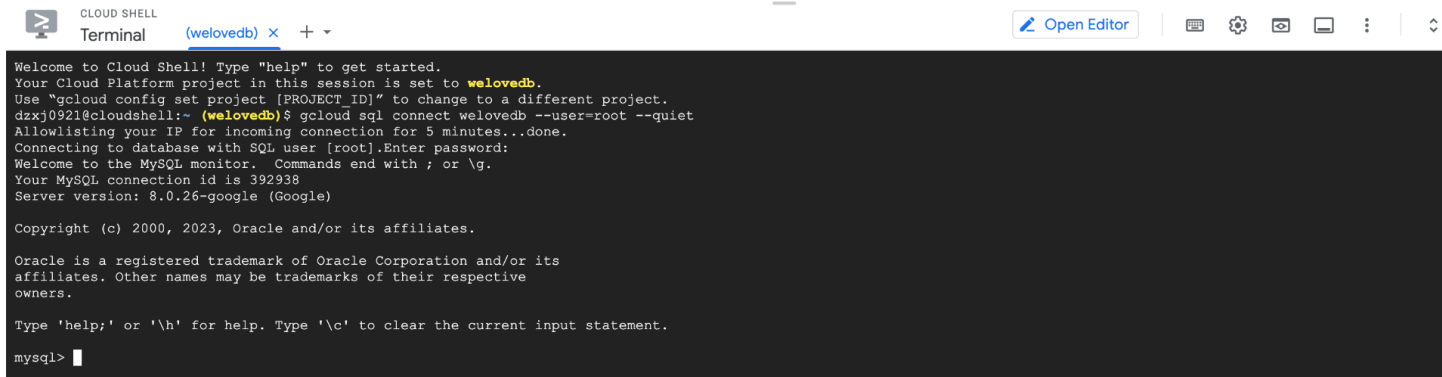


CS 411 Project Track 1 – Stage 3 Report

1) Implementation of MySQL database on Google Cloud Platform



The screenshot shows a Google Cloud Shell terminal window. The title bar includes 'CLOUD SHELL', 'Terminal', and a tab for '(welovedb)'. There is an 'Open Editor' button and several icons on the right. The terminal output shows the following text:

```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to welovedb.
Use "gcloud config set project [PROJECT ID]" to change to a different project.
dzzj0921@cloudshell:~ (welovedb)$ gcloud sql connect welovedb --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 392938
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>
```

2) DDL Commands to create the relations

```
CREATE TABLE Users (  
    user_id INT NOT NULL,  
    user_name VARCHAR(255) NOT NULL,  
    user_email VARCHAR(320) NOT NULL,  
    user_password VARCHAR(100) NOT NULL,  
    PRIMARY KEY (user_id)  
);
```

```
CREATE TABLE Tags (  
    tag_id INT NOT NULL,  
    tag_description VARCHAR(255),  
    PRIMARY KEY (tag_id)  
);
```

```
CREATE TABLE Categories (  
    category_id INT NOT NULL,  
    category_description VARCHAR(255),  
    PRIMARY KEY (category_id)  
);
```

```
CREATE TABLE Channels (  
    channel_id VARCHAR(63) NOT NULL,  
    channel_title VARCHAR(255),  
    PRIMARY KEY (channel_id)  
);
```

```
CREATE TABLE Favorite_Videos (  
    user_id INT,  
    video_id VARCHAR(255),  
    favorite_date DATE,  
    PRIMARY KEY(user_id, video_id),  
    FOREIGN KEY (user_id) REFERENCES Users(user_id),  
    FOREIGN KEY (video_id) REFERENCES Videos(video_id)  
);
```

```
CREATE TABLE Favorite_Tags (  
    user_id INT,  
    tag_id INT,  
    favorite_date DATE,  
    PRIMARY KEY(user_id, tag_id),  
    FOREIGN KEY (user_id) REFERENCES Users(user_id),  
    FOREIGN KEY (tag_id) REFERENCES Tags(tag_id)  
);
```

```
CREATE TABLE Video_Tags (  
    video_id VARCHAR(255),  
    tag_id INT,  
    PRIMARY KEY(video_id, tag_id),  
    FOREIGN KEY (video_id) REFERENCES Videos(video_id),  
    FOREIGN KEY (tag_id) REFERENCES Tags(tag_id)  
);
```

```
CREATE TABLE Videos (  
    video_id VARCHAR(255) NOT NULL,  
    video_title VARCHAR(255),  
    uploaded_date DATETIME,  
    channel_id VARCHAR(63),  
    channel_title VARCHAR(255),  
    category_id INT,  
    trending_date DATETIME,  
    view_count INT,  
    like_count INT,  
    dislike_count INT,  
    comment_count INT,  
    thumbnail_link VARCHAR(512),  
    comments_disabled BOOLEAN,
```

```

ratings_disabled BOOLEAN,
PRIMARY KEY (video_id),
FOREIGN KEY (category_id) REFERENCES Categories(category_id),
FOREIGN KEY (channel_id) REFERENCES Channels(channel_id)
);

```

As per comments from Stage 2, we altered Videos to be a weak entity set of Channels.

3) Insertion of more than 1000 records in Tags, Videos and Video_Tags

```

mysql> SELECT COUNT(*) FROM Tags;
+-----+
| COUNT(*) |
+-----+
|    192970 |
+-----+
1 row in set (0.06 sec)

```

```

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

```

```

mysql> SELECT COUNT(*) FROM Video_Tags;
+-----+
| COUNT(*) |
+-----+
|     21041 |
+-----+
1 row in set (0.00 sec)

```

4) Two advanced queries with querying results

Query 1 – finding videos which either have more likes than the average number of likes or more views than the average number of views.

```

SELECT video_title
FROM Videos
WHERE like_count > (SELECT AVG(like_count) FROM Videos)
UNION
SELECT video_title
FROM Videos
WHERE view_count > (SELECT AVG(view_count) FROM Videos)
LIMIT 15;

```

MySQL Workbench

Administration

Schemas

MANAGEMENT

Server Status

Client Connections

Users and Privileges

Status and System Variables

Data Export

Data Import/Restore

INSTANCE

Startup / Shutdown

Server Logs

Options File

PERFORMANCE

Dashboard

Performance Reports

Performance Schema Setup

Object Info

Session

No object selected

Query 1

Limit to 1000 rows

1

2

3

4

5

6

7

8

9

SELECT video_title

FROM Videos

WHERE like_count > (SELECT AVG(like_count) FROM Videos)

UNION

SELECT video_title

FROM Videos

WHERE view_count > (SELECT AVG(view_count) FROM Videos)

LIMIT 15;

Result Grid

Filter Rows: Search

Export:

video_title
Lapiz Conciente - 9 Dias
YoungBoy Never Broke Again - Kacey talk
Internet Money - Lemonade ft. Don Toliver, Gunna & Nav (Dir. by @_ColeBennett_)
Enola Holmes I Official Trailer Netflix
Billie Eilish - my future (Live)
What Really Happened
Conan Gray - Heather (Official Music Video)
My Decaying Mind in Quarantine
Juice WRLD & The Weeknd - Smile (Official Video)
Jackson Wang & Galantis - Pretty Please (Official Music Video)
Suicide Squad: Kill the Justice League Official Teaser Trailer
GAME (Full Video) Shooter Kahlon I Sidhu Moose Wala I Hunny PK Films I Gold Media I 5911 Records
This is Goodbye
I ASKED HER TO BE MY GIRLFRIEND...
DJ Khaled ft. Drake - POPSTAR (Official Music Video - Starring Justin Bieber)

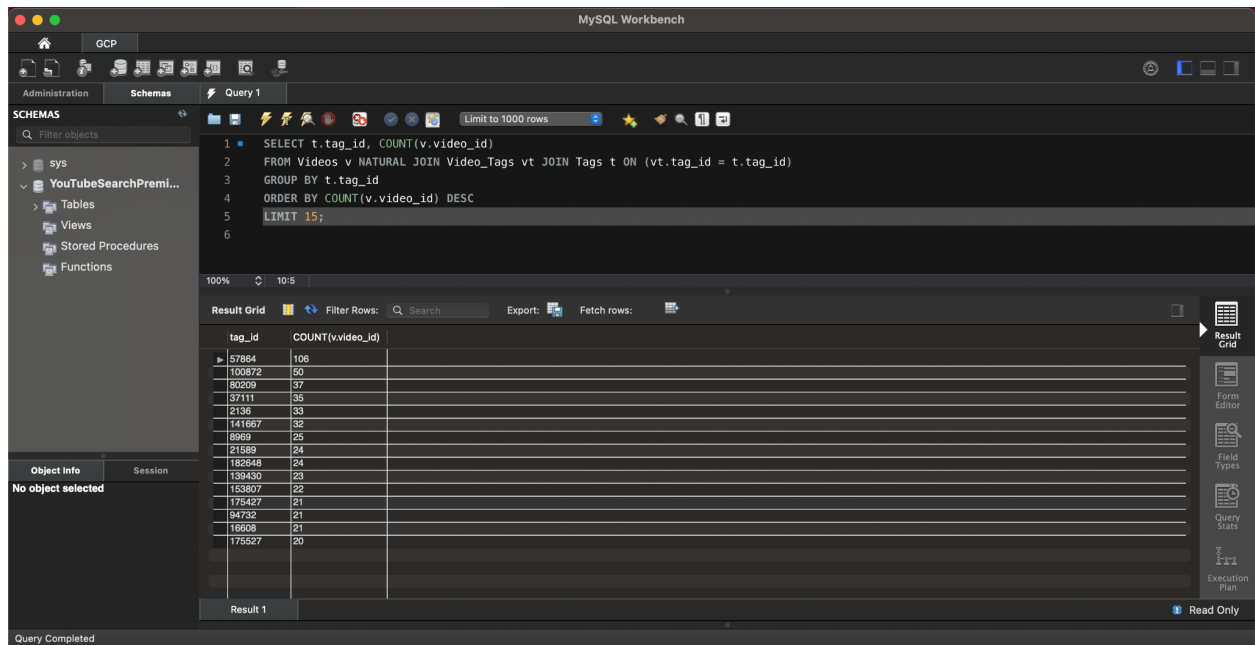
Result 3

Read Only

Query Completed

Query 2 – finding most popular tags of trending YouTube videos

```
SELECT t.tag_id, COUNT(v.video_id)
FROM Videos v NATURAL JOIN Video_Tags vt JOIN Tags t ON
(vt.tag_id = t.tag_id)
GROUP BY t.tag_id
ORDER BY COUNT(v.video_id) DESC
LIMIT 15;
```



The screenshot shows the MySQL Workbench interface. The query editor at the top contains the SQL query for finding the most popular tags. The 'Schemas' pane on the left shows the database structure. The 'Result Grid' at the bottom displays the query results in a table format.

Query:

```
1 SELECT t.tag_id, COUNT(v.video_id)
2 FROM Videos v NATURAL JOIN Video_Tags vt JOIN Tags t ON (vt.tag_id = t.tag_id)
3 GROUP BY t.tag_id
4 ORDER BY COUNT(v.video_id) DESC
5 LIMIT 15;
```

Result Grid:

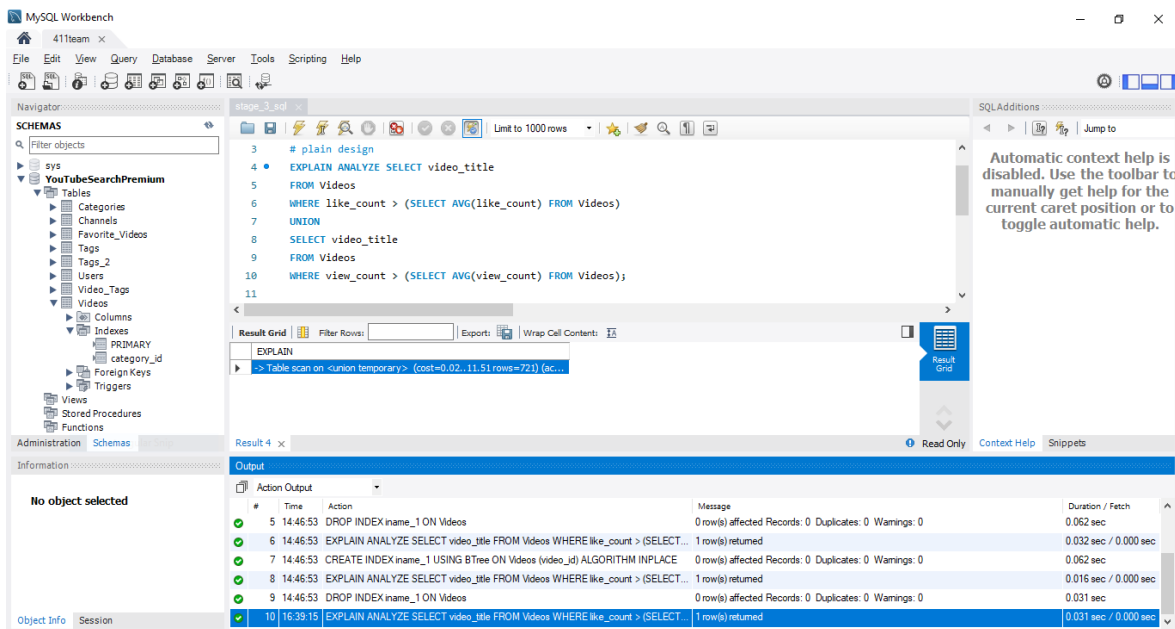
tag_id	COUNT(v.video_id)
57884	106
100872	50
80209	37
37111	35
2136	33
141867	32
8989	25
21589	24
182648	24
138430	23
153807	22
175427	21
94732	21
16608	21
175527	20

5) Indexing Design and Analysis for the advanced queries

Query 1

```
SELECT video_title
FROM Videos
WHERE like_count > (SELECT AVG(like_count) FROM Videos)
UNION
SELECT video_title
FROM Videos
WHERE view_count > (SELECT AVG(view_count) FROM Videos);
```

Default Indexing:



The screenshot shows the MySQL Workbench interface. The left sidebar displays the 'SCHEMAS' tree with 'YouTubeSearchPremium' selected. The main editor shows a query labeled 'stage_3.sql' with the following SQL code:

```
3 # plain design
4 EXPLAIN ANALYZE SELECT video_title
5 FROM Videos
6 WHERE like_count > (SELECT AVG(like_count) FROM Videos)
7 UNION
8 SELECT video_title
9 FROM Videos
10 WHERE view_count > (SELECT AVG(view_count) FROM Videos);
11
```

The 'Result Grid' tab is active, showing the execution plan for the query. The plan includes the following steps:

- EXPLAIN
- Table scan on <union temporary> (cost=0.02..11.51 rows=721) (actual time=0.001..0.019 rows=268 loops=1)
- Union materialize with deduplication (cost=156.27..167.76 rows=721) (actual time=4.056..4.089 rows=268 loops=1)
- Filter: (Videos.like_count > (select #2)) (cost=42.06 rows=361) (actual time=2.163..2.612 rows=206 loops=1)
- Table scan on Videos (cost=42.06 rows=1082) (actual time=0.692..1.021 rows=1049 loops=1)
- Select #2 (subquery in condition; run only once)
- Aggregate: avg(Videos.like_count) (cost=222.40 rows=1082) (actual time=1.393..1.393 rows=1 loops=1)

The 'Output' tab is also visible, showing the results of the query execution.

EXPLAIN

```
-> Table scan on <union temporary> (cost=0.02..11.51 rows=721) (actual
time=0.001..0.019 rows=268 loops=1)
    -> Union materialize with deduplication (cost=156.27..167.76 rows=721)
(actual time=4.056..4.089 rows=268 loops=1)
        -> Filter: (Videos.like_count > (select #2)) (cost=42.06 rows=361)
(actual time=2.163..2.612 rows=206 loops=1)
            -> Table scan on Videos (cost=42.06 rows=1082) (actual
time=0.692..1.021 rows=1049 loops=1)
                -> Select #2 (subquery in condition; run only once)
                    -> Aggregate: avg(Videos.like_count) (cost=222.40
rows=1082) (actual time=1.393..1.393 rows=1 loops=1)
```

```

-> Table scan on Videos (cost=114.20 rows=1082)
(actual time=0.028..1.253 rows=1049 loops=1)
  -> Filter: (Videos.view_count > (select #4)) (cost=42.06 rows=361)
(actual time=0.677..1.101 rows=228 loops=1)
    -> Table scan on Videos (cost=42.06 rows=1082) (actual
time=0.214..0.520 rows=1049 loops=1)
      -> Select #4 (subquery in condition; run only once)
        -> Aggregate: avg(Videos.view_count) (cost=222.40
rows=1082) (actual time=0.392..0.392 rows=1 loops=1)
          -> Table scan on Videos (cost=114.20 rows=1082)
(actual time=0.023..0.290 rows=1049 loops=1)

```

Duration: 0.031 sec / Fetch: 0.000 sec

Design 1: B+ Tree Index of Videos.video_id with in-place algorithm

The screenshot shows the MySQL Workbench interface. The SQL editor contains the following script:

```

13 CREATE INDEX iname_1 USING BTree ON Videos (video_id) ALGORITHM INPLACE;
14 # the table will use this index design
15 EXPLAIN ANALYZE SELECT video_title
16 FROM Videos
17 WHERE like_count > (SELECT AVG(like_count) FROM Videos)
18 UNION
19 SELECT video_title
20 FROM Videos
21 WHERE view_count > (SELECT AVG(view_count) FROM Videos);
22 DROP INDEX iname_1 ON Videos;

```

The SQL editor shows the execution plan for the query. The output pane displays the following results:

#	Time	Action	Message	Duration / Fetch
9	14:46:53	DROP INDEX iname_1 ON Videos	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0	0.031 sec
10	16:39:15	EXPLAIN ANALYZE SELECT video_title FROM Videos WHERE like_count > (SELECT...	1 row(s) returned	0.031 sec / 0.000 sec
11	16:47:46	CREATE INDEX iname_1 USING BTree ON Videos (video_id) ALGORITHM INPLACE	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0	0.078 sec
12	16:47:47	EXPLAIN ANALYZE SELECT video_title FROM Videos WHERE like_count > (SELECT...	1 row(s) returned	0.016 sec / 0.000 sec
13	16:47:47	DROP INDEX iname_1 ON Videos	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0	0.047 sec

EXPLAIN

```

-> Table scan on <union temporary> (cost=0.02..11.51 rows=721) (actual
time=0.001..0.021 rows=268 loops=1)
  -> Union materialize with deduplication (cost=156.27..167.76 rows=721)
(actual time=2.840..2.875 rows=268 loops=1)
    -> Filter: (Videos.like_count > (select #2)) (cost=42.06 rows=361)
(actual time=1.238..1.679 rows=206 loops=1)
      -> Table scan on Videos (cost=42.06 rows=1082) (actual
time=0.122..0.455 rows=1049 loops=1)
        -> Select #2 (subquery in condition; run only once)
          -> Aggregate: avg(Videos.like_count) (cost=222.40
rows=1082) (actual time=1.095..1.096 rows=1 loops=1)
            -> Table scan on Videos (cost=114.20 rows=1082)

```

```

(actual time=0.634..0.979 rows=1049 loops=1)
    -> Filter: (Videos.view_count > (select #4)) (cost=42.06 rows=361)
(actual time=0.449..0.867 rows=228 loops=1)
    -> Table scan on Videos (cost=42.06 rows=1082) (actual
time=0.032..0.350 rows=1049 loops=1)
    -> Select #4 (subquery in condition; run only once)
    -> Aggregate: avg(Videos.view_count) (cost=222.40
rows=1082) (actual time=0.408..0.408 rows=1 loops=1)
    -> Table scan on Videos (cost=114.20 rows=1082)
(actual time=0.022..0.296 rows=1049 loops=1)

```

Duration: 0.016 sec / Fetch: 0.000 sec

Design 2: B+ Trees Index of Videos.view_count and Videos.like_count, both with in-place algorithms

The screenshot shows the MySQL Workbench interface. The SQL editor contains the following queries:

```

25 • CREATE INDEX iname21 USING BTree ON Videos (like_count) ALGORITHM INPLACE;
26 • CREATE INDEX iname22 USING BTree ON Videos (view_count) ALGORITHM INPLACE;
27 • EXPLAIN ANALYZE SELECT video_title
28 FROM Videos
29 WHERE like_count > (SELECT AVG(like_count) FROM Videos)
30 UNION
31 SELECT video_title
32 FROM Videos
33 WHERE view_count > (SELECT AVG(view_count) FROM Videos);
34 • DROP INDEX iname21 ON Videos;
35 • DROP INDEX iname22 ON Videos;

```

The 'EXPLAIN' tab is selected, showing the execution plan for the query. The output shows a table scan on a union temporary table, followed by a filter operation.

The 'Output' tab shows the execution results:

#	Time	Action	Message	Duration / Fetch
17	17:09:23	CREATE INDEX iname21 USING BTree ON Videos (like_count) ALGORITHM INPLACE	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0	0.063 sec
18	17:09:23	CREATE INDEX iname22 USING BTree ON Videos (view_count) ALGORITHM INPLACE	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0	0.063 sec
19	17:09:23	EXPLAIN ANALYZE SELECT video_title FROM Videos WHERE like_count > (SELECT ...)	1 row(s) returned	0.016 sec / 0.016 sec
20	17:09:23	DROP INDEX iname21 ON Videos	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0	0.047 sec

EXPLAIN

```

-> Table scan on <union temporary> (cost=0.02..7.92 rows=434) (actual
time=0.002..0.019 rows=268 loops=1)
    -> Union materialize with deduplication (cost=239.24..247.15 rows=434)
(actual time=6.396..6.429 rows=268 loops=1)
    -> Filter: (Videos.like_count > (select #2)) (cost=92.96 rows=206)
(actual time=0.764..3.461 rows=206 loops=1)
    -> Index range scan on Videos using iname21 (cost=92.96
rows=206) (actual time=0.759..3.425 rows=206 loops=1)
    -> Select #2 (subquery in condition; run only once)
    -> Aggregate: avg(Videos.like_count) (cost=222.40
rows=1082) (actual time=0.346..0.346 rows=1 loops=1)

```



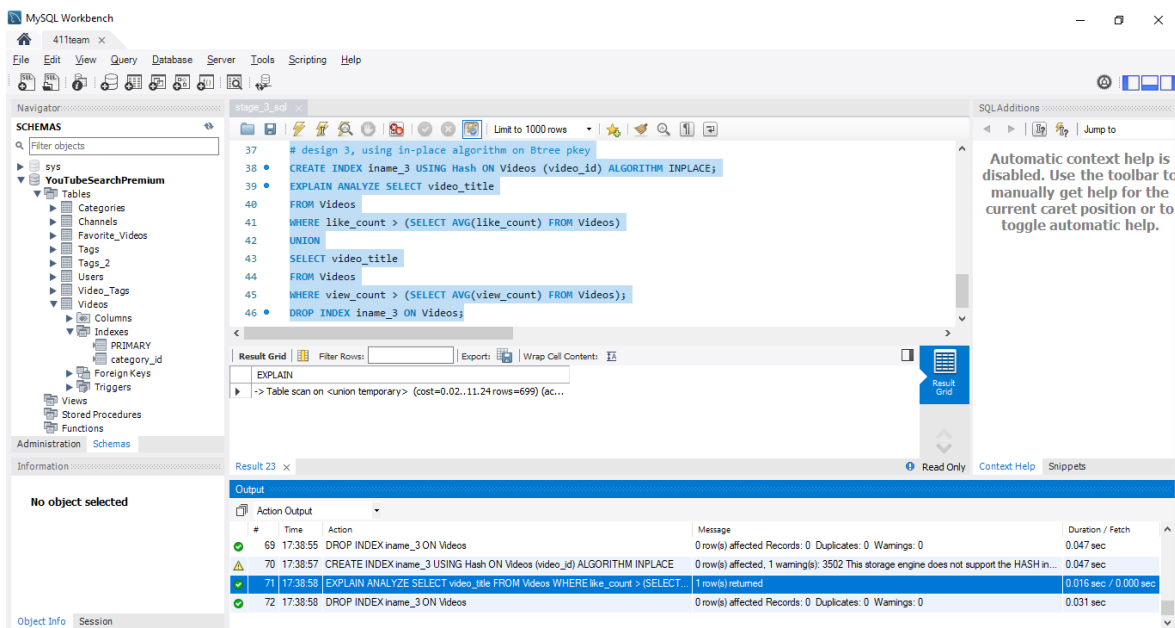
```

-> Index scan on Videos using iname21 (cost=114.20
rows=1082) (actual time=0.041..0.246 rows=1049 loops=1)
  -> Filter: (Videos.view_count > (select #4)) (cost=102.86
rows=228) (actual time=2.254..2.586 rows=228 loops=1)
    -> Index range scan on Videos using iname22 (cost=102.86
rows=228) (actual time=2.250..2.550 rows=228 loops=1)
      -> Select #4 (subquery in condition; run only once)
        -> Aggregate: avg(Videos.view_count) (cost=222.40
rows=1082) (actual time=0.322..0.322 rows=1 loops=1)
          -> Index scan on Videos using iname22 (cost=114.20
rows=1082) (actual time=0.029..0.213 rows=1049 loops=1)

```

Duration: 0.016 sec / Fetch: 0.016 sec

Design 3: Hash Table Index of Videos.video_id with in-place algorithm



The screenshot shows the MySQL Workbench interface. The SQL editor contains the following script:

```

37 # design 3, using in-place algorithm on Btree pkey
38 CREATE INDEX iname_3 USING Hash ON Videos (video_id) ALGORITHM INPLACE;
39 EXPLAIN ANALYZE SELECT video_title
40 FROM Videos
41 WHERE like_count > (SELECT AVG(like_count) FROM Videos)
42 UNION
43 SELECT video_title
44 FROM Videos
45 WHERE view_count > (SELECT AVG(view_count) FROM Videos);
46 DROP INDEX iname_3 ON Videos;

```

The output pane shows the execution plan for the query:

```

EXPLAIN
-> Table scan on <union temporary> (cost=0.02..11.24 rows=699) (actual
time=0.001..0.019 rows=268 loops=1)
  -> Union materialize with deduplication (cost=149.37..160.59 rows=699)
(actual time=2.091..2.124 rows=268 loops=1)
    -> Filter: (Videos.like_count > (select #2)) (cost=39.71 rows=350)
(actual time=0.444..0.963 rows=206 loops=1)
      -> Table scan on Videos (cost=39.71 rows=1049) (actual
time=0.046..0.453 rows=1049 loops=1)
        -> Select #2 (subquery in condition; run only once)
          -> Aggregate: avg(Videos.like_count) (cost=214.55
rows=1049) (actual time=0.385..0.385 rows=1 loops=1)

```

The action output pane shows the execution details for each step:

#	Time	Action	Message	Duration / Fetch
69	17:38:55	DROP INDEX iname_3 ON Videos	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0	0.047 sec
70	17:38:57	CREATE INDEX iname_3 USING Hash ON Videos (video_id) ALGORITHM INPLACE	0 row(s) affected. 1 warning(s): 3502 This storage engine does not support the HASH in...	0.047 sec
71	17:38:58	EXPLAIN ANALYZE SELECT video_title FROM Videos WHERE like_count > (SELECT...	1 row(s) returned	0.016 sec / 0.000 sec
72	17:38:58	DROP INDEX iname_3 ON Videos	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0	0.031 sec

EXPLAIN

```

-> Table scan on <union temporary> (cost=0.02..11.24 rows=699) (actual
time=0.001..0.019 rows=268 loops=1)
  -> Union materialize with deduplication (cost=149.37..160.59 rows=699)
(actual time=2.091..2.124 rows=268 loops=1)
    -> Filter: (Videos.like_count > (select #2)) (cost=39.71 rows=350)
(actual time=0.444..0.963 rows=206 loops=1)
      -> Table scan on Videos (cost=39.71 rows=1049) (actual
time=0.046..0.453 rows=1049 loops=1)
        -> Select #2 (subquery in condition; run only once)
          -> Aggregate: avg(Videos.like_count) (cost=214.55
rows=1049) (actual time=0.385..0.385 rows=1 loops=1)

```

```

-> Table scan on Videos (cost=109.65 rows=1049)
(actual time=0.018..0.289 rows=1049 loops=1)
  -> Filter: (Videos.view_count > (select #4)) (cost=39.71 rows=350)
(actual time=0.408..0.827 rows=228 loops=1)
    -> Table scan on Videos (cost=39.71 rows=1049) (actual
time=0.025..0.339 rows=1049 loops=1)
      -> Select #4 (subquery in condition; run only once)
        -> Aggregate: avg(Videos.view_count) (cost=214.55
rows=1049) (actual time=0.376..0.376 rows=1 loops=1)
          -> Table scan on Videos (cost=109.65 rows=1049)
(actual time=0.018..0.273 rows=1049 loops=1)

```

Duration: 0.016 sec / Fetch: 0.000 sec

Analysis: The default indexing executes the query around 31 milliseconds. When using the in-place algorithm option, the execution time is halved to 16 milliseconds, regardless of using B+ Tree or hashing as the indexing implementation.

One reason to explain this in-place advantage is our query objective is that COPY algorithm would generate overhead undo/redo logging statements for the query, and not having pre-sorted secondary indices (according to MySQL documentation <https://dev.mysql.com/doc/refman/8.0/en/innodb-online-ddl-operations.html>), Even though the `INPLACE` algorithm still requires copying to a clustered indexing structure, not having these extra tasks resulted in a faster query.

Also, one key difference between indexing on primary key (Design 1) and indexing on the direct searching criteria (Design 2) is that primary key is unique and naturally clustered, but other columns such as `view_count` or `like_count` do not have these properties. Even though Design 2 indexed `view_count` and `like_count` directly, the unclustered nature hindered the advantage of direct access, resulting in on par querying efficiency as compared to Design 1.

Lastly, the hashing attempt. Google Cloud Platform uses InnoDB to support MySQL queries, and the hashing implementation of indexing is not possible by design. The remedy is to use the `storage engine default` which uses clustered primary key and the design of [adaptive hashing](#), which has similar performance to a B+ tree indexing design.

Our group will choose Design 1 for its simplicity (no overhead computations) and straightforwardness directly from the primary keys. The drawbacks of Design 2 is that the trick of indexing the selection criteria does not improve as much as we would expect. The folly of design #3 is that the default engine on GCP is not really

using a hash table at all, due to systematic memory optimization issues.

Query 2:

```
SELECT t.tag_id, COUNT(v.video_id)
FROM Videos v NATURAL JOIN Video_Tags vt JOIN Tags t ON
(vt.tag_id = t.tag_id)
GROUP BY t.tag_id
ORDER BY COUNT(v.video_id) DESC;
```

Default:

```
-> Sort: `COUNT(v.video_id)` DESC (actual time=60.924..61.788 rows=14562
loops=1)
    -> Table scan on <temporary> (actual time=0.001..0.671 rows=14562
loops=1)
        -> Aggregate using temporary table (actual time=53.108..54.641
rows=14562 loops=1)
            -> Nested loop inner join (cost=9236.50 rows=18095) (actual
time=0.071..42.589 rows=21041 loops=1)
                -> Nested loop inner join (cost=2903.12 rows=18095)
(actual time=0.062..12.527 rows=21041 loops=1)
                    -> Index scan on v using category_id (cost=114.20
rows=1082) (actual time=0.040..0.340 rows=1049 loops=1)
                        -> Index lookup on vt using PRIMARY
(video_id=v.video_id) (cost=0.91 rows=17) (actual time=0.005..0.010
rows=20 loops=1049)
                            -> Single-row index lookup on t using PRIMARY
(tag_id=vt.tag_id) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1
loops=21041)
```

Duration: 0.078 s

Design 1: Index on Videos.video_id, Video_Tags.tag_id, Tags.tag_id

```
1 • USE YouTubeSearchPremium;
2
3 • CREATE INDEX idx_video_tags_tag_id ON Video_Tags(tag_id);
4 • CREATE INDEX idx_videos_video_id ON Videos(video_id);
5 • CREATE INDEX idx_tags_tag_id ON Tags(tag_id);
6
7 • EXPLAIN ANALYZE SELECT t.tag_id, COUNT(v.video_id)
8 FROM Videos v NATURAL JOIN Video_Tags vt JOIN Tags t ON (vt.tag_id = t.tag_id)
9 GROUP BY t.tag_id
0 ORDER BY COUNT(v.video_id) DESC;
1
2
3
4
```

```

-> Sort: `COUNT(v.video_id)` DESC (actual time=63.321..64.139 rows=14562
loops=1)
    -> Table scan on <temporary> (actual time=0.002..0.668 rows=14562
loops=1)
        -> Aggregate using temporary table (actual time=55.550..57.089
rows=14562 loops=1)
            -> Nested loop inner join (cost=9236.50 rows=18095) (actual
time=0.088..43.390 rows=21041 loops=1)
                -> Nested loop inner join (cost=2903.12 rows=18095)
(actual time=0.078..12.748 rows=21041 loops=1)
                    -> Index scan on v using category_id (cost=114.20
rows=1082) (actual time=0.053..0.393 rows=1049 loops=1)
                        -> Index lookup on vt using PRIMARY
(video_id=v.video_id) (cost=0.91 rows=17) (actual time=0.005..0.010
rows=20 loops=1049)
                            -> Single-row index lookup on t using PRIMARY
(tag_id=vt.tag_id) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1
loops=21041)

```

Duration: 0.078 s

Design 2: (Hash Table index on Video_Tags.tag_id and Video_Tags.video_id)

```

1 • USE YouTubeSearchPremium;
2
3 • CREATE INDEX idx_video_tags_tag_id_video_id USING hash ON Video_Tags(tag_id, video_id);
4
5 • EXPLAIN ANALYZE SELECT t.tag_id, COUNT(v.video_id)
6 FROM Videos v NATURAL JOIN Video_Tags vt JOIN Tags t ON (vt.tag_id = t.tag_id)
7 GROUP BY t.tag_id
8 ORDER BY COUNT(v.video_id) DESC;
9
10
11
-> Sort: `COUNT(v.video_id)` DESC (actual time=55.713..56.516 rows=14562
loops=1)
    -> Table scan on <temporary> (actual time=0.001..0.661 rows=14562
loops=1)
        -> Aggregate using temporary table (actual time=50.087..51.614
rows=14562 loops=1)
            -> Nested loop inner join (cost=9236.50 rows=18095) (actual
time=0.083..40.775 rows=21041 loops=1)
                -> Nested loop inner join (cost=2903.12 rows=18095)
(actual time=0.074..12.048 rows=21041 loops=1)
                    -> Index scan on v using category_id (cost=114.20
rows=1082) (actual time=0.049..0.323 rows=1049 loops=1)
                        -> Index lookup on vt using PRIMARY
(video_id=v.video_id) (cost=0.91 rows=17) (actual time=0.005..0.010
rows=20 loops=1049)
                            -> Single-row index lookup on t using PRIMARY
(tag_id=vt.tag_id) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1
loops=21041)

```

Duration: 0.078 s

Design 3: Index of Videos.video_id and Video_Tags.video_id

```
1 • USE YouTubeSearchPremium;
2
3 • CREATE INDEX idx_videos_video_id ON Videos(video_id);
4 • CREATE INDEX idx_video_tags_video_id ON Video_Tags(video_id);
5
6 • EXPLAIN ANALYZE SELECT t.tag_id, COUNT(v.video_id)
7 FROM Videos v NATURAL JOIN Video_Tags vt JOIN Tags t ON (vt.tag_id = t.tag_id)
8 GROUP BY t.tag_id
9 ORDER BY COUNT(v.video_id) DESC;
10
11
12
13
```

```
-> Sort: `COUNT(v.video_id)` DESC (actual time=55.441..56.235 rows=14562
loops=1)
-> Table scan on <temporary> (actual time=0.002..0.642 rows=14562
loops=1)
-> Aggregate using temporary table (actual time=49.842..51.324
rows=14562 loops=1)
-> Nested loop inner join (cost=8953.72 rows=17543) (actual
time=0.075..40.313 rows=21041 loops=1)
-> Nested loop inner join (cost=2813.51 rows=17543)
(actual time=0.065..11.753 rows=21041 loops=1)
-> Index scan on v using category_id (cost=109.65
rows=1049) (actual time=0.040..0.300 rows=1049 loops=1)
-> Index lookup on vt using PRIMARY
(video_id=v.video_id) (cost=0.91 rows=17) (actual time=0.005..0.010
rows=20 loops=1049)
-> Single-row index lookup on t using PRIMARY
(tag_id=vt.tag_id) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1
loops=21041)
)
```

Duration: 0.078 s

Analysis: Of the four indexing schemes, the two best were designs 2 and 3. Looking at the output from EXPLAIN ANALYZE, we can see that both take around 55-56 milliseconds to complete, while the default indexing and design 1 take around 60-64 milliseconds. I believe that it is improved due to the fact that the fields we created the indexes on are being used during the joins. Design 2 makes the JOIN between Video_Tags and Tags more efficient, while Design 3 makes the NATURAL JOIN on Videos and Video_Tags faster.

