



Database Implementation On GCP



Connection Name

cs411

Host:

localhost

Socket:

/mysql/mysql.sock

Port:

3306

Version:

8.0.31-google ((Google))

Compiled For:

Linux (x86_64)

Configuration File:

unknown

Running Since:

Wed Jul 17 13:32:35 2024 (0:09)

Refresh

Available Server Features

Performance Schema:	<input type="radio"/> Off
Thread Pool:	<input type="radio"/> n/a
Memcached Plugin:	<input type="radio"/> n/a
Semisync Replication Plugin:	<input type="radio"/> n/a
SSL Availability:	<input checked="" type="radio"/> On

Server Directories

Base Directory:	/usr/
Data Directory:	/mysql/datadir/
Disk Space in Data Dir:	unable to retrieve
Plugins Directory:	/usr/lib/mysql/plugin/
Tmp Directory:	/mysql/tmp
Error Log:	<input checked="" type="radio"/> On stderr
General Log:	<input type="radio"/> Off
Slow Query Log:	<input type="radio"/> Off

This screenshot shows the connection between MySQL Workbench and GCP. MySQL Workbench is used to access our database stored in GCP.

DDL Commands

```
CREATE TABLE UserInfo (userId INT PRIMARY KEY AUTO_INCREMENT,  
    username VARCHAR(32) UNIQUE,  
    password VARCHAR(32)  
);
```

```
CREATE TABLE Arxiv (paperId INT PRIMARY KEY AUTO_INCREMENT,  
    arxivId VARCHAR(20),  
    submitter VARCHAR(100),  
    title VARCHAR(500),  
    comments VARCHAR(1500),  
    doi VARCHAR (250),  
    views INT,  
);
```

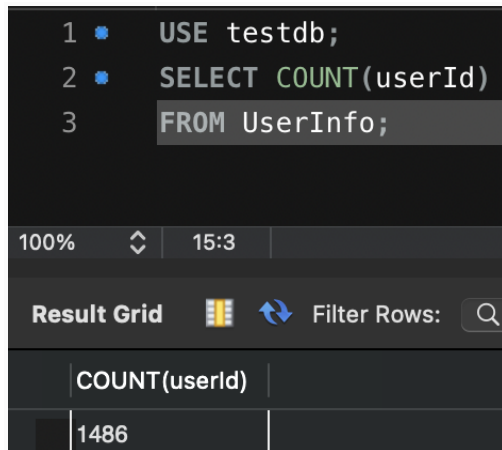
```
CREATE TABLE Notes (noteId INT PRIMARY KEY AUTO_INCREMENT,  
    paperId INT,  
    userId INT,  
    noteContent VARCHAR(1000),  
    FOREIGN KEY (paperId) REFERENCES Arxiv(paperId),  
    FOREIGN KEY (userId) REFERENCES UserInfo(userId),  
    ON DELETE CASCADE  
);
```

```
CREATE TABLE Comments (commentId INT PRIMARY KEY AUTO_INCREMENT,  
    userId INT,  
    paperId INT,  
    commentContent VARCHAR(600),  
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
    FOREIGN KEY (userId) REFERENCES UserInfo(userId),  
    FOREIGN KEY (paperId) REFERENCES Arxiv(paperId)  
    ON DELETE CASCADE  
);
```

```
CREATE TABLE Actions (actionId INT PRIMARY KEY AUTO_INCREMENT,  
    userId INT,  
    paperId INT,  
    savedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
    actionType VARCHAR(10),  
    FOREIGN KEY (paperId) REFERENCES Arxiv(paperId),  
    FOREIGN KEY (userId) REFERENCES UserInfo(userId),  
    ON DELETE CASCADE  
);
```

Table Entries

UserInfo (script)



The screenshot shows a SQL query editor with three lines of code: `USE testdb;`, `SELECT COUNT(userId)`, and `FROM UserInfo;`. Below the editor, the result grid displays a single row with the value 1486 under the column header COUNT(userId).

```
1 • USE testdb;  
2 • SELECT COUNT(userId)  
3   FROM UserInfo;
```

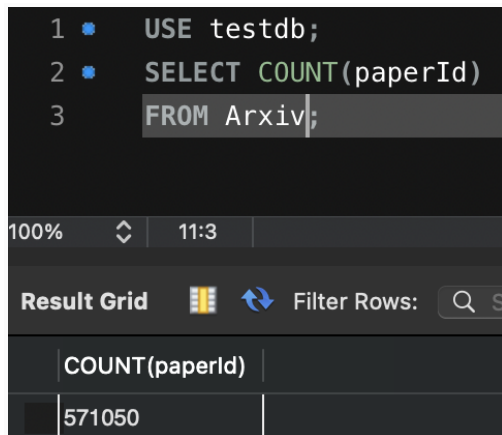
100% 15:3

Result Grid Filter Rows:

COUNT(userId)
1486

Count = 1486 entries (≥ 1000)

Arxiv (real data)



The screenshot shows a SQL query editor with three lines of code: `USE testdb;`, `SELECT COUNT(paperId)`, and `FROM Arxiv;`. Below the editor, the result grid displays a single row with the value 571050 under the column header COUNT(paperId).

```
1 • USE testdb;  
2 • SELECT COUNT(paperId)  
3   FROM Arxiv;
```

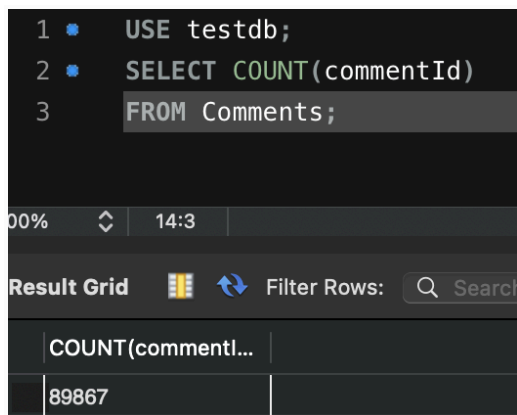
100% 11:3

Result Grid Filter Rows:

COUNT(paperId)
571050

Count = 571050 entries (≥ 1000)

Comments (script)



The screenshot shows a SQL query editor with three lines of code: `USE testdb;`, `SELECT COUNT(commentId)`, and `FROM Comments;`. Below the editor, the result grid displays a single row with the value 89867 under the column header COUNT(commentId).

```
1 • USE testdb;  
2 • SELECT COUNT(commentId)  
3   FROM Comments;
```

100% 14:3

Result Grid Filter Rows:

COUNT(commentId)
89867

Count = 89867 entries (≥ 1000)

Actions (script)

```
1 • USE testdb;  
2 • SELECT COUNT(*)  
3   FROM Actions;
```

Result Grid		Filter Rows:
	COUNT(*)	
▶	676184	

Count = 676184 entries (≥ 1000)

Notes (script)

```
1 • USE testdb;  
2 • SELECT COUNT(*)  
3   FROM Notes;
```

Result Grid		Filter Rows:
	COUNT(*)	
▶	22541	

Count = 22541 entries (≥ 1000)

Queries

Query 1

Find all saved papers where a specific user was the top commenter on the paper. Includes if they tied for most comments with another user.

```
SELECT DISTINCT a.paperId, u.username, ar.title, comment_count
FROM Actions a JOIN Comments c ON a.userId = c.userId AND a.paperId =
c.paperId JOIN Arxiv ar on a.paperId = ar.paperId JOIN UserInfo u on
a.userId = u.userId, (SELECT(COUNT(c.userId)) as comment_count
FROM Comments c NATURAL JOIN UserInfo u
WHERE u.username = [username]
GROUP BY c.paperId) as comment_count_tbl
WHERE a.paperId in (SELECT c2.paperId
FROM Comments c2
WHERE (SELECT(COUNT(c3.userId))
FROM Comments c3
WHERE c2.paperId = c3.paperId
AND c3.userId = c2.userId)
>=
ALL(SELECT(COUNT(c3.userId))
FROM Comments c3
WHERE c2.paperId = c3.paperId
AND c2.userId = c.userId
)
AND u.username = [username]
Order By paperId
```

Top 15 Rows:

paperId	username	title	comment_cou...
6	miguel29	Probing eccentric higher order modes through e...	1
11861	miguel29	In vivo impact on rabbit subchondral bone of vis...	1
17019	miguel29	SurgPLAN: Surgical Phase Localization Networ...	1
36884	miguel29	Distributed Localization in Dynamic Networks vi...	1
43814	miguel29	Convergence Analysis of a Spectral Numerical...	1
49327	miguel29	Optimal Stirring Strategies for Passive Scalars i...	1
59618	miguel29	Motion-I2V: Consistent and Controllable Image-t...	1
64371	miguel29	Subsidence and current strain patterns on Tener...	1
65898	miguel29	An overview of existing and new nuclear and as...	1
66338	miguel29	An inevitably aging world -- Analysis on the evol...	1
74342	miguel29	Multi-objective Binary Coordinate Search for Fe...	1
75612	miguel29	Multigap superconductivity in lithium intercalate...	1
80166	miguel29	Brick Wall Quantum Circuits with Global Fermio...	1
103773	miguel29	Selected Open Problems in Continuous-Time Q...	1
106735	miguel29	Fair Lotteries for Participatory Budgeting	1

Note - all users initially have 1 comment on each paper they commented on due to the script we used to populate user actions and info.

Query 2

Get all comments and notes of a user with a keyword.

```
SELECT userId, content, contentType
FROM (
    SELECT userId, commentContent AS content, 'comment' AS contentType
    FROM Comments
    WHERE commentContent LIKE '%[keyword]%'

    UNION ALL

    Select userId, noteContent AS content, 'note' AS contentType
    FROM Notes
    WHERE noteContent LIKE '%[keyword]%'
) AS subq
WHERE userId = [userid]
ORDER BY contentType DESC;
```

Top Rows:

	userId	content	contentType
▶	2	Break would explain worry. Heavy major food second rock would suffer. Give seem evidence.	note
	2	Any work trouble rule friend find bed parent such film real season art which skill.	comment

Note - Set commentContent keyword to '%bed%' and noteContent keyword to '%food%'. Set userId to 2. Only returns two values instead of >=15 because user with userId = 2 has not posted more than one comment with 'bed' in it or written more than one note with 'food' in it.

Query 3

Top Engaged Papers: Papers that were saved were also left comments OR notes on 90% of the time.

```
SELECT activity.paperId, ax.title, ratio
FROM (
    SELECT
        a.paperId,
        COUNT(DISTINCT CASE WHEN c.userId IS NOT NULL OR n.userId IS
NOT NULL
                                THEN a.userId END) / COUNT(DISTINCT a.userId) AS ratio
    FROM Actions a
    LEFT JOIN Comments c ON a.paperId = c.paperId AND a.userId = c.userId
    LEFT JOIN Notes n ON a.paperId = n.paperId AND a.userId = n.userId
    WHERE a.actionType = 'save'
    GROUP BY a.paperId
) AS activity JOIN Arxiv ax ON activity.paperId = ax.paperId
```

```
WHERE ratio >= 0.9
ORDER BY ratio DESC, paperId DESC
LIMIT 15;
```

Top 15 Rows:

	paperId	title	ratio
▶	571045	The Fluctuation Induced Pseudogap in the Infrar...	1.0000
	571017	S-35 Beta Irradiation of a Tin Strip in a State of ...	1.0000
	570972	Dynamical Symmetry Approach to Periodic Hamilt...	1.0000
	570942	On a Schwarzian PDE associated with the KdV Hi...	1.0000
	570918	Discrete Z^a and Painleve equations	1.0000
	570895	Integrable Systems in the Infinite Genus Limit	1.0000
	570889	A New Class of Optical Solitons	1.0000
	570888	On the equivalence of the discrete nonlinear Sch...	1.0000
	570881	Paraconformal Structures and Integrable Systems	1.0000
	570849	Solitons in a 3d integrable model	1.0000
	570827	Singularity Structure Analysis, Integrability, Solit...	1.0000
	570770	Coverings and integrability of the Gauss-Mainar...	1.0000
	570755	Darboux Transformations and solutions for an e...	1.0000
	570723	Lax Pairs for Integrable Lattice Systems	1.0000
	570704	On fusion algebra of chiral $SU(N)_k$ models	1.0000

Query 4

Top 15 papers that have been saved plus viewed the most times. Also finds cases where no saves are present but have views. For tie-breakers, the lower paperId value comes first.

```
SELECT ax.paperId, (COALESCE(subq.saveCount, 0) + ax.views) AS total
FROM Arxiv ax LEFT JOIN (SELECT paperId, COUNT(*) AS saveCount
                        FROM Actions
                        WHERE actionType = 'save'
                        GROUP BY paperId
                        ) AS subq
ON ax.paperId = subq.paperId
ORDER BY total DESC, ax.paperId
LIMIT 15;
```

Top 15 Rows:

paperId	total
356862	20
521156	20
29769	18
358236	17
452616	17
33510	16
199226	16
199930	16
278445	16
291555	16
301669	16
337865	16
551668	16
19420	15
39338	15

Indexing Analysis

Query 1

Default Index: COST: 86.74

```
1  --> Limit: 15 row(s) (actual time=7.650..7.652 rows=15 loops=1)
2      --> Sort: a.paperId, limit input to 15 row(s) per chunk (actual time=7.649..7.651 rows=15 loops=1)
3          --> Table scan on <temporary> (cost=86.74..86.74 rows=0) (actual time=7.616..7.624 rows=61 loops=1)
4              --> Temporary table with deduplication (cost=84.24..84.24 rows=0) (actual time=7.614..7.614 rows=61 loops=1)
5                  --> Inner hash join (no condition) (cost=84.24 rows=0) (actual time=2.647..3.717 rows=6624 loops=1)
6                      --> Table scan on comment_count_tbl (cost=2.50..2.50 rows=0) (actual time=0.134..0.143 rows=72 loops=1)
7                          --> Materialize (cost=0.00..0.00 rows=0) (actual time=0.133..0.133 rows=72 loops=1)
8                              --> Table scan on <temporary> (actual time=0.110..0.117 rows=72 loops=1)
9                                  --> Aggregate using temporary table (actual time=0.109..0.109 rows=72 loops=1)
10                                      --> Index lookup on Comments_ibfk_1 (userId='108') (cost=25.20 rows=72) (actual time=0.075..0.083 rows=72 loops=1)
11                                          --> Hash
12                                              --> Nested loop inner join (cost=80.74 rows=4) (actual time=0.165..2.473 rows=92 loops=1)
13                                                  --> Nested loop semijoin (cost=79.44 rows=4) (actual time=0.156..2.247 rows=92 loops=1)
14                                                      --> Nested loop inner join (cost=78.08 rows=4) (actual time=0.123..0.779 rows=107 loops=1)
15                                                          --> Filter: (c.paperId is not null) (cost=25.20 rows=72) (actual time=0.103..0.121 rows=72 loops=1)
16                                                              --> Index lookup on c using Comments_ibfk_1 (userId='108') (cost=25.20 rows=72) (actual time=0.102..0.112 rows=72 loops=1)
17                                                                  --> Filter: (a.userId = '108') (cost=0.52 rows=0.05) (actual time=0.007..0.009 rows=1 loops=72)
18                                                                      --> Index lookup on a using Actions_ibfk_1 (paperId=c.paperId) (cost=0.52 rows=2) (actual time=0.007..0.008 rows=3 loops=72)
19                                                                          --> Filter: <not>(<in_optimizer>((select #4),<exists>(select #5))) (cost=0.32 rows=1) (actual time=0.014..0.014 rows=1 loops=107)
20                                                                              --> Index lookup on c2 using Comments_ibfk_2 (paperId=c.paperId) (cost=0.32 rows=1) (actual time=0.004..0.004 rows=1 loops=107)
21                                                                                  --> Select #4 (subquery in condition; dependent)
22                                                                                      --> Aggregate: count(c3.userId) (cost=0.29 rows=1) (actual time=0.004..0.004 rows=1 loops=123)
23                                                                                          --> Filter: (c3.userId = c2.userId) (cost=0.28 rows=0.1) (actual time=0.003..0.003 rows=1 loops=123)
24                                                                                              --> Index lookup on c3 using Comments_ibfk_2 (paperId=c2.paperId) (cost=0.28 rows=1) (actual time=0.003..0.003 rows=1 loops=123)
25                                                                                                  --> Select #4 (subquery in condition; dependent)
26                                                                                                      --> Aggregate: count(c3.userId) (cost=0.29 rows=1) (actual time=0.004..0.004 rows=1 loops=123)
27                                                                                                          --> Filter: (c3.userId = c2.userId) (cost=0.28 rows=0.1) (actual time=0.003..0.003 rows=1 loops=123)
28                                                                                                              --> Index lookup on c3 using Comments_ibfk_2 (paperId=c2.paperId) (cost=0.28 rows=1) (actual time=0.003..0.003 rows=1 loops=123)
29                                                                                                                  --> Select #5 (subquery in condition; dependent)
30                                                                                                                      --> Limit: 1 row(s) (cost=0.49 rows=1) (actual time=0.004..0.004 rows=0 loops=123)
31                                                                                                                          --> Filter: <if>(outer_field_is_not_null, (<cache>((select #4)) <<ref_null_helper>(count(c3.userId))), true) (cost=0.49 rows=1) (actual time=0.003..0.003 rows=0 loops=123)
32                                                                                                                              --> Aggregate: count(c3.userId) (cost=0.49 rows=1) (actual time=0.003..0.003 rows=1 loops=123)
33                                                                                                                                  --> Index lookup on c3 using Comments_ibfk_2 (paperId=c2.paperId) (cost=0.38 rows=1) (actual time=0.002..0.003 rows=1 loops=123)
34                                                                                                                                      --> Select #4 (subquery in condition; dependent)
35                                                                                                                                          --> Aggregate: count(c3.userId) (cost=0.29 rows=1) (actual time=0.004..0.004 rows=1 loops=123)
36                                                                                                                                              --> Filter: (c3.userId = c2.userId) (cost=0.28 rows=0.1) (actual time=0.003..0.003 rows=1 loops=123)
37                                                                                                                                      --> Index lookup on c3 using Comments_ibfk_2 (paperId=c2.paperId) (cost=0.28 rows=1) (actual time=0.003..0.003 rows=1 loops=123)
38 --> Single-row index lookup on ar using PRIMARY (paperId=c.paperId) (cost=0.28 rows=1) (actual time=0.002..0.002 rows=1 loops=92)
```

Index Design 1: COST: 71.51

CREATE INDEX username_idx on UserInfo(username); (forced index usage)

```
1  --> Limit: 15 row(s) (actual time=0.820..0.822 rows=15 loops=1)
2      --> Sort: a.paperId, limit input to 15 row(s) per chunk (actual time=0.819..0.820 rows=15 loops=1)
3          --> Table scan on <temporary> (cost=71.51..71.51 rows=0) (actual time=0.783..0.792 rows=61 loops=1)
4              --> Temporary table with deduplication (cost=69.01..69.01 rows=0) (actual time=0.782..0.782 rows=61 loops=1)
5                  --> Inner hash join (no condition) (cost=69.01 rows=0) (actual time=2.638..3.837 rows=6624 loops=1)
6                      --> Table scan on comment_count_tbl (cost=2.50..2.50 rows=0) (actual time=0.182..0.192 rows=72 loops=1)
7                          --> Materialize (cost=0.00..0.00 rows=0) (actual time=0.182..0.182 rows=72 loops=1)
8                              --> Table scan on <temporary> (actual time=0.157..0.164 rows=72 loops=1)
9                                  --> Aggregate using temporary table (actual time=0.156..0.156 rows=72 loops=1)
10                                      --> Index lookup on Comments_ibfk_1 (userId='108') (cost=25.20 rows=72) (actual time=0.128..0.129 rows=72 loops=1)
11                                          --> Hash
12                                              --> Nested loop inner join (cost=65.53 rows=3) (actual time=0.144..2.412 rows=92 loops=1)
13                                                  --> Nested loop semijoin (cost=64.49 rows=3) (actual time=0.135..2.194 rows=92 loops=1)
14                                                      --> Nested loop inner join (cost=63.40 rows=3) (actual time=0.107..0.764 rows=107 loops=1)
15                                                          --> Nested loop inner join (cost=20.95 rows=58) (actual time=0.091..0.133 rows=72 loops=1)
16                                                              --> Covering index lookup on u using username_idx (username='migue129') (cost=0.72 rows=1) (actual time=0.012..0.017 rows=1 loops=1)
17                                                                  --> Filter: (c.paperId is not null) (cost=20.23 rows=58) (actual time=0.079..0.111 rows=72 loops=1)
18                                                                      --> Index lookup on c using Comments_ibfk_1 (userId=u.userId) (cost=20.23 rows=58) (actual time=0.078..0.102 rows=72 loops=1)
19                                                                          --> Filter: (a.userId = u.userId) (cost=0.52 rows=0.05) (actual time=0.007..0.009 rows=1 loops=72)
20                                                                              --> Index lookup on a using Actions_ibfk_1 (paperId=c.paperId) (cost=0.52 rows=2) (actual time=0.007..0.008 rows=3 loops=72)
21                                                                                  --> Filter: <not>(<in_optimizer>((select #4),<exists>(select #5))) (cost=0.33 rows=1) (actual time=0.013..0.013 rows=1 loops=107)
22                                                                                          --> Index lookup on c2 using Comments_ibfk_2 (paperId=c.paperId) (cost=0.33 rows=1) (actual time=0.003..0.003 rows=1 loops=107)
23                                                                                              --> Select #4 (subquery in condition; dependent)
24                                                                                                  --> Aggregate: count(c3.userId) (cost=0.29 rows=1) (actual time=0.004..0.004 rows=1 loops=123)
25                                                                                                      --> Filter: (c3.userId = c2.userId) (cost=0.28 rows=0.1) (actual time=0.003..0.003 rows=1 loops=123)
26                                                                                                          --> Index lookup on c3 using Comments_ibfk_2 (paperId=c2.paperId) (cost=0.28 rows=1) (actual time=0.002..0.003 rows=1 loops=123)
27                                                                                                              --> Select #4 (subquery in condition; dependent)
28                                                                                                                  --> Aggregate: count(c3.userId) (cost=0.29 rows=1) (actual time=0.004..0.004 rows=1 loops=123)
29                                                                                                                      --> Filter: (c3.userId = c2.userId) (cost=0.28 rows=0.1) (actual time=0.003..0.003 rows=1 loops=123)
30                                                                                                                          --> Index lookup on c3 using Comments_ibfk_2 (paperId=c2.paperId) (cost=0.28 rows=1) (actual time=0.002..0.003 rows=1 loops=123)
31                                                                                                                              --> Select #5 (subquery in condition; dependent)
32                                                                                                                                  --> Limit: 1 row(s) (cost=0.49 rows=1) (actual time=0.004..0.004 rows=0 loops=123)
33                                                                                                                                      --> Filter: <if>(outer_field_is_not_null, (<cache>(select #4)) <<ref_null_helper>(count(c3.userId))), true) (cost=0.49 rows=1) (actual time=0.004..0.004 rows=0 loops=123)
34                                                                                                                                          --> Aggregate: count(c3.userId) (cost=0.49 rows=1) (actual time=0.003..0.003 rows=1 loops=123)
35                                                                                                                                              --> Index lookup on c3 using Comments_ibfk_2 (paperId=c2.paperId) (cost=0.38 rows=1) (actual time=0.003..0.003 rows=1 loops=123)
36                                                                                                                                      --> Select #4 (subquery in condition; dependent)
37                                                                                                                                          --> Aggregate: count(c3.userId) (cost=0.29 rows=1) (actual time=0.004..0.004 rows=1 loops=123)
38                                                                                                                                              --> Filter: (c3.userId = c2.userId) (cost=0.28 rows=0.1) (actual time=0.003..0.003 rows=1 loops=123)
39                                                                                                                                      --> Index lookup on c3 using Comments_ibfk_2 (paperId=c2.paperId) (cost=0.28 rows=1) (actual time=0.002..0.003 rows=1 loops=123)
40 --> Single-row index lookup on ar using PRIMARY (paperId=c.paperId) (cost=0.28 rows=1) (actual time=0.002..0.003 rows=1 loops=92)
```

Analysis:

No other indices were possible for this query, because no other attributes were used other than default indices for primary keys ('userId'/'paperId') and 'username'. While inspecting the "EXPLAIN ANALYZE" result, we initially saw that 'username_idx' was not used, possibly because the smaller number of users made it more efficient to just look through the existing table rather than taking the index. However, when we force the query to use the index, the cost did drop. The reason the cost is so low compared to other queries is that we are looking for 1 specific username – querying this for all usernames results in a cost of 79916.

Query 2

Default Index

Result:

```
1  -> Sort: subq.contentType DESC (cost=24.62..24.62 rows=5) (actual time=0.188..0.189 rows=2 loops=1)
2  -> Table scan on subq (cost=20.79..22.86 rows=5) (actual time=0.177..0.177 rows=2 loops=1)
3  -> Union all materialize (cost=20.29..20.29 rows=5) (actual time=0.175..0.175 rows=2 loops=1)
4  -> Filter: (Comments.commentContent like '%bed%') (cost=9.66 rows=4) (actual time=0.084..0.113
    rows=1 loops=1)
5  -> Index lookup on Comments using Comments_ibfk_1 (userId=2) (cost=9.66 rows=37) (actual
    time=0.075..0.080 rows=37 loops=1)
6  -> Filter: (Notes.noteContent like '%food%') (cost=10.11 rows=1) (actual time=0.044..0.046 rows=1
    loops=1)
7  -> Index lookup on Notes using Notes_ibfk_2 (userId=2) (cost=10.11 rows=10) (actual
    time=0.023..0.024 rows=10 loops=1)
```

Total Cost: 24.62

Index Design 1

Added fulltext index for commentContent. Also changed line in query from

```
WHERE commentContent LIKE '%bed%'
```

to

```
WHERE MATCH(commentContent) AGAINST ('bed')
```

Result:

```
1  -> Sort: subq.contentType DESC (cost=13.98..13.98 rows=1) (actual time=8.845..8.845 rows=2 loops=1)
2  -> Table scan on subq (cost=13.40..13.74 rows=1) (actual time=8.830..8.830 rows=2 loops=1)
3  -> Union all materialize (cost=11.23..11.23 rows=1) (actual time=8.828..8.828 rows=2 loops=1)
4  -> Filter: ((Comments.userId = 2) and (match Comments.commentContent against ('bed'))) (cost=1.01
    rows=0.05) (actual time=0.341..8.738 rows=1 loops=1)
5  -> Full-text index search on Comments using commentContent_index (commentContent='bed')
    (cost=1.01 rows=1) (actual time=0.265..8.614 rows=1054 loops=1)
6  -> Filter: (Notes.noteContent like '%food%') (cost=10.11 rows=1) (actual time=0.057..0.059 rows=1
    loops=1)
7  -> Index lookup on Notes using Notes_ibfk_2 (userId=2) (cost=10.11 rows=10) (actual
    time=0.036..0.040 rows=10 loops=1)
```

Total Cost: 13.98

Analysis: This has shown a drastic decrease in query costs, from 24.62 to 13.98. With the use of a fulltext index, particularly on text-heavy attributes, the database is able to perform efficient index scans on text data. This dramatically reduces the time required for searches and lowers resource consumption. This was the case for finding the keyword 'bed' within comments. In order to get it working, we had to change the query to using MATCH and AGAINST instead of LIKE.

Index Design 2

Added fulltext index for noteContent. Also changed line from

```
WHERE noteContent LIKE '%food%'
```

to

```
WHERE MATCH(noteContent) AGAINST ('food')
```

Result:

```
1  -> Sort: subq.contentType DESC (cost=14.90..14.90 rows=4) (actual time=1.474..1.474 rows=2 loops=1)
2  -> Table scan on subq (cost=11.70..13.63 rows=4) (actual time=1.458..1.459 rows=2 loops=1)
3  -> Union all materialize (cost=11.08..11.08 rows=4) (actual time=1.455..1.455 rows=2 loops=1)
4  -> Filter: (Comments.commentContent like '%bed%') (cost=9.66 rows=4) (actual time=0.130..0.160
    rows=1 loops=1)
5  -> Index lookup on Comments using Comments_ibfk_1 (userId=2) (cost=9.66 rows=37) (actual
    time=0.120..0.127 rows=37 loops=1)
6  -> Filter: ((Notes.userId = 2) and (match Notes.noteContent against ('food'))) (cost=1.01 rows=0.05)
    (actual time=0.113..1.278 rows=1 loops=1)
7  -> Full-text index search on Notes using noteContent_index (noteContent='food') (cost=1.01 rows=1)
    (actual time=0.092..1.237 rows=453 loops=1)
```

Total Cost: 14.90

Analysis: Now, we wanted to see if we can get the same result for using a fulltext index on note contents. Similarly, we got the same result and a heavy performance gain was shown, with the cost going from 24.62 to 14.90. To get this working for notes, we had to change the query to using MATCH and AGAINST instead of LIKE.

Index Design 3 **FINAL DESIGN**

Added index for commentContent and noteContent.

Final Query Code:

```
SELECT userId, content, contentType
FROM (
    SELECT userId, commentContent AS content, 'comment' AS contentType
    FROM Comments
    WHERE MATCH(commentContent) AGAINST ('bed')

    UNION ALL

    Select userId, noteContent AS content, 'note' AS contentType
    FROM Notes
    WHERE MATCH(noteContent) AGAINST ('food')
) AS subq
WHERE userId = 2
ORDER BY contentType DESC;
```

Result:

```

1  -> Sort: subq.contentType DESC (cost=4.62..4.62 rows=0.1) (actual time=6.215..6.216 rows=2 loops=1)
2    -> Table scan on subq (cost=4.52..4.52 rows=0.1) (actual time=6.194..6.194 rows=2 loops=1)
3      -> Union all materialize (cost=2.02..2.02 rows=0.1) (actual time=6.190..6.190 rows=2 loops=1)
4        -> Filter: ((Comments.userId = 2) and (match Comments.commentContent against ('bed'))) (cost=1.01
rows=0.05) (actual time=0.319..4.911 rows=1 loops=1)
5          -> Full-text index search on Comments using commentContent_index (commentContent='bed')
(cost=1.01 rows=1) (actual time=0.253..4.814 rows=1054 loops=1)
6            -> Filter: ((Notes.userId = 2) and (match Notes.noteContent against ('food'))) (cost=1.01 rows=0.05)
(actual time=0.122..1.253 rows=1 loops=1)
7              -> Full-text index search on Notes using noteContent_index (noteContent='food') (cost=1.01 rows=1)
(actual time=0.105..1.216 rows=453 loops=1)

```

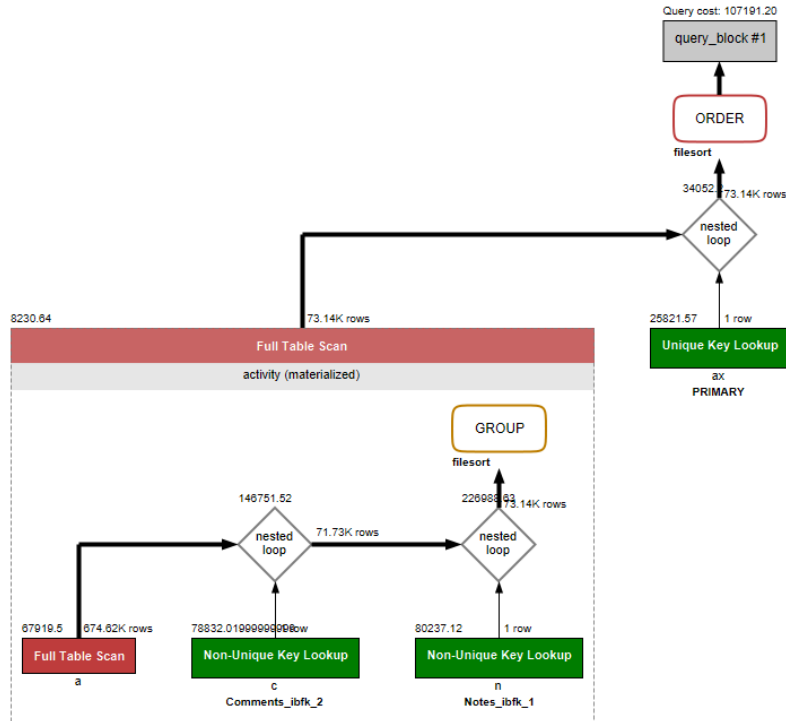
Total Cost: 4.62

Analysis: Since the previous two index designs have shown great promise in reducing the cost of the query, we have decided to combine them. By implementing a fulltext index for both tables, we were able to reduce the cost by more than 80%, from 24.62 to 4.62. This will be our final index design for this query.

Query 3

Without indexing: cost=107191.2

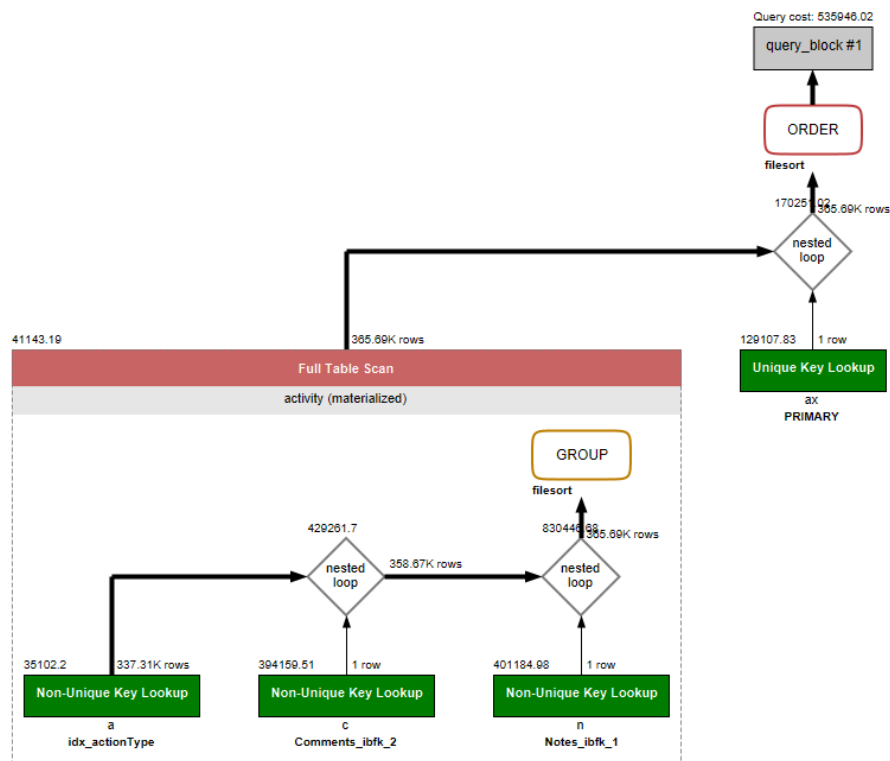
Index on Actions.actionType: cost=535946.02



```

1  -> Nested loop inner join (cost=2175494.71 rows=731392) (actual time=1523.638..1527.561 rows=1248 loops=1)
2    -> Sort: activity.ratio DESC, activity.paperId DESC (cost=2083847.88..2083847.88 rows=731392) (actual time=1523.620..1523.742 rows=1248 loops=1)
3    -> Filter: (activity.paperId is not null) (cost=503653.14..585937.08 rows=731392) (actual time=1522.614..1522.826 rows=1248 loops=1)
4    -> Table scan on activity (cost=503653.04..512797.92 rows=731392) (actual time=1522.613..1522.742 rows=1248 loops=1)
5    -> Materialize (cost=503653.03..503653.03 rows=731392) (actual time=1522.610..1522.610 rows=1248 loops=1)
6    -> Filter: (((count(distinct c.userId) + count(distinct n.userId)) / count(distinct a.userId)) / 2) >= 0.9) (cost=430513.87 rows=731392) (actual time=397.128..1521.095 rows=1248 loops=1)
7    -> Group aggregate: count(distinct a.userId), count(distinct n.userId), count(distinct c.userId) (cost=430513.87 rows=731392) (actual time=395.193..1451.336 rows=154577 loops=1)
8    -> Nested loop left join (cost=357374.71 rows=731392) (actual time=395.172..1355.442 rows=180089 loops=1)
9    -> Nested loop left join (cost=211312.35 rows=717343) (actual time=395.157..956.670 rows=180089 loops=1)
10   -> Sort: a.paperId (cost=67919.50 rows=674625) (actual time=395.108..414.207 rows=180084 loops=1)
11   -> Filter: (a.actionType = 'save') (cost=67919.50 rows=674625) (actual time=0.199..285.958 rows=180084 loops=1)
12   -> Table scan on a (cost=67919.50 rows=674625) (actual time=0.065..212.152 rows=676184 loops=1)
13   -> Filter: (c.userId = a.userId) (cost=1.06 rows=1) (actual time=0.003..0.003 rows=0 loops=180084)
14   -> Index lookup on c using Comments_ibfk_2 (paperId=a.paperId) (cost=1.06 rows=1) (actual time=0.003..0.003 rows=0 loops=180084)
15   -> Filter: (n.userId = a.userId) (cost=1.02 rows=1) (actual time=0.002..0.002 rows=0 loops=180089)
16   -> Index lookup on n using Notes_ibfk_1 (paperId=a.paperId) (cost=1.02 rows=1) (actual time=0.002..0.002 rows=0 loops=180089)
17   -> Single-row index lookup on ax using PRIMARY (paperId=activity.paperId) (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=1248)

```



```

1  -> Nested loop inner join (cost=1786221.49 rows=365695) (actual time=1904.735..1908.691 rows=1248 loops=1)
2    -> Sort: activity.ratio DESC, activity.paperId DESC (cost=1657113.63..1657113.63 rows=365695) (actual time=1904.718..1904.834 rows=1248 loops=1)
3    -> Filter: (activity.paperId is not null) (cost=903585.84..944728.94 rows=365695) (actual time=1903.743..1903.954 rows=1248 loops=1)
4    -> Table scan on activity (cost=903585.74..908159.42 rows=365695) (actual time=1903.741..1903.865 rows=1248 loops=1)
5    -> Materialize (cost=903585.73..903585.73 rows=365695) (actual time=1903.739..1903.739 rows=1248 loops=1)
6    -> Filter: (((count(distinct c.userId) + count(distinct n.userId)) / count(distinct a.userId)) / 2) >= 0.9) (cost=867016.20 rows=365695) (actual time=795.719..1902.401 rows=1248 loops=1)
7    -> Group aggregate: count(distinct a.userId), count(distinct n.userId), count(distinct c.userId) (cost=867016.20 rows=365695) (actual time=793.404..1834.375 rows=154577 loops=1)
8    -> Nested loop left join (cost=830446.68 rows=365695) (actual time=793.379..1740.119 rows=180089 loops=1)
9      -> Nested loop left join (cost=429261.70 rows=358671) (actual time=793.360..1344.191 rows=180089 loops=1)
10        -> Sort: a.paperId (cost=35102.20 rows=337312) (actual time=793.309..811.345 rows=180084 loops=1)
11        -> Index lookup on a using idx_actionType (actionType='save') (actual time=0.054..675.926 rows=180084 loops=1)
12        -> Filter: (c.userId = a.userId) (cost=1.06 rows=1) (actual time=0.003..0.003 rows=0 loops=180084)
13        -> Index lookup on c using Comments_ibfk_2 (paperId=a.paperId) (cost=1.06 rows=1) (actual time=0.002..0.003 rows=0 loops=180084)
14        -> Filter: (n.userId = a.userId) (cost=1.02 rows=1) (actual time=0.002..0.002 rows=0 loops=180089)
15        -> Index lookup on n using Notes_ibfk_1 (paperId=a.paperId) (cost=1.02 rows=1) (actual time=0.002..0.002 rows=0 loops=180089)
16      -> Single-row index lookup on ax using PRIMARY (paperId=activity.paperId) (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=1248)

```

Analysis:

Looking at the visualized explain output from MySQL Workbench, we found that the full table scan used when joining the tables together was actually faster than the non-unique key lookup when adding an index on Actions.actionType. This could be that there are a lot of actions with the same action type so traversing the index is slower than a sequential table scan which can be relatively faster on disk. For this query, the best option is to just leave the primary and foreign key indices without adding any extra indices.

Query 4

Default index

Result:

```
27 # Default
28 -> Limit: 15 row(s) (actual time=2507.100..2507.101 rows=15 loops=1)
29 -> Sort: total DESC, ax.paperId, limit input to 15 row(s) per chunk (actual time=2507.099..2507.099 rows=15 loops=1)
30 -> Stream results (cost=3571395207.46 rows=35700076132) (actual time=1409.374..2435.932 rows=571050 loops=1)
31 -> Nested loop left join (cost=3571395207.46 rows=35700076132) (actual time=1409.368..2348.278 rows=571050 loops=1)
32 -> Table scan on ax (cost=64595.04 rows=529184) (actual time=0.209..296.203 rows=571050 loops=1)
33 -> Index lookup on subq using <auto_key0> (paperId=ax.paperId) (actual time=0.003..0.003 rows=0 loops=571050)
34 -> Materialize (cost=81412.00..81412.00 rows=67463) (actual time=1409.142..1409.142 rows=154577 loops=1)
35 -> Group aggregate: count(0) (cost=74665.75 rows=67463) (actual time=0.389..1225.756 rows=154577 loops=1)
36 -> Filter: (Actions.actionType = 'save') (cost=67919.50 rows=67463) (actual time=0.382..1192.797 rows=180084 loops=1)
37 -> Index scan on Actions using Actions_ibfk_1 (cost=67919.50 rows=674625) (actual time=0.375..1108.628 rows=676184 loops=1)
```

Total Cost = 3571395207.46

Analysis: The default indices were Actions(actionId, paperId, userId) and Arxiv(paperId). Other than these we can use Actions(actionType) and Arxiv(views) as possible indices.

Index Design 1

Indexing on only views.

CREATE INDEX idx_views ON Arxiv(views);

Result:

```
51 # views
52 -> Limit: 15 row(s) (actual time=2631.157..2631.159 rows=15 loops=1)
53 -> Sort: total DESC, ax.paperId, limit input to 15 row(s) per chunk (actual time=2631.156..2631.157 rows=15 loops=1)
54 -> Stream results (cost=3571395207.46 rows=35700076132) (actual time=1431.483..2557.880 rows=571050 loops=1)
55 -> Nested loop left join (cost=3571395207.46 rows=35700076132) (actual time=1431.476..2469.494 rows=571050 loops=1)
56 -> Covering index scan on ax using idx_views (cost=64595.04 rows=529184) (actual time=0.017..338.724 rows=571050 loops=1)
57 -> Index lookup on subq using <auto_key0> (paperId=ax.paperId) (actual time=0.004..0.004 rows=0 loops=571050)
58 -> Materialize (cost=81412.00..81412.00 rows=67463) (actual time=1431.440..1431.440 rows=154577 loops=1)
59 -> Group aggregate: count(0) (cost=74665.75 rows=67463) (actual time=0.347..1237.568 rows=154577 loops=1)
60 -> Filter: (Actions.actionType = 'save') (cost=67919.50 rows=67463) (actual time=0.342..1203.968 rows=180084 loops=1)
61 -> Index scan on Actions using Actions_ibfk_1 (cost=67919.50 rows=674625) (actual time=0.337..1120.357 rows=676184 loops=1)
```

Total Cost = 3571395207.46

Analysis: Adding **views** did not change the cost. This is probably because views are only used to calculate the *total* after the join operation and do not directly benefit from the index on views.

Index Design 2 FINAL DESIGN

Indexing on only actionType.

CREATE INDEX idx_actionType ON Actions(actionType);

Result:

```
39 # actionType
40 -> Limit: 15 row(s) (actual time=2032.804..2032.807 rows=15 loops=1)
41 -> Sort: total DESC, ax.paperId, limit input to 15 row(s) per chunk (actual time=2032.803..2032.805 rows=15 loops=1)
42 -> Stream results (cost=1387562.88 rows=0) (actual time=844.975..1944.181 rows=571050 loops=1)
43 -> Nested loop left join (cost=1387562.88 rows=0) (actual time=844.965..1846.321 rows=571050 loops=1)
44 -> Table scan on ax (cost=64595.04 rows=529184) (actual time=0.041..299.288 rows=571050 loops=1)
45 -> Index lookup on subq using <auto_key0> (paperId=ax.paperId) (actual time=0.002..0.003 rows=0 loops=571050)
46 -> Materialize (cost=0.00..0.00 rows=0) (actual time=844.910..844.910 rows=154577 loops=1)
47 -> Table scan on <temporary> (actual time=607.345..630.024 rows=154577 loops=1)
48 -> Aggregate using temporary table (actual time=607.343..607.343 rows=154577 loops=1)
49 -> Index lookup on Actions using idx_actionType (actionType='save') (cost=35102.20 rows=337312) (actual time=0.172..488.801
```

Total Cost = 1387562.88

Analysis: Adding actionType as an index decreased the cost significantly from 3571395207.46 to 1387562.88. This should be because it allows the database to quickly filter and retrieve only the relevant rows where actionType = 'save' without scanning the entire Actions table.

Index Design 3

Indexing on **actionType** and views.

CREATE INDEX idx_views ON Arxiv(views);

CREATE INDEX idx_actionType ON Actions(actionType);

Result:

```
64 # actionType + views
65 -> Limit: 15 row(s) (actual time=2251.807..2251.809 rows=15 loops=1)
66   -> Sort: total DESC, ax.paperId, limit input to 15 row(s) per chunk (actual time=2251.806..2251.807 rows=15 loops=1)
67     -> Stream results (cost=1387562.88 rows=0) (actual time=1097.058..2172.417 rows=571050 loops=1)
68       -> Nested loop left join (cost=1387562.88 rows=0) (actual time=1097.043..2078.667 rows=571050 loops=1)
69         -> Covering index scan on ax using idx_views (cost=64595.04 rows=529184) (actual time=0.033..157.449 rows=571050 loops=1)
70         -> Index lookup on subq using <auto_key0> (paperId=ax.paperId) (actual time=0.003..0.003 rows=0 loops=571050)
71         -> Materialize (cost=0.00..0.00 rows=0) (actual time=1096.997..1096.997 rows=154577 loops=1)
72         -> Table scan on <temporary> (actual time=801.432..826.399 rows=154577 loops=1)
73         -> Aggregate using temporary table (actual time=801.429..801.429 rows=154577 loops=1)
74         -> Index lookup on Actions using idx_actionType (actionType='save') (cost=35102.20 rows=337312) (actual time=0.191..682.471
```

Total Cost = 1387562.88

Analysis: Adding **views** to **actionType** also did not change the cost. This should be due to the same reasons as in Index Design 1. Views are only used in calculations. So our final index design for this query would be Design 2 since it drops the cost from 3571395207.46 to 1387562.88 which is a significant performance improvement.