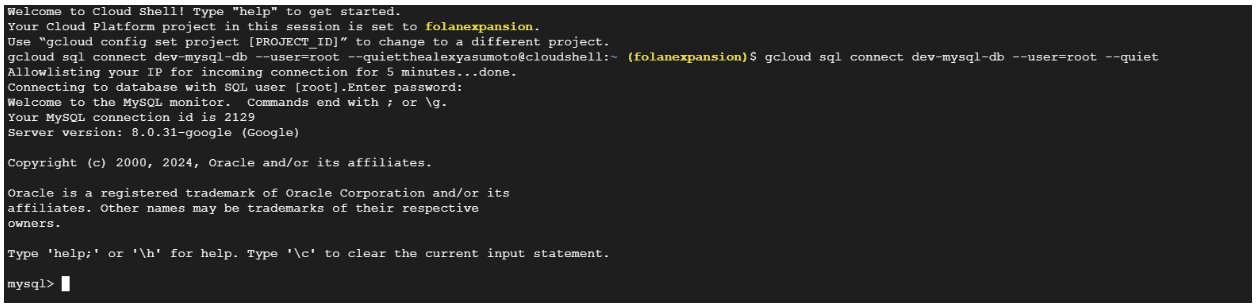
Our database connected:



```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to folanexpansion.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
gcloud sql connect dev-mysql-db --user=root --quietthealexyasumoto@cloudshell:~ (folanexpansion)$ gcloud sql connect dev-mysql-db --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 2129
Server version: 8.0.31-google (Google)

Copyright (c) 2000, 2024, 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>
```

5 Main Tables:

User, Papers, Authors, Leaderboards, Search_History

Junction Tables:

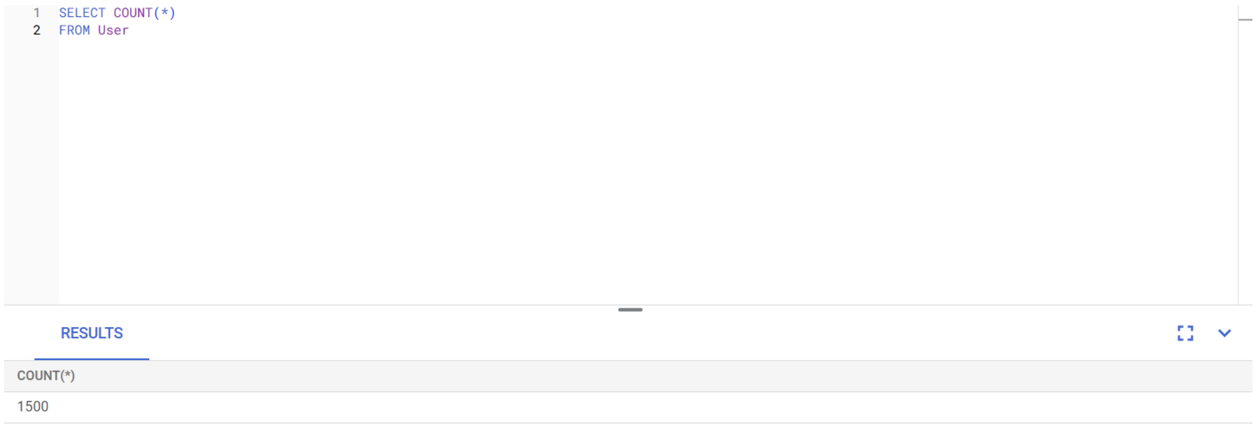
Writes, Likes, AppearsIn

DDL Commands For Each Table

```
CREATE TABLE User (
   user_id INT NOT NULL AUTO_INCREMENT,
   username VARCHAR(50) NOT NULL,
   email VARCHAR(100) NOT NULL UNIQUE,
   password VARCHAR(255) NOT NULL,
   PRIMARY KEY (user_id)
);

CREATE TABLE Papers (
   paper_id INT NOT NULL AUTO_INCREMENT,
   title TEXT NOT NULL,
   abstract TEXT NOT NULL,
   journal_name VARCHAR(255),
   citation_num INT NOT NULL,
   cluster_id INT NOT NULL,
   PRIMARY KEY (paper_id)
);

CREATE TABLE Authors (
   author_id INT NOT NULL AUTO_INCREMENT,
   first_name VARCHAR(255) NOT NULL,
   last_name VARCHAR(255) NOT NULL,
   PRIMARY KEY (author_id)
);

CREATE TABLE Writes (
   author_id INT NOT NULL,
   paper_id INT NOT NULL,
   PRIMARY KEY (author_id, paper_id),
   FOREIGN KEY (author_id) REFERENCES Authors(author_id) ON DELETE CASCADE,
   FOREIGN KEY (paper_id) REFERENCES Papers(paper_id) ON DELETE CASCADE
);

CREATE TABLE Leaderboards (
   leaderboard_id INT AUTO_INCREMENT,
   time_period_days INT NOT NULL,
   ranking_date DATE NOT NULL,
   PRIMARY KEY (leaderboard_id)
);



CREATE TABLE Search_History (
   search_id INT AUTO_INCREMENT,
   search_query VARCHAR(100) NOT NULL,
   search_date TIMESTAMP NOT NULL,
   results_count INT NOT NULL,
   user_id INT NOT NULL,
   PRIMARY KEY (search_id),
   FOREIGN KEY (user_id) REFERENCES User(user_id) ON DELETE CASCADE
);

CREATE TABLE Likes (
   user_id INT NOT NULL,
   paper_id INT NOT NULL,
   time_liked TIMESTAMP,
   PRIMARY KEY (user_id, paper_id),
   FOREIGN KEY (user_id) REFERENCES User(user_id) ON DELETE CASCADE,
   FOREIGN KEY (paper_id) REFERENCES Papers(paper_id) ON DELETE CASCADE
);


CREATE TABLE AppearsIn (
   paper_id INT NOT NULL,
   leaderboard_id INT NOT NULL,
   ranking INT NOT NULL,
   PRIMARY KEY (paper_id, leaderboard_id),
   FOREIGN KEY (paper_id) REFERENCES Papers(paper_id) ON DELETE CASCADE,
   FOREIGN KEY (leaderboard_id) REFERENCES Leaderboards(leaderboard_id) ON DELETE CASCADE
);
```

Tables with > 1000 rows:

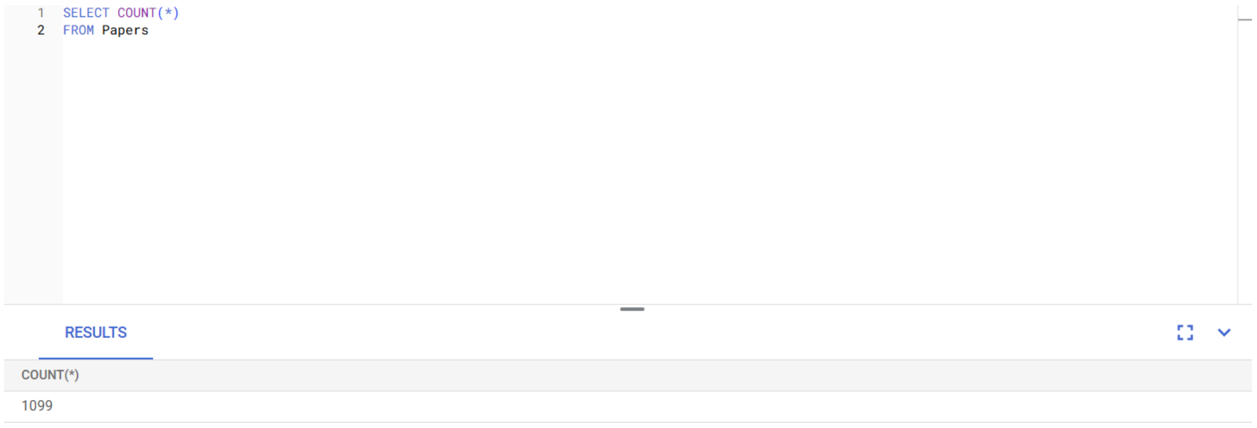
User



```
SELECT COUNT(*)
FROM User
```

```
COUNT(*)
1500
```

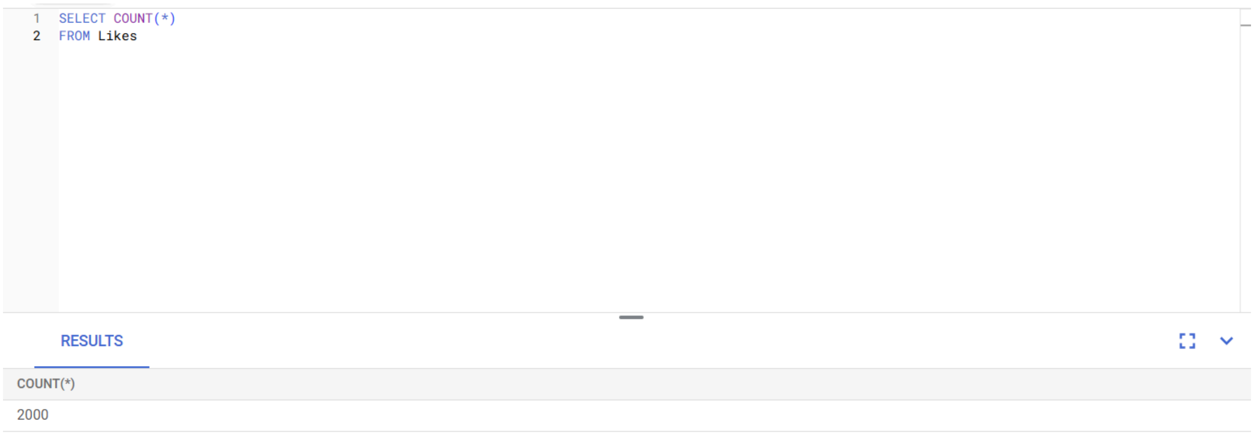
Papers



```
SELECT COUNT(*)
FROM Papers
```

```
COUNT(*)
1099
```

Likes



```sql
SELECT COUNT(*)
FROM Likes
```

```
COUNT(*)
2000
```

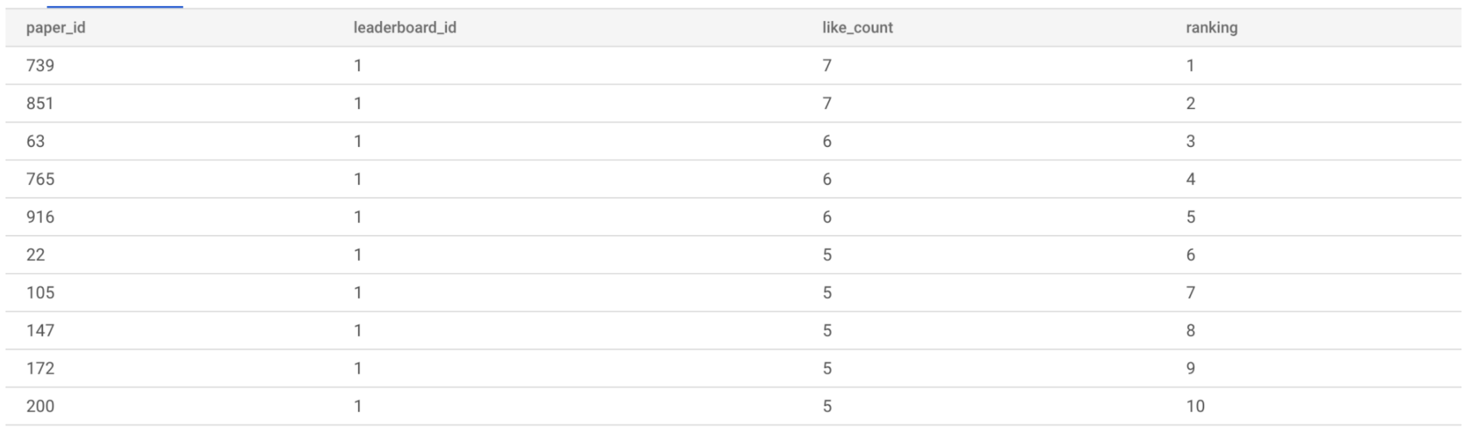
Advanced SQL Queries:

1:

This query determines the ten top ranked papers that will appear in the given leaderboard. For the sake of this demonstration, we used leaderboard 1. The purpose of this command is to then take the 10 papers returned and add them to the AppearsIn table (with appropriate paper_id, leaderboard_id, and ranking).

```
SELECT
   p.paper_id,
   l.leaderboard_id,
   COUNT(li.paper_id) AS like_count,
   ROW_NUMBER() OVER (ORDER BY COUNT(li.paper_id) DESC) ranking
FROM
   Papers p
JOIN Likes li ON p.paper_id = li.paper_id
JOIN Leaderboards l ON l.leaderboard_id = 1
WHERE
   li.time_liked BETWEEN DATE_SUB(l.ranking_date, INTERVAL l.time_period_days DAY) AND
l.ranking_date
GROUP BY
   p.paper_id, l.leaderboard_id
ORDER BY
   like_count DESC
LIMIT 10;
```

Output from our dataset is below. Note: This query is only meant to return 10 rows (since we'd only obtain the top 10 liked papers for a given leaderboard)

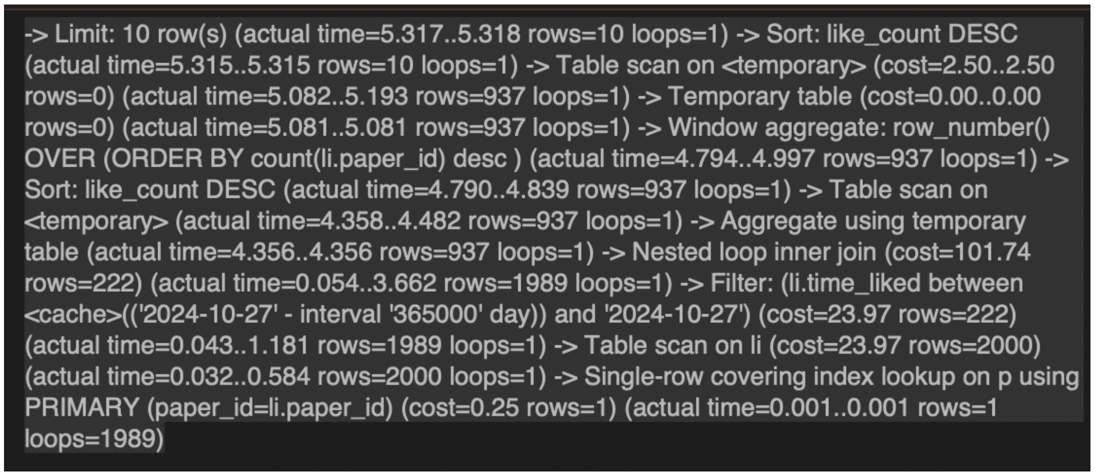


paper_id	leaderboard_id	like_count	ranking
739	1	7	1
851	1	7	2
63	1	6	3
765	1	6	4
916	1	6	5
22	1	5	6
105	1	5	7
147	1	5	8
172	1	5	9
200	1	5	10

Indexing:

We created a few different indexes in an effort to improve our query's performance.

For reference, here is the baseline result of the EXPLAIN ANALYZE command:



```
-> Limit: 10 row(s) (actual time=5.317..5.318 rows=10 loops=1) -> Sort: like_count DESC
(actual time=5.315..5.315 rows=10 loops=1) -> Table scan on <temporary> (cost=2.50..2.50
rows=0) (actual time=5.082..5.193 rows=937 loops=1) -> Temporary table (cost=0.00..0.00
rows=0) (actual time=5.081..5.081 rows=937 loops=1) -> Window aggregate: row_number()
OVER (ORDER BY count(li.paper_id) desc ) (actual time=4.794..4.997 rows=937 loops=1) ->
Sort: like_count DESC (actual time=4.790..4.839 rows=937 loops=1) -> Table scan on
<temporary> (actual time=4.358..4.482 rows=937 loops=1) -> Aggregate using temporary
table (actual time=4.356..4.356 rows=937 loops=1) -> Nested loop inner join (cost=101.74
rows=222) (actual time=0.054..3.662 rows=1989 loops=1) -> Filter: (li.time_liked between
<cache>(('2024-10-27' - interval '365000' day)) and '2024-10-27') (cost=23.97 rows=222)
(actual time=0.043..1.181 rows=1989 loops=1) -> Table scan on li (cost=23.97 rows=2000)
(actual time=0.032..0.584 rows=2000 loops=1) -> Single-row covering index lookup on p using
PRIMARY (paper_id=li.paper_id) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1
loops=1989)
```

Index 1:
We created the following index since we access the time_liked attribute in the WHERE clause of our query.

```
CREATE INDEX idx_likes_timeliked ON Likes (time_liked);
```

Here is the result of the EXPLAIN ANALYZE command after adding this index:

```
-> Limit: 10 row(s) (actual time=5.799..5.800 rows=10 loops=1) -> Sort: like_count DESC (actual time=5.798..5.798 rows=10 loops=1) -> Table scan on <temporary> (cost=2.50..2.50
rows=0) (actual time=5.414..5.542 rows=937 loops=1) -> Temporary table (cost=0.00..0.00 rows=0) (actual time=5.413..5.413 rows=937 loops=1) -> Window aggregate: row_number()
OVER (ORDER BY count(li.paper_id) desc ) (actual time=5.020..5.238 rows=937 loops=1) -> Sort: like_count DESC (actual time=5.015..5.068 rows=937 loops=1) -> Table scan on
<temporary> (actual time=4.428..4.544 rows=937 loops=1) -> Aggregate using temporary table (actual time=4.426..4.426 rows=937 loops=1) -> Nested loop inner join (cost=101.74
rows=222) (actual time=0.082..3.703 rows=1989 loops=1) -> Filter: (li.time_liked between <cache>(('2024-10-27' - interval '365000' day)) and '2024-10-27') (cost=23.97 rows=222) (actual
time=0.065..1.158 rows=1989 loops=1) -> Covering index scan on li using idx_likes_timeliked (cost=23.97 rows=2000) (actual time=0.054..0.604 rows=2000 loops=1) -> Single-row
covering index lookup on p using PRIMARY (paper_id=li.paper_id) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1989)
```

Unfortunately, this index did not improve our cost at all (across the multiple different parts of the query). We suspect that this is because our dataset isn't particularly large (as our User, Papers, and Likes tables are all under 2000 rows). When tables aren't large enough, it can appear as though adding an index isn't improving performance. This is likely the reason why our costs are the same before and after adding the index.

Index 2:

We created the following index since we access the ranking_date attribute in the WHERE clause of our query.

```
CREATE INDEX idx_leaderboards_ranking_date ON Leaderboards (ranking_date);
```

This second design is using two indexes:
`idx_likes_timeliked` and `idx_leaderboards_ranking_date`

Here is the result of the EXPLAIN ANALYZE command after adding this index:

```
-> Limit: 10 row(s) (actual time=6.644..6.646 rows=10 loops=1) -> Sort: like_count DESC (actual time=6.643..6.644 rows=10 loops=1) -> Table scan on <temporary> (cost=2.50..2.50
rows=0) (actual time=6.213..6.323 rows=937 loops=1) -> Temporary table (cost=0.00..0.00 rows=0) (actual time=6.213..6.213 rows=937 loops=1) -> Window aggregate: row_number()
OVER (ORDER BY count(li.paper_id) desc ) (actual time=5.916..6.120 rows=937 loops=1) -> Sort: like_count DESC (actual time=5.911..5.957 rows=937 loops=1) -> Table scan on
<temporary> (actual time=5.390..5.501 rows=937 loops=1) -> Aggregate using temporary table (actual time=5.388..5.388 rows=937 loops=1) -> Nested loop inner join (cost=101.74
rows=222) (actual time=0.357..4.635 rows=1989 loops=1) -> Filter: (li.time_liked between <cache>(('2024-10-27' - interval '365000' day)) and '2024-10-27') (cost=23.97 rows=222) (actual
time=0.241..1.419 rows=1989 loops=1) -> Covering index scan on li using idx_likes_timeliked (cost=23.97 rows=2000) (actual time=0.214..0.853 rows=2000 loops=1) -> Single-row
covering index lookup on p using PRIMARY (paper_id=li.paper_id) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1989)
```

Once again, we notice that the cost does not change. This is likely once again due to the small sizes of our tables, which is causing our index to prove unhelpful in this use case. However, we can reasonably deduce that the index would be helpful in a much larger dataset.

Index 3:

We created the following index since we access the time_period_days attribute in the WHERE clause of our query.

```
CREATE INDEX idx_leaderboards_time_period_days ON Leaderboards (time_period_days);
```

This third design is now using three indexes:
`idx_likes_timeliked, idx_leaderboards_ranking_date, AND idx_leaderboards_time_period_days`

Here is the result of the EXPLAIN ANALYZE command after adding this index:

```
-> Limit: 10 row(s) (actual time=6.595..6.596 rows=10 loops=1) -> Sort: like_count DESC (actual time=6.593..6.594 rows=10 loops=1) -> Table scan on <temporary> (cost=2.50..2.50
rows=0) (actual time=6.111..6.246 rows=937 loops=1) -> Temporary table (cost=0.00..0.00 rows=0) (actual time=6.110..6.110 rows=937 loops=1) -> Window aggregate: row_number()
OVER (ORDER BY count(li.paper_id) desc ) (actual time=5.775..5.988 rows=937 loops=1) -> Sort: like_count DESC (actual time=5.769..5.820 rows=937 loops=1) -> Table scan on
<temporary> (actual time=5.181..5.339 rows=937 loops=1) -> Aggregate using temporary table (actual time=5.177..5.177 rows=937 loops=1) -> Nested loop inner join (cost=101.74
rows=222) (actual time=0.068..4.300 rows=1989 loops=1) -> Filter: (li.time_liked between <cache>(('2024-10-27' - interval '365000' day)) and '2024-10-27') (cost=23.97 rows=222) (actual
time=0.054..1.234 rows=1989 loops=1) -> Covering index scan on li using idx_likes_timeliked (cost=23.97 rows=2000) (actual time=0.047..0.650 rows=2000 loops=1) -> Single-row
covering index lookup on p using PRIMARY (paper_id=li.paper_id) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1989)
```

Yet again, we notice that the cost stays stagnant and does not go down. We once again infer that this is due to the tables not being large enough.

Final Index Design:

We decided that the best final index design would likely be the first index design, which only contains `idx_likes_timeliked`. This is because the other two indexes wouldn't help that much since the Leaderboards table will never be that large, compared to the Likes table. This means that indexing the time_liked attribute is far more helpful due to the large Likes table compared to also indexing time_period_days and ranking_date in the small Leaderboards table.

Best final index design contains the following additional index:

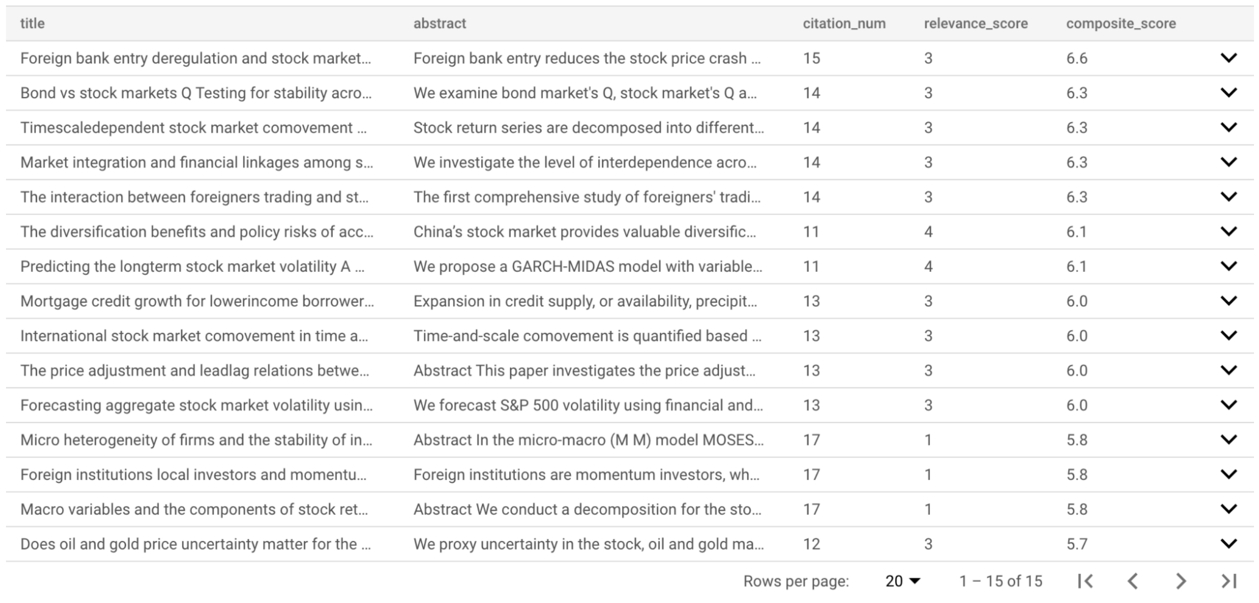
```
idx_likes_timeliked
```

2:
This query does the following:

1. **Identifies papers that contain at least one of the three keywords ("stock market," "economy," "housing") in either their title or abstract.**
2. **Scores and ranks those papers based on how many times and where the keywords appear (with more weight given to matches in the title) and how often the paper has been cited.**
3. **Returns the top 20 papers that are the most relevant based on a combination of keyword relevance and citation count, giving priority to papers that match all three keywords.**

Query on Next Page

```
WITH KeywordMatches AS (
   SELECT
       p.*,
       (
           (CASE WHEN p.title LIKE '%stock market%' THEN 2 ELSE 0 END +
            CASE WHEN p.abstract LIKE '%stock market%' THEN 1 ELSE 0 END +
            CASE WHEN p.title LIKE '%economy%' THEN 2 ELSE 0 END +
            CASE WHEN p.abstract LIKE '%economy%' THEN 1 ELSE 0 END +
            CASE WHEN p.title LIKE '%housing%' THEN 2 ELSE 0 END +
            CASE WHEN p.abstract LIKE '%housing%' THEN 1 ELSE 0 END
           )
       ) AS relevance_score,
       (
           (CASE WHEN p.title LIKE '%stock market%' OR p.abstract LIKE '%stock market%' THEN 1
ELSE 0 END) +
           (CASE WHEN p.title LIKE '%economy%' OR p.abstract LIKE '%economy%' THEN 1 ELSE 0 END)
+
           (CASE WHEN p.title LIKE '%housing%' OR p.abstract LIKE '%housing%' THEN 1 ELSE 0 END)
       ) AS keywords_matched_count
   FROM
       Papers p
   WHERE
       (p.title LIKE '%stock market%' OR p.abstract LIKE '%stock market%')
       OR
       (p.title LIKE '%economy%' OR p.abstract LIKE '%economy%')
       OR
       (p.title LIKE '%housing%' OR p.abstract LIKE '%housing%')
)

SELECT
   km.title,
   km.abstract,
   km.citation_num,
   km.relevance_score,
   (0.7 * km.relevance_score) + (0.3 * km.citation_num) AS composite_score
FROM
   KeywordMatches km
ORDER BY
   (CASE WHEN km.keywords_matched_count = 3 THEN 1 ELSE 0 END) DESC,
   composite_score DESC
LIMIT 15;
```



title	abstract	citation_num	relevance_score	composite_score
Foreign bank entry deregulation and stock market...	Foreign bank entry reduces the stock price crash ...	15	3	6.6
Bond vs stock markets Q Testing for stability acro...	We examine bond market's Q, stock market's Q a...	14	3	6.3
Timescaledependent stock market comovement ...	Stock return series are decomposed into different...	14	3	6.3
Market integration and financial linkages among s...	We investigate the level of interdependence acro...	14	3	6.3
The interaction between foreigners trading and st...	The first comprehensive study of foreigners' tradi...	14	3	6.3
The diversification benefits and policy risks of acc...	China's stock market provides valuable diversific...	11	4	6.1
Predicting the longterm stock market volatility A ...	We propose a GARCH-MIDAS model with variable...	11	4	6.1
Mortgage credit growth for lowerincome borrower...	Expansion in credit supply, or availability, precipit...	13	3	6.0
International stock market comovement in time a...	Time-and-scale comovement is quantified based ...	13	3	6.0
The price adjustment and leadlag relations betwe...	Abstract This paper investigates the price adjust...	13	3	6.0
Forecasting aggregate stock market volatility usin...	We forecast S&P 500 volatility using financial and...	13	3	6.0
Micro heterogeneity of firms and the stability of in...	Abstract In the micro-macro (M M) model MOSES...	17	1	5.8
Foreign institutions local investors and momentu...	Foreign institutions are momentum investors, wh...	17	1	5.8
Macro variables and the components of stock ret...	Abstract We conduct a decomposition for the sto...	17	1	5.8
Does oil and gold price uncertainty matter for the ...	We proxy uncertainty in the stock, oil and gold ma...	12	3	5.7

Rows per page: 20 1 – 15 of 15

Original ANALYZE:

```
-> Limit: 15 row(s) (cost=121.15 rows=15) (actual time=79.385..79.412 rows=15 loops=1) -> Sort: (case when ((((case when ((p.title like '%stock market%') or (p.abstract like '%stock market%')) then 1 else 0 end) + (case when ((p.title like '%economy%') or (p.abstract like '%economy%')) then 1 else 0 end)) + (case when ((p.title like '%housing%') or (p.abstract like '%housing%')) then 1 else 0 end)) = 3) then 1 else 0 end) DESC, composite_score DESC, limit input to 15 row(s) per chunk (cost=121.15 rows=969) (actual time=79.383..79.410 rows=15 loops=1) -> Filter: ((p.title like '%stock market%') or (p.abstract like '%stock market%') or (p.title like '%economy%') or (p.abstract like '%economy%') or (p.title like '%housing%') or (p.abstract like '%housing%')) (cost=121.15 rows=969) (actual time=0.060..60.308 rows=180 loops=1) -> Table scan on p (cost=121.15 rows=969) (actual time=0.036..2.864 rows=1099 loops=1)
```

Tried adding index:

1. `CREATE INDEX idx_abstract ON Papers(abstract(100))`

   Reason: The `abstract` column is a large text field, which is frequently queried in this analysis. This index allows MySQL to perform faster searches.

```
-> Limit: 15 row(s) (cost=121.15 rows=15) (actual time=51.299..51.328 rows=15 loops=1) -> Sort: (case when ((((case when ((p.title like '%stock market%') or (p.abstract like '%stock market%')) then 1 else 0 end) + (case when ((p.title like '%economy%') or (p.abstract like '%economy%')) then 1 else 0 end)) + (case when ((p.title like '%housing%') or (p.abstract like '%housing%')) then 1 else 0 end)) = 3) then 1 else 0 end) DESC, composite_score DESC, limit input to 15 row(s) per chunk (cost=121.15 rows=969) (actual time=51.298..51.325 rows=15 loops=1) -> Filter: ((p.title like '%stock market%') or (p.abstract like '%stock market%') or (p.title like '%economy%') or (p.abstract like '%economy%') or (p.title like '%housing%') or (p.abstract like '%housing%')) (cost=121.15 rows=969) (actual time=0.046..38.530 rows=180 loops=1) -> Table scan on p (cost=121.15 rows=969) (actual time=0.030..1.303 rows=1099 loops=1)
```

2. `CREATE INDEX idx_title ON Papers(title(100));`

   Reason: Similar to the abstract field, the title is also being searched with LIKE conditions multiple times during query.

```
-> Limit: 15 row(s) (cost=121.15 rows=15) (actual time=53.476..53.552 rows=15 loops=1) -> Sort: (case when ((((case when ((p.title like '%stock market%') or (p.abstract like '%stock market%')) then 1 else 0 end) + (case when ((p.title like '%economy%') or (p.abstract like '%economy%')) then 1 else 0 end)) + (case when ((p.title like '%housing%') or (p.abstract like '%housing%')) then 1 else 0 end)) = 3) then 1 else 0 end) DESC, composite_score DESC, limit input to 15 row(s) per chunk (cost=121.15 rows=969) (actual time=53.475..53.550 rows=15 loops=1) -> Filter: ((p.title like '%stock market%') or (p.abstract like '%stock market%') or (p.title like '%economy%') or (p.abstract like '%economy%') or (p.title like '%housing%') or (p.abstract like '%housing%')) (cost=121.15 rows=969) (actual time=0.048..39.650 rows=180 loops=1) -> Table scan on p (cost=121.15 rows=969) (actual time=0.030..1.283 rows=1099 loops=1)
```

3. `CREATE INDEX idx_citation_num ON Papers(citation_num);`

Reason: Since the query sorts results based on a calculated score that includes citation_num, indexing this column enables the database to perform the sorting operation more efficiently

```
-> Limit: 15 row(s) (cost=121.15 rows=15) (actual time=50.553..50.637 rows=15 loops=1) -> Sort: (case when ((((case when ((p.title like '%stock market%') or (p.abstract like
'%stock market%')) then 1 else 0 end) + (case when ((p.title like '%economy%') or (p.abstract like '%economy%')) then 1 else 0 end)) + (case when ((p.title like '%housing%') or
(p.abstract like '%housing%')) then 1 else 0 end)) = 3) then 1 else 0 end) DESC, composite_score DESC, limit input to 15 row(s) per chunk (cost=121.15 rows=969) (actual
time=50.551..50.634 rows=15 loops=1) -> Filter: ((p.title like '%stock market%') or (p.abstract like '%stock market%') or (p.title like '%economy%') or (p.abstract like
'%economy%') or (p.title like '%housing%') or (p.abstract like '%housing%')) (cost=121.15 rows=969) (actual time=0.091..38.348 rows=180 loops=1) -> Table scan on p
(cost=121.15 rows=969) (actual time=0.030..1.250 rows=1099 loops=1)
```

There were no actual improvements in Cost after comparing the original analyze and all three index. The reason could be that full-text pattern matching can be difficult for the optimizer to estimate correctly. Even with the index, it still has to scan a lot of data to check for matches. Also, our dataset is small so the index might not have any improvements on the original search method.

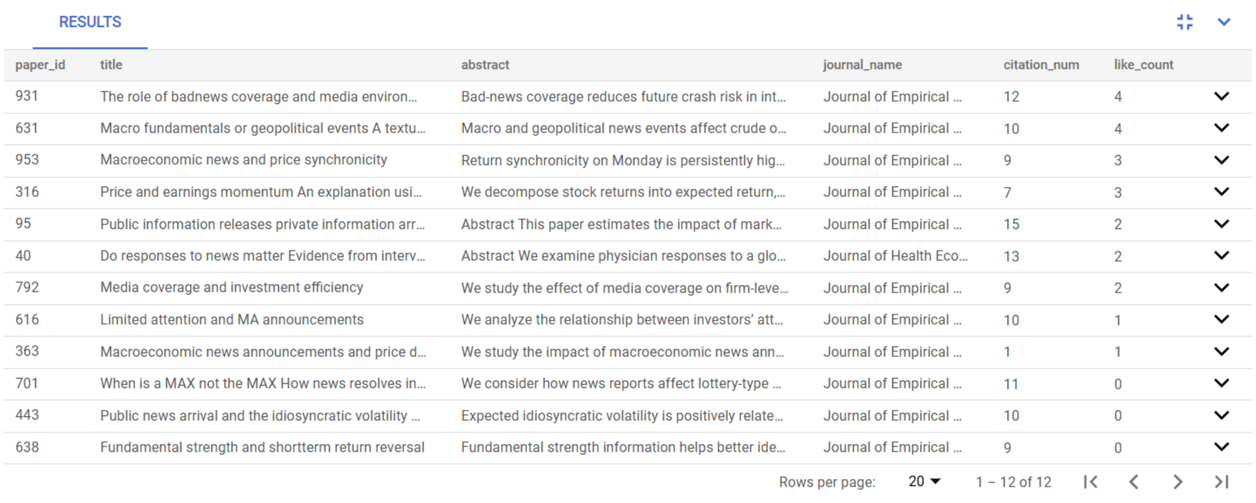
Final Index:
CREATE INDEX title_abstract_idx ON Papers(title(255), abstract(255));
Justification: This index allows the program to quickly locate rows that contain keywords in either the title or abstract, making it the most efficient way to process multiple LIKE filters on both fields once the datasets gets larger and larger. We access the abstract very frequently in this query since we use it to determine the papers that are the most relevant. As a result, it makes a lot of sense to add an index on this attribute.

3.

This query recommends a user 50 papers based on their existing likes' cluster_ids

```
SELECT
    p.paper_id,
    p.title,
    p.abstract,
    p.journal_name,
    p.citation_num,
    COUNT(l_all.user_id) AS like_count
FROM
    Papers p
JOIN
    (
        SELECT DISTINCT
            paper.cluster_id
        FROM
```

```
            Papers paper
        JOIN
            Likes l ON paper.paper_id = l.paper_id
        WHERE
            l.user_id = 1
    ) AS user_clusters ON p.cluster_id = user_clusters.cluster_id
LEFT JOIN
    Likes l_user ON p.paper_id = l_user.paper_id AND l_user.user_id = 1
LEFT JOIN
    Likes l_all ON p.paper_id = l_all.paper_id
WHERE
    l_user.paper_id IS NULL
GROUP BY
    p.paper_id,
    p.title,
    p.abstract,
    p.journal_name,
    p.citation_num
ORDER BY
    like_count DESC,
    p.citation_num DESC
LIMIT
    50;
```


RESULTS

paper_id	title	abstract	journal_name	citation_num	like_count
931	The role of badnews coverage and media environ...	Bad-news coverage reduces future crash risk in int...	Journal of Empirical ...	12	4
631	Macro fundamentals or geopolitical events A textu...	Macro and geopolitical news events affect crude o...	Journal of Empirical ...	10	4
953	Macroeconomic news and price synchronicity	Return synchronicity on Monday is persistently hig...	Journal of Empirical ...	9	3
316	Price and earnings momentum An explanation usi...	We decompose stock returns into expected return,...	Journal of Empirical ...	7	3
95	Public information releases private information arr...	Abstract This paper estimates the impact of mark...	Journal of Empirical ...	15	2
40	Do responses to news matter Evidence from interv...	Abstract We examine physician responses to a glo...	Journal of Health Eco...	13	2
792	Media coverage and investment efficiency	We study the effect of media coverage on firm-leve...	Journal of Empirical ...	9	2
616	Limited attention and MA announcements	We analyze the relationship between investors' att...	Journal of Empirical ...	10	1
363	Macroeconomic news announcements and price d...	We study the impact of macroeconomic news ann...	Journal of Empirical ...	1	1
701	When is a MAX not the MAX How news resolves in...	We consider how news reports affect lottery-type ...	Journal of Empirical ...	11	0
443	Public news arrival and the idiosyncratic volatility ...	Expected idiosyncratic volatility is positively relate...	Journal of Empirical ...	10	0
638	Fundamental strength and shortterm return reversal	Fundamental strength information helps better ide...	Journal of Empirical ...	9	0

Rows per page: 20 1 – 12 of 12

Since there are only 12 papers with similar cluster_id to liked cluster_ids we display the 12

Indexing attempts:

1. CREATE INDEX idx_papers_cluster_id ON Papers(cluster_id)
2. CREATE INDEX idx_papers_cluster_citation_num ON Papers(cluster_id, citation_num);
3. Using both of the above at the same time

1. idx_papers_cluster_id
Purpose: Intended to improve filtering speed by quickly locating papers with a matching cluster_id in the Papers table.
Result: No measurable performance gain; query remains at approximately 230.19 cost.
2. idx_papers_cluster_citation_num
Purpose: Created to enhance sorting efficiency, especially when ordering by citation_num within each cluster_id.
Result: No measurable performance gain; query remains at 230.65 cost.
3. Combined
Purpose: Combining indexes is theoretically aimed at improving performance across filtering, joining, and sorting operations simultaneously.
Result: No improvement in query time, maintaining around 230.65 cost in the combination of the two indexes.

```
Base:
-> Limit: 50 row(s) (actual time=11.872..11.927 rows=12 loops=1) -> Sort: like_count DESC, p.citation_num DESC, limit input to 50
row(s) per chunk (actual time=11.870..11.923 rows=12 loops=1) -> Table scan on <temporary> (actual time=8.443..8.450 rows=12
loops=1) -> Aggregate using temporary table (actual time=8.441..8.441 rows=12 loops=1) -> Nested loop left join (cost=30.65
rows=28) (actual time=4.991..5.219 rows=25 loops=1) -> Filter: (l_user.paper_id is null) (cost=21.43 rows=13) (actual
time=4.973..5.109 rows=12 loops=1) -> Nested loop antijoin (cost=21.43 rows=13) (actual time=4.970..5.103 rows=12 loops=1) ->
Nested loop inner join (cost=13.68 rows=13) (actual time=4.935..5.010 rows=13 loops=1) -> Table scan on user_clusters
(cost=5.93..5.93 rows=1) (actual time=4.130..4.133 rows=1 loops=1) -> Materialize (cost=3.41..3.41 rows=1) (actual
time=4.129..4.129 rows=1 loops=1) -> Table scan on <temporary> (cost=3.31..3.31 rows=1) (actual time=0.113..0.113 rows=1 loops=1)
-> Temporary table with deduplication (cost=0.80..0.80 rows=1) (actual time=0.109..0.109 rows=1 loops=1) -> Nested loop inner
join (cost=0.70 rows=1) (actual time=0.087..0.092 rows=1 loops=1) -> Covering index lookup on l using PRIMARY (user_id=1)
(cost=0.35 rows=1) (actual time=0.065..0.070 rows=1 loops=1) -> Single-row index lookup on paper using PRIMARY
(paper_id=l.paper_id) (cost=0.35 rows=1) (actual time=0.019..0.019 rows=1 loops=1) -> Index lookup on p using
idx_papers_cluster_id (cluster_id=user_clusters.cluster_id) (cost=3.88 rows=13) (actual time=0.803..0.872 rows=13 loops=1) ->
Single-row covering index lookup on l_user using PRIMARY (user_id=1, paper_id=p.paper_id) (cost=0.25 rows=1) (actual
time=0.007..0.007 rows=0 loops=13) -> Covering index lookup on l_all using paper_id (paper_id=p.paper_id) (cost=0.26 rows=2)
(actual time=0.006..0.008 rows=2 loops=12)


Config 1:
-> Limit: 50 row(s) (actual time=6.553..6.591 rows=12 loops=1) -> Sort: like_count DESC, p.citation_num DESC, limit input to 50
row(s) per chunk (actual time=6.551..6.587 rows=12 loops=1) -> Table scan on <temporary> (actual time=6.464..6.469 rows=12
loops=1) -> Aggregate using temporary table (actual time=6.462..6.462 rows=12 loops=1) -> Nested loop left join (cost=30.65
rows=28) (actual time=1.006..1.821 rows=25 loops=1) -> Filter: (l_user.paper_id is null) (cost=21.43 rows=13) (actual
time=0.991..1.479 rows=12 loops=1) -> Nested loop antijoin (cost=21.43 rows=13) (actual time=0.988..1.473 rows=12 loops=1) ->
Nested loop inner join (cost=13.68 rows=13) (actual time=0.975..1.382 rows=13 loops=1) -> Table scan on user_clusters
(cost=5.93..5.93 rows=1) (actual time=0.903..0.905 rows=1 loops=1) -> Materialize (cost=3.41..3.41 rows=1) (actual
time=0.902..0.902 rows=1 loops=1) -> Table scan on <temporary> (cost=3.31..3.31 rows=1) (actual time=0.803..0.804 rows=1 loops=1)
-> Temporary table with deduplication (cost=0.80..0.80 rows=1) (actual time=0.801..0.801 rows=1 loops=1) -> Nested loop inner
join (cost=0.70 rows=1) (actual time=0.777..0.781 rows=1 loops=1) -> Covering index lookup on l using PRIMARY (user_id=1)
```

```
(cost=0.35 rows=1) (actual time=0.755..0.759 rows=1 loops=1) -> Single-row index lookup on paper using PRIMARY
(paper_id=l.paper_id) (cost=0.35 rows=1) (actual time=0.019..0.020 rows=1 loops=1) -> Index lookup on p using
idx_papers_cluster_citation_num (cluster_id=user_clusters.cluster_id) (cost=3.88 rows=13) (actual time=0.071..0.471 rows=13
loops=1) -> Single-row covering index lookup on l_user using PRIMARY (user_id=1, paper_id=p.paper_id) (cost=0.25 rows=1) (actual
time=0.006..0.006 rows=0 loops=13) -> Covering index lookup on l_all using paper_id (paper_id=p.paper_id) (cost=0.26 rows=2)
(actual time=0.024..0.028 rows=2 loops=12)

Config 2:
-> Limit: 50 row(s) (actual time=17.266..17.303 rows=12 loops=1) -> Sort: like_count DESC, p.citation_num DESC, limit input to 50
row(s) per chunk (actual time=17.110..17.145 rows=12 loops=1) -> Table scan on <temporary> (actual time=16.986..16.991 rows=12
loops=1) -> Aggregate using temporary table (actual time=16.982..16.982 rows=12 loops=1) -> Nested loop left join (cost=230.18
rows=207) (actual time=3.281..9.046 rows=25 loops=1) -> Filter: (l_user.paper_id is null) (cost=160.99 rows=97) (actual
time=3.237..8.878 rows=12 loops=1) -> Nested loop antijoin (cost=160.99 rows=97) (actual time=3.235..8.870 rows=12 loops=1) ->
Inner hash join (p.cluster_id = user_clusters.cluster_id) (cost=127.08 rows=97) (actual time=3.209..8.748 rows=13 loops=1) ->
Table scan on p (cost=16.97 rows=969) (actual time=0.018..7.283 rows=1099 loops=1) -> Hash -> Table scan on user_clusters
(cost=5.93..5.93 rows=1) (actual time=1.171..1.171 rows=1 loops=1) -> Materialize (cost=3.41..3.41 rows=1) (actual
time=1.170..1.170 rows=1 loops=1) -> Table scan on <temporary> (cost=3.31..3.31 rows=1) (actual time=0.097..0.098 rows=1 loops=1)
-> Temporary table with deduplication (cost=0.80..0.80 rows=1) (actual time=0.095..0.095 rows=1 loops=1) -> Nested loop inner
join (cost=0.70 rows=1) (actual time=0.076..0.080 rows=1 loops=1) -> Covering index lookup on l using PRIMARY (user_id=1)
(cost=0.35 rows=1) (actual time=0.054..0.058 rows=1 loops=1) -> Single-row index lookup on paper using PRIMARY
(paper_id=l.paper_id) (cost=0.35 rows=1) (actual time=0.019..0.019 rows=1 loops=1) -> Single-row covering index lookup on l_user
using PRIMARY (user_id=1, paper_id=p.paper_id) (cost=0.13 rows=1) (actual time=0.009..0.009 rows=0 loops=13) -> Covering index
lookup on l_all using paper_id (paper_id=p.paper_id) (cost=0.25 rows=2) (actual time=0.009..0.013 rows=2 loops=12)

Config 3:
-> Limit: 50 row(s) (actual time=7.509..7.552 rows=12 loops=1) -> Sort: like_count DESC, p.citation_num DESC, limit input to 50
row(s) per chunk (actual time=7.507..7.548 rows=12 loops=1) -> Table scan on <temporary> (actual time=7.462..7.467 rows=12
loops=1) -> Aggregate using temporary table (actual time=7.460..7.460 rows=12 loops=1) -> Nested loop left join (cost=30.65
rows=28) (actual time=0.229..0.487 rows=25 loops=1) -> Filter: (l_user.paper_id is null) (cost=21.43 rows=13) (actual
time=0.215..0.372 rows=12 loops=1) -> Nested loop antijoin (cost=21.43 rows=13) (actual time=0.212..0.366 rows=12 loops=1) ->
Nested loop inner join (cost=13.68 rows=13) (actual time=0.203..0.288 rows=13 loops=1) -> Table scan on user_clusters
(cost=5.93..5.93 rows=1) (actual time=0.181..0.184 rows=1 loops=1) -> Materialize (cost=3.41..3.41 rows=1) (actual
time=0.180..0.180 rows=1 loops=1) -> Table scan on <temporary> (cost=3.31..3.31 rows=1) (actual time=0.131..0.131 rows=1 loops=1)
-> Temporary table with deduplication (cost=0.80..0.80 rows=1) (actual time=0.128..0.128 rows=1 loops=1) -> Nested loop inner
join (cost=0.70 rows=1) (actual time=0.110..0.114 rows=1 loops=1) -> Covering index lookup on l using PRIMARY (user_id=1)
(cost=0.35 rows=1) (actual time=0.092..0.095 rows=1 loops=1) -> Single-row index lookup on paper using PRIMARY
(paper_id=l.paper_id) (cost=0.35 rows=1) (actual time=0.016..0.017 rows=1 loops=1) -> Index lookup on p using
idx_papers_cluster_id (cluster_id=user_clusters.cluster_id) (cost=3.88 rows=13) (actual time=0.021..0.099 rows=13 loops=1) ->
Single-row covering index lookup on l_user using PRIMARY (user_id=1, paper_id=p.paper_id) (cost=0.25 rows=1) (actual
time=0.006..0.006 rows=0 loops=13) -> Covering index lookup on l_all using paper_id (paper_id=p.paper_id) (cost=0.26 rows=2)
(actual time=0.006..0.009 rows=2 loops=12)
```

Final design:

Index 1: idx_likes_timeliked ON Likes(time_liked)

Reason: This index will significantly enhance filtering and searching efficiency when querying Likes based on time, especially in scenarios where date range filtering is common (like in query 1).

Index 2: idx_papers_cluster_id ON Papers(cluster_id)

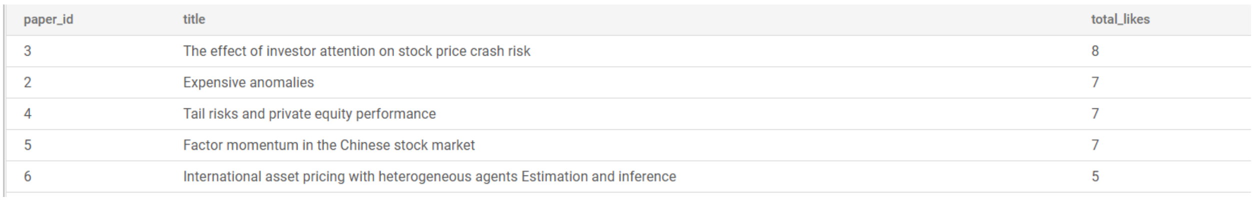
Reason: This index effectively supports efficient retrieval of papers within the same cluster_id, which is essential for recommendation queries that require quick filtering by cluster. As the Papers dataset expands, this index will help the recommendation system scale better by reducing scan times, focusing retrieval efforts directly on the relevant cluster_id subset.

Using these two indexes should help maintain scalability and support performance in filtering, joining, and time-based querying as data volume increases.

4.

This query returns the most liked papers for a given user in the last 30 days. We use user_id 7 for the purpose of this demonstration.

```
SELECT
    p.paper_id,
    p.title,
    COUNT(l_all.user_id) AS total_likes
FROM
    Likes l
JOIN
    Papers p ON l.paper_id = p.paper_id
JOIN
    Likes l_all ON p.paper_id = l_all.paper_id
WHERE
    l.user_id = 7 AND l.time_liked >= NOW() - INTERVAL 30 DAY
GROUP BY
    p.paper_id, p.title
ORDER BY
    total_likes DESC
LIMIT 15;
```

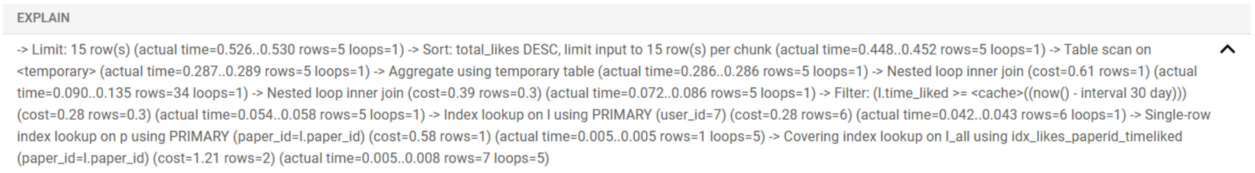


paper_id	title	total_likes
3	The effect of investor attention on stock price crash risk	8
2	Expensive anomalies	7
4	Tail risks and private equity performance	7
5	Factor momentum in the Chinese stock market	7
6	International asset pricing with heterogeneous agents Estimation and inference	5

This query only displays 5 rows because the user (user_id = 7) only liked 5 papers in our auto-generated data .

Indexing

<u>Before Indexing</u>



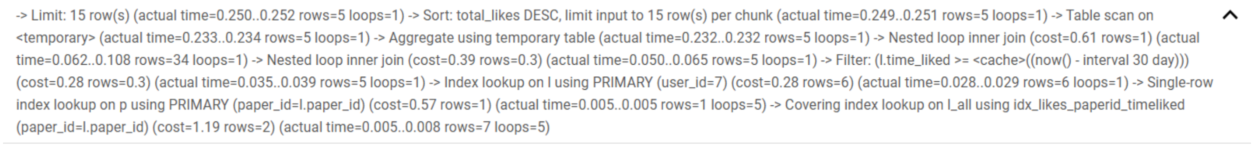
EXPLAIN

```
-> Limit: 15 row(s) (actual time=0.526..0.530 rows=5 loops=1) -> Sort: total_likes DESC, limit input to 15 row(s) per chunk (actual time=0.448..0.452 rows=5 loops=1) -> Table scan on <temporary> (actual time=0.287..0.289 rows=5 loops=1) -> Aggregate using temporary table (actual time=0.286..0.286 rows=5 loops=1) -> Nested loop inner join (cost=0.61 rows=1) (actual time=0.090..0.135 rows=34 loops=1) -> Nested loop inner join (cost=0.39 rows=0.3) (actual time=0.072..0.086 rows=5 loops=1) -> Filter: (l.time_liked >= <cache>((now() - interval 30 day))) (cost=0.28 rows=0.3) (actual time=0.054..0.058 rows=5 loops=1) -> Index lookup on l using PRIMARY (user_id=7) (cost=0.28 rows=6) (actual time=0.042..0.043 rows=6 loops=1) -> Single-row index lookup on p using PRIMARY (paper_id=l.paper_id) (cost=0.58 rows=1) (actual time=0.005..0.005 rows=1 loops=5) -> Covering index lookup on l_all using idx_likes_paperid_timeliked (paper_id=l.paper_id) (cost=1.21 rows=2) (actual time=0.005..0.008 rows=7 loops=5)
```

<u>First Indexing</u>

CREATE INDEX idx_user_time ON Likes(user_id, time_liked);

We chose this indexing because it will enable the database to quickly filter rows based on our requirements related to user_id and time_liked in the query.

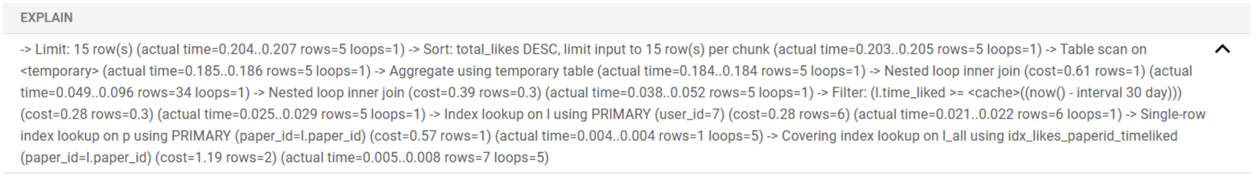


```
-> Limit: 15 row(s) (actual time=0.250..0.252 rows=5 loops=1) -> Sort: total_likes DESC, limit input to 15 row(s) per chunk (actual time=0.249..0.251 rows=5 loops=1) -> Table scan on <temporary> (actual time=0.233..0.234 rows=5 loops=1) -> Aggregate using temporary table (actual time=0.232..0.232 rows=5 loops=1) -> Nested loop inner join (cost=0.61 rows=1) (actual time=0.062..0.108 rows=34 loops=1) -> Nested loop inner join (cost=0.39 rows=0.3) (actual time=0.050..0.065 rows=5 loops=1) -> Filter: (l.time_liked >= <cache>((now() - interval 30 day))) (cost=0.28 rows=0.3) (actual time=0.035..0.039 rows=5 loops=1) -> Index lookup on l using PRIMARY (user_id=7) (cost=0.28 rows=6) (actual time=0.028..0.029 rows=6 loops=1) -> Single-row index lookup on p using PRIMARY (paper_id=l.paper_id) (cost=0.57 rows=1) (actual time=0.005..0.005 rows=1 loops=5) -> Covering index lookup on l_all using idx_likes_paperid_timeliked (paper_id=l.paper_id) (cost=1.19 rows=2) (actual time=0.005..0.008 rows=7 loops=5)
```

<u>Second Indexing</u>

CREATE INDEX idx_likes_paper_id ON Likes(paper_id);

We chose this indexing because it might optimize the JOIN between the Papers table and the Likes table.



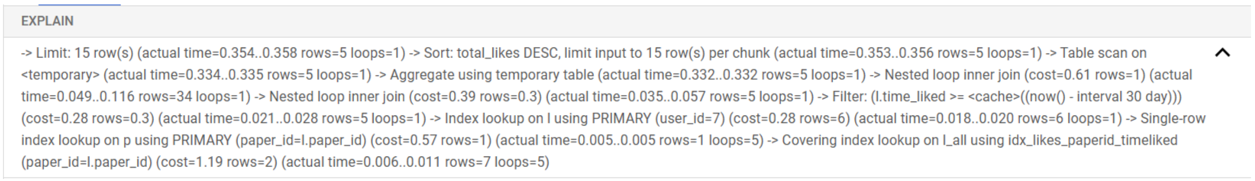
EXPLAIN

```
-> Limit: 15 row(s) (actual time=0.204..0.207 rows=5 loops=1) -> Sort: total_likes DESC, limit input to 15 row(s) per chunk (actual time=0.203..0.205 rows=5 loops=1) -> Table scan on <temporary> (actual time=0.185..0.186 rows=5 loops=1) -> Aggregate using temporary table (actual time=0.184..0.184 rows=5 loops=1) -> Nested loop inner join (cost=0.61 rows=1) (actual time=0.049..0.096 rows=34 loops=1) -> Nested loop inner join (cost=0.39 rows=0.3) (actual time=0.038..0.052 rows=5 loops=1) -> Filter: (l.time_liked >= <cache>((now() - interval 30 day))) (cost=0.28 rows=0.3) (actual time=0.025..0.029 rows=5 loops=1) -> Index lookup on l using PRIMARY (user_id=7) (cost=0.28 rows=6) (actual time=0.021..0.022 rows=6 loops=1) -> Single-row index lookup on p using PRIMARY (paper_id=l.paper_id) (cost=0.57 rows=1) (actual time=0.004..0.004 rows=1 loops=5) -> Covering index lookup on l_all using idx_likes_paperid_timeliked (paper_id=l.paper_id) (cost=1.19 rows=2) (actual time=0.005..0.008 rows=7 loops=5)
```

<u>Third Indexing</u>

CREATE INDEX idx_papers_title ON Papers(title(255));

In the query, we also tried to retrieve the title of the paper, so we want to check if indexing on the title attribute from the Papers table will help.



EXPLAIN

```
-> Limit: 15 row(s) (actual time=0.354..0.358 rows=5 loops=1) -> Sort: total_likes DESC, limit input to 15 row(s) per chunk (actual time=0.353..0.356 rows=5 loops=1) -> Table scan on <temporary> (actual time=0.334..0.335 rows=5 loops=1) -> Aggregate using temporary table (actual time=0.332..0.332 rows=5 loops=1) -> Nested loop inner join (cost=0.61 rows=1) (actual time=0.049..0.116 rows=34 loops=1) -> Nested loop inner join (cost=0.39 rows=0.3) (actual time=0.035..0.057 rows=5 loops=1) -> Filter: (l.time_liked >= <cache>((now() - interval 30 day))) (cost=0.28 rows=0.3) (actual time=0.021..0.028 rows=5 loops=1) -> Index lookup on l using PRIMARY (user_id=7) (cost=0.28 rows=6) (actual time=0.018..0.020 rows=6 loops=1) -> Single-row index lookup on p using PRIMARY (paper_id=l.paper_id) (cost=0.57 rows=1) (actual time=0.005..0.005 rows=1 loops=5) -> Covering index lookup on l_all using idx_likes_paperid_timeliked (paper_id=l.paper_id) (cost=1.19 rows=2) (actual time=0.006..0.011 rows=7 loops=5)
```

<u>Final Index Design</u>

While my three different indexing choices did improve the cost compared to the design without indexing, the cost for the three designs is generally the same. Therefore, I would prefer more on the first indexing design, where we created the composite index for Likes(user_id, time_liked). This index will further optimize the where clause in the query in cases where the database is large. This is because we use both user_id and time_liked in the where clause, and will frequently access these attributes. As a result, adding an index for these attributes is a good idea.