CS 411 PT1 Stage 3: Database Creation and Indexing

DDL queries for all the tables:

1. Table Papers:

```
CREATE TABLE Papers (
   paper_id VARCHAR(50) PRIMARY KEY,
   paper_title VARCHAR(255) NOT NULL,
   abstract MEDIUMTEXT,
   pdf_url VARCHAR(500),
   upload_timestamp DATETIME,
   status VARCHAR(50),
   venue_id VARCHAR(10),
   project_id VARCHAR(50),
   dataset_id VARCHAR(50),
   FOREIGN KEY (venue_id) REFERENCES Venues(venue_id),
   FOREIGN KEY (project_id) REFERENCES Projects(project_id),
   FOREIGN KEY (dataset_id) REFERENCES Datasets(dataset_id)
);
```

Note: We can also add On Delete Cascade, however we haven't done it for this stage, since some of the data was hard coded.

2. Table Projects:

```
CREATE TABLE Projects (
   project_id VARCHAR(50) PRIMARY KEY,
```

```
    project_title VARCHAR(255) NOT NULL,
    description TEXT,
    project_date DATE
);
```

3. Table Venues

```
CREATE TABLE Venues (
    venue_id VARCHAR(10) PRIMARY KEY,
    venue_name VARCHAR(255) NOT NULL,
    venue_type VARCHAR(50),
    publisher VARCHAR(100),
    year SMALLINT
);
```

4. Table Datasets:

```
CREATE TABLE Datasets (
    dataset_id VARCHAR(50) PRIMARY KEY,
    dataset_name VARCHAR(255) NOT NULL,
    dataset_url VARCHAR(500),
    domain VARCHAR(100),
    access_type VARCHAR(50)
);
```

5. Table Users:

```
CREATE TABLE Users (
  user_id     VARCHAR(50)  NOT NULL,
```

```
  user_name    VARCHAR(255) NOT NULL,
  email        VARCHAR(320) UNIQUE NOT NULL,
  affiliation  VARCHAR(255),
  profile_url  VARCHAR(500),
  is_reviewer TINYINT(1) NOT NULL DEFAULT 0,
  PRIMARY KEY (user_id),
) ENGINE=InnoDB;
```

6. Table Reviews:

```
CREATE TABLE Reviews (
  review_id        VARCHAR(50)  NOT NULL,
  user_id          VARCHAR(50)  NOT NULL,
  paper_id         VARCHAR(50)  NOT NULL,
  comment          TEXT,
  review_timestamp  DATETIME,
  PRIMARY KEY (review_id),
  CONSTRAINT fk_reviews_user   FOREIGN KEY (user_id)  REFERENCES Users(user_id),
  CONSTRAINT fk_reviews_paper  FOREIGN KEY (paper_id) REFERENCES Papers(paper_id),
) ENGINE=InnoDB;
```

7. Table Authorship:

```
CREATE TABLE Authorship (
  user_id    VARCHAR(50) NOT NULL,
  paper_id   VARCHAR(50) NOT NULL,
  is_primary TINYINT(1)  NOT NULL DEFAULT 0,
  PRIMARY KEY (user_id, paper_id),
```

CONSTRAINT fk_authorship_user FOREIGN KEY (user_id) REFERENCES Users(user_id),

CONSTRAINT fk_authorship_paper FOREIGN KEY (paper_id) REFERENCES Papers(paper_id)

) ENGINE=InnoDB;

8. Table RelatedPapers:
CREATE TABLE RelatedPapers (

paper_id   VARCHAR(50) NOT NULL,

related_paper_id VARCHAR(50) NOT NULL,

PRIMARY KEY (paper_id, related_paper_id),

CONSTRAINT chk_not_self CHECK (paper_id <> related_paper_id),

CONSTRAINT fk_relpapers_left   FOREIGN KEY (paper_id)   REFERENCES Papers(paper_id),

CONSTRAINT fk_relpapers_right FOREIGN KEY (related_paper_id) REFERENCES Papers(paper_id)

) ENGINE=InnoDB;

Count of tables where rows>1000

```
mysql> Select Count(*) from datasets;
+----------+
| Count(*) |
+----------+
|     2005 |
+----------+
1 row in set (0.00 sec)

mysql> Select count(*) from papers;
+----------+
| count(*) |
+----------+
|     4000 |
+----------+
1 row in set (0.00 sec)

mysql> Select count(*) from projects;
+----------+
| count(*) |
+----------+
|     3281 |
+----------+
1 row in set (0.00 sec)
```

Count(*) for other tables:

```
mysql> Select count(*) from Venues;
+----------+
| count(*) |
+----------+
|      100 |
+----------+
1 row in set (0.00 sec)

mysql> Select count(*) from Users;
+----------+
| count(*) |
+----------+
|       40 |
+----------+
1 row in set (0.00 sec)

mysql> Select count(*) from Authorship;
+----------+
| count(*) |
+----------+
|     2564 |
+----------+
1 row in set (0.00 sec)

mysql> Select count(*) from Reviews;
+----------+
| count(*) |
+----------+
|      570 |
+----------+
1 row in set (0.00 sec)

mysql> Select count(*) from RelatedPapers;
+----------+
| count(*) |
+----------+
|       10 |
+----------+
1 row in set (0.00 sec)
```

Advanced Queries:

```
-- Q1 --

-- Lists all projects and papers authored by a selected user(U005) since a given date,

-- showing how many reviews each paper has received.

EXPLAIN SELECT

    pr.project_id,

    pr.project_title,

    p.paper_id,

    p.paper_title,

    p.upload_timestamp,

    COUNT(r.review_id) AS review_count

FROM Authorship a
```

```
JOIN Papers p
  ON a.paper_id = p.paper_id
JOIN Projects pr
  ON p.project_id = pr.project_id
LEFT JOIN Reviews r
  ON p.paper_id = r.paper_id
WHERE a.user_id = 'U005'
 AND p.upload_timestamp >= '2018-01-01'
GROUP BY pr.project_id, pr.project_title, p.paper_id, p.paper_title, p.upload_timestamp
ORDER BY p.upload_timestamp DESC, review_count DESC
LIMIT 15;
```

Output:

project_id	project_title	paper_id	paper_title	upload_timestamp	review_count
8c53e26c6973dabc	gated path planning networks	ee34b900db20f6bd	Temporal Difference Variational Auto-Encoder	2018-06-08 00:00:00	0
d3b0aac7098ffad5	low shot learning with large scale diffusion	53ddebc92aba3210	BOCK : Bayesian Optimization with Cylindrical…	2018-06-05 00:00:00	0
54f45c0fbfba9c11	selfless sequential learning	9c7a6cf8da44f6e8	A Survey of Domain Adaptation for Neural Mach…	2018-06-01 00:00:00	0
7ecd8d51d64e5aa7	wikiref wikilinks as a route to recommending ap…	14b5c13a85d1f781	Approximate Knowledge Compilation by Online…	2018-05-31 00:00:00	0
883f3f83cebef926	to understand deep learning we need to underst…	9525f4a1587ab8eb	CRRN: Multi-Scale Guided Concurrent Reflectio…	2018-05-30 00:00:00	0
515d6cb116c675b3	learning deep resnet blocks sequentially using b…	96e8d75c74d892ce	Polyglot Semantic Role Labeling	2018-05-29 00:00:00	3
00dbf4e597303af6	learning in pomdps with monte carlo tree search	30928cec4b22ae91	Sigsoftmax: Reanalysis of the Softmax Bottleneck	2018-05-28 00:00:00	3
23b1b59fd3fe031e	teaching multiple concepts to a forgetful learner	5f8380ddc9c21f5c	Lipschitz regularity of deep neural networks: an…	2018-05-28 00:00:00	3
24080539b4589947	a survey on open information extraction	f2db5fb0e9e19dd7	Reliability and Learnability of Human Bandit Fee…	2018-05-27 00:00:00	0
fb0a1d686ffc86f7	entity commonsense representation for neural a…	3723fd9a9f84a718	Towards More Efficient Stochastic Decentralize…	2018-05-25 00:00:00	0
41fd68588b0d1939	semaxis a lightweight framework to characterize…	8b228622f75db232	Heterogeneous Bitwidth Binarization in Convolu…	2018-05-25 00:00:00	0
fb0a1d686ffc86f7	entity commonsense representation for neural a…	fd51028e3fd8728c	Robust Distant Supervision Relation Extraction…	2018-05-24 00:00:00	2
134db750be9874c6	smhd a large scale resource for exploring online…	73fedf627e66a4fd	Optimizing the F-measure for Threshold-free Sa…	2018-05-19 00:00:00	0
349d93e371a67885	bringing replication and reproduction together wi…	0e089ac357993330	PG-TS: Improved Thompson Sampling for Logis…	2018-05-18 00:00:00	0
966bba78b68db3f9	generative neural machine translation	c1ca887445078048	Extrapolation in NLP	2018-05-17 00:00:00	3

```
-- Q2 --
-- Displays all venues with the count of published papers in or after 2018.
-- Helps identify recent publication activity per venue across years.
EXPLAIN SELECT
```

```sql
    v.venue_id,
    v.venue_name,
    v.year,
    COUNT(p.paper_id) AS total_papers
FROM Venues v
JOIN Papers p
    ON v.venue_id = p.venue_id
WHERE v.year >= 2018
  AND p.status IN ('Published')
GROUP BY v.venue_id, v.venue_name, v.year
ORDER BY v.year DESC, total_papers DESC
LIMIT 15;
```

Output:

venue_id	venue_name	year	total_pape...
V00027	evaluation-of-unsupervised-compositional-1	2018	51
V0000R	gated-path-planning-networks-1	2018	51
V0000M	Unknown Conference	2018	50
V0000G	constraining-the-dynamics-of-deep-1	2018	49
V0001E	unsupervised-training-for-3d-morphable-model-1	2018	49
V0002L	on-accurate-evaluation-of-gans-for-language-1	2018	49
V0001M	a-dataset-for-building-code-mixed-goal-2	2018	48
V0000W	minimal-i-map-mcmc-for-scalable-structure-1	2018	48
V0000A	snap-ml-a-hierarchical-framework-for-machine-1	2018	48
V0000H	deforming-autoencoders-unsupervised-1	2018	48
V0000S	multimodal-grounding-for-language-processing-1	2018	48
V0000K	ncrf-an-open-source-neural-sequence-labeling-1	2018	47
V00011	learning-towards-minimum-hyperspherical-1	2018	46
V0002D	there-are-many-consistent-explanations-of-1	2018	46
V0000Z	gile-a-generalized-input-label-embedding-for-1	2018	45

```sql
-- Q3 --

-- Ranks reviewers based on the number of reviews they provided within a single day.

-- Filters users marked as reviewers and counts reviews within a time range.

SELECT

    u.user_id,

    u.user_name,

    u.affiliation,

    COUNT(DISTINCT a.paper_id) AS total_papers_authored,

    COUNT(r.review_id) AS total_reviews_received

FROM Users u

JOIN Authorship a

    ON u.user_id = a.user_id

JOIN Reviews r

    ON a.paper_id = r.paper_id

WHERE u.is_reviewer = TRUE

  AND r.review_timestamp BETWEEN '2024-02-15 00:00:00' AND '2024-05-15 23:00:00'

GROUP BY u.user_id, u.user_name, u.affiliation

HAVING COUNT(r.review_id) > 0

ORDER BY total_reviews_received DESC

LIMIT 15;
```

Output: The number of records are less than 15. This if for 3 months in the year 2024, hence the limited amount of output data.

user_id	user_name	affiliation	total_papers_author...	total_reviews_receiv...
U006	Farhan Malik	University of Toronto	21	48
U004	David Patel	UC Berkeley	19	47
U005	Elena Garcia	Carnegie Mellon University	19	47
U002	Brian Chen	Stanford University	18	43
U008	Henry Nguyen	ETH Zurich	19	38
U003	Catherine Li	MIT CSAIL	15	35
U010	Jack Miller	Harvard SEAS	16	35
U007	Grace Zhou	Oxford University	13	31
U001	Alice Kim	UIUC	13	28
U009	Isha Sharma	IIT Delhi	12	25

```
-- Q4 --

-- Lists all papers authored by a given user and reports both:

--  (a) how many total reviews each paper has, and

--  (b) the most recent review timestamp.

-- Allows tracking which of the author's works are actively discussed or recently reviewed.


EXPLAIN SELECT

    p.paper_id,

    p.paper_title,

    COUNT(r.review_id) AS review_count,

    MAX(r.review_timestamp) AS last_review_at

FROM Authorship a

JOIN Papers p

    ON a.paper_id = p.paper_id

LEFT JOIN Reviews r
```

```
    ON p.paper_id = r.paper_id
WHERE a.user_id = 'U010'
GROUP BY p.paper_id, p.paper_title
ORDER BY review_count DESC, last_review_at DESC
LIMIT 15;
```

Output:

paper_id	paper_title	review_count	last_review_at
a024340399f521d0	Net2Vec: Quantifying and Explaining how Conc…	3	2024-06-29 17:07:00
507e6eeab1ddee7c	Human Pose Estimation using Global and Local…	3	2024-06-26 11:14:00
a7490d5461da5afa	Coloring with Words: Guiding Image Colorizatio…	3	2024-06-03 12:31:00
6f72b53719831b73	Cut, Paste and Learn: Surprisingly Easy Synthe…	3	2024-04-14 10:17:00
a3c38391d3f7d130	Seq2SQL: Generating Structured Queries from…	3	2024-03-23 16:11:00
7ebeb187f78dfbf6	Predict Responsibly: Improving Fairness and Ac…	2	2024-06-23 17:41:00
d4e496e148d2839e	Efficient Video Object Segmentation via Networ…	2	2024-05-29 16:43:00
9085c520c1fdabee	Importance Weighted Transfer of Samples in Re…	2	2024-05-27 10:04:00
0ecfab1ecb257327	Autoregressive Convolutional Neural Networks f…	2	2024-05-26 15:36:00
9568a08410f139a7	ISO-Standard Domain-Independent Dialogue Ac…	2	2024-05-15 12:02:00
2bee38f1ba7075cb	COCO-Stuff: Thing and Stuff Classes in Context	2	2024-05-13 17:45:00
c4528cdae0dcb06c	Gradient Estimators for Implicit Models	2	2024-05-04 10:55:00
3242cd37c75f7607	MAP inference via Block-Coordinate Frank-Wolf…	2	2024-04-08 08:01:00
a565fe77cb1693ad	Learning Structure and Strength of CNN Filters f…	2	2024-03-07 11:19:00
12289695e3f51161	Probabilistic Model-Agnostic Meta-Learning	1	2024-04-03 14:25:00

Indexing Analysis:

We tested three different indexing designs for the attributes upload_timestamp, review_timestamp, and year to study their effect on query performance. Indexing showed clear improvement for upload_timestamp and review_timestamp because these columns are not naturally sorted and are often used in range-based filters. MySQL used the indexes to quickly find the relevant records within specific time ranges, reducing the cost and improving query efficiency.

In contrast, indexing on the year column in the Venues table provided limited benefits because the data was already grouped and relatively sorted. When the query filtered across many years, MySQL determined that scanning the entire table was faster than using the index. Overall, indexing performed best for ranged queries on unsorted attributes, where it helped MySQL access only the required records instead of reading the entire table.

Indexing of Attributes –

Advanced query1:

Before indexing – for year 2024

'-> Limit: 15 row(s)\n   -> Sort: p.upload_timestamp DESC, review_count DESC, limit input to 15 row(s) per chunk\n   -> Table scan on <temporary>\n   -> Aggregate using temporary table\n   -> Nested loop left join (cost=77.6 rows=111)\n   -> Nested loop inner join (cost=52.6 rows=52.4)\n   -> Nested loop inner join (cost=34.3 rows=52.4)\n -> Covering index lookup on a using PRIMARY (user_id = \'U005\') (cost=9.06 rows=72)\n -> Filter: ((p.upload_timestamp >= TIMESTAMP\'2018-01-01 00:00:00\') and (p.project_id is not null)) (cost=0.251 rows=0.728)\n   -> Single-row index lookup on p using PRIMARY (paper_id = a.paper_id) (cost=0.251 rows=1)\n   -> Single-row index lookup on pr using PRIMARY (project_id = p.project_id) (cost=0.252 rows=1)\n   -> Covering index lookup on r using idx_reviews_paper (paper_id = a.paper_id) (cost=0.267 rows=2.13)\n'

After indexing – for year 2024

Index : CREATE INDEX ix_papers_upload_time ON Papers(upload_timestamp);

'-> Limit: 15 row(s)\n   -> Sort: p.upload_timestamp DESC, review_count DESC, limit input to 15 row(s) per chunk\n   -> Table scan on <temporary>\n   -> Aggregate using temporary table\n   -> Nested loop left join (cost=77.6 rows=111)\n   -> Nested loop inner join (cost=52.6 rows=52.4)\n   -> Nested loop inner join (cost=34.3 rows=52.4)\n -> Covering index lookup on a using PRIMARY (user_id = \'U005\') (cost=9.06 rows=72)\n -> Filter: ((p.upload_timestamp >= TIMESTAMP\'2018-01-01 00:00:00\') and (p.project_id is not null)) (cost=0.251 rows=0.728)\n   -> Single-row index lookup on p using PRIMARY (paper_id = a.paper_id) (cost=0.251 rows=1)\n   -> Single-row index lookup on pr using PRIMARY (project_id = p.project_id) (cost=0.252 rows=1)\n   -> Covering index lookup on r using idx_reviews_paper (paper_id = a.paper_id) (cost=0.267 rows=2.13)\n'

Before indexing – for year 2018

'-> Limit: 15 row(s)\n -> Sort: p.upload_timestamp DESC, review_count DESC, limit input to 15 row(s) per chunk\n -> Table scan on <temporary>\n -> Aggregate using temporary table\n -> Nested loop left join (cost=54.1 rows=51)\n -> Nested loop inner join (cost=42.7 rows=24)\n -> Nested loop inner join (cost=34.3 rows=24)\n -> Covering index lookup on a using PRIMARY (user_id = \'U005\') (cost=9.06 rows=72)\n -> Filter: ((p.upload_timestamp >= TIMESTAMP\'2024-01-01 00:00:00\') and (p.project_id is not null)) (cost=0.25 rows=0.333)\n -> Single-row index lookup on p using PRIMARY (paper_id = a.paper_id) (cost=0.25 rows=1)\n -> Single-row index lookup on pr using PRIMARY (project_id = p.project_id) (cost=0.254 rows=1)\n -> Covering index lookup on r using idx_reviews_paper (paper_id = a.paper_id) (cost=0.272 rows=2.13)\n'

After indexing – for year 2018

Index: CREATE INDEX ix_papers_upload_time ON Papers(upload_timestamp);

-> Limit: 15 row(s)

-> Sort: p.upload_timestamp DESC, review_count DESC, limit input to 15 row(s) per chunk

-> Table scan on <temporary>

-> Aggregate using temporary table

-> Nested loop left join (cost=1.89 rows=2.13)

-> Nested loop inner join (cost=1.41 rows=1)

-> Nested loop inner join (cost=1.06 rows=1)

-> Filter: (p.project_id is not null) (cost=0.71 rows=1)

-> Index range scan on p using ix_papers_upload_time over ('2024-01-01 00:00:00' <= upload_timestamp), with index condition: (p.upload_timestamp >= TIMESTAMP'2024-01-01 00:00:00') (cost=0.71 rows=1)

-> Single-row covering index lookup on a using PRIMARY (user_id = 'U005', paper_id = p.paper_id) (cost=0.35 rows=1)

-> Single-row index lookup on pr using PRIMARY (project_id = p.project_id) (cost=0.35 rows=1)

-> Covering index lookup on r using idx_reviews_paper (paper_id = p.paper_id) (cost=0.476 rows=2.13)

Justification:

When MySQL runs a query, it tries to choose the fastest way to get the data. It decides whether to use an index or to read the entire table by comparing which method will take less effort overall.

An index is helpful only when the condition in the query filters out a small portion of rows. This is because MySQL can then directly jump to the few rows that match the condition. However, if the condition matches a large number of rows, the index becomes less useful. In such cases, MySQL finds it faster to simply read the whole table in order, instead of using the index to find each matching row one by one.

In this case, when we use the date 2024-01-01, very few papers meet the condition. MySQL uses the index to directly look up those few records, which saves time and reduces cost.

When we use the date 2018-01-01, many papers meet the condition. Using the index now means MySQL would still need to access almost every row, but it would have to do it in small random steps. That takes longer than just reading the table from start to end. So MySQL decides not to use the index and instead performs a simple table scan.

In short, indexes work best when the query looks for a small number of rows. If most rows match the condition, scanning the table directly is faster and more efficient.

Query 2:

Before Indexing: Year **2020**

'-> Limit: 15 row(s)\n   -> Sort: v.`year` DESC, total_papers DESC, limit input to 15 row(s) per chunk\n   -> Table scan on <temporary>\n   -> Aggregate using temporary table\n -> Nested loop inner join (cost=345 rows=95.6)\n   -> Filter: (v.`year` >= 2020) (cost=10.2 rows=33.3)\n   -> Table scan on v (cost=10.2 rows=100)\n   -> Filter: (p.`status` = \'Published\') (cost=7.18 rows=2.87)\n   -> Index lookup on p using venue_id (venue_id = v.venue_id) (cost=7.18 rows=28.7)\n'

After Indexing : **2020**
Index:
CREATE INDEX ix_venues_year

'-> Limit: 15 row(s)\n   -> Sort: v.`year` DESC, total_papers DESC, limit input to 15 row(s) per chunk\n -> Table scan on <temporary>\n -> Aggregate using temporary table\n -> Nested loop inner join (cost=10.7 rows=2.87)\n -> Index range scan on v using ix_venues_year over (2020 <= year), with index condition: (v.`year` >= 2020) (cost=0.71 rows=1)\n -> Filter: (p.`status` = \'Published\') (cost=7.46 rows=2.87)\n -> Index lookup on p using venue_id (venue_id = v.venue_id) (cost=7.46 rows=28.7)\n'

Before Indexing: 2020

'-> Limit: 15 row(s)\n -> Sort: v.`year` DESC, total_papers DESC, limit input to 15 row(s) per chunk\n -> Table scan on <temporary>\n -> Aggregate using temporary table\n -> Nested loop inner join (cost=345 rows=95.6)\n -> Filter: (v.`year` >= 2018) (cost=10.2 rows=33.3)\n -> Table scan on v (cost=10.2 rows=100)\n -> Filter: (p.`status` = \'Published\') (cost=7.18 rows=2.87)\n -> Index lookup on p using venue_id (venue_id = v.venue_id) (cost=7.18 rows=28.7)\n'

After indexing: 2020

'-> Limit: 15 row(s)\n -> Sort: v.`year` DESC, total_papers DESC, limit input to 15 row(s) per chunk\n -> Table scan on <temporary>\n -> Aggregate using temporary table\n -> Nested loop inner join (cost=543 rows=235)\n -> Filter: ((p.`status` = \'Published\') and (p.venue_id is not null)) (cost=442 rows=290)\n -> Table scan on p (cost=442 rows=2896)\n -> Filter: (v.`year` >= 2018) (cost=0.25 rows=0.81)\n -> Single-row index lookup on v using PRIMARY (venue_id = p.venue_id) (cost=0.25 rows=1)\n'

When MySQL executed the query before indexing, it had to read the entire **Venues** table to find the records where year >= 2020. It performed a **table scan** on the Venues table, meaning it checked every row one by one, even if only a few rows matched the condition. This resulted in a relatively high cost of **345**, as MySQL had to process many unnecessary rows before joining them with the Papers table. After the index was created on the year column (ix_venues_year), MySQL was able to directly locate only the relevant records where the year was greater than or equal to 2020. The plan changed to an **index range scan**, which allowed MySQL to quickly jump to the matching years instead of reading the entire table. This reduced the cost significantly to around **10.7**, showing a clear improvement in efficiency when filtering recent years.

However, when the condition was relaxed to year >= 2018, the optimizer changed its strategy. In this case, a larger number of rows matched the condition, so MySQL found that using the index would still involve reading most of the table. It decided instead to perform a **table scan on the Papers table** and use a primary key lookup for each venue. The total cost increased to

543, higher than before, because even though an index existed, the optimizer determined that scanning the table directly was faster than using the index for such a broad range. This shows that while indexing improves performance for selective filters, it can increase cost when the condition covers too many rows.

3

Before indexing:

'-> Limit: 15 row(s)\n   -> Sort: total_reviews_received DESC\n   -> Filter: (count(reviews.review_id) > 0)\n   -> Stream results\n   -> Group aggregate: count(reviews.review_id), count(distinct authorship.paper_id), count(reviews.review_id)\n -> Sort: u.user_id, u.user_name, u.affiliation\n   -> Stream results (cost=147 rows=15.8)\n   -> Nested loop inner join (cost=147 rows=15.8)\n -> Nested loop inner join (cost=91.9 rows=158)\n   -> Filter: (r.review_timestamp between \'2024-02-15 00:00:00\' and \'2024-05-15 23:00:00\') (cost=58.5 rows=63.3)\n   -> Covering index scan on r using idx_reviews_paper_time (cost=58.5 rows=570)\n   -> Covering index lookup on a using ix_authorship_paper (paper_id = r.paper_id) (cost=0.281 rows=2.5)\n   -> Filter:

(u.is_reviewer = true) (cost=0.25 rows=0.1)\n   -> Single-row index lookup on u using PRIMARY (user_id = a.user_id) (cost=0.25 rows=1)\n'

After indexing:

'-> Limit: 15 row(s)\n   -> Sort: total_reviews_received DESC\n   -> Filter: (count(reviews.review_id) > 0)\n   -> Stream results\n   -> Group aggregate: count(reviews.review_id), count(distinct authorship.paper_id), count(reviews.review_id)\n -> Sort: u.user_id, u.user_name, u.affiliation\n   -> Stream results (cost=147 rows=15.8)\n   -> Nested loop inner join (cost=147 rows=15.8)\n -> Nested loop inner join (cost=91.9 rows=158)\n   -> Filter: (r.review_timestamp between \'2024-02-15 00:00:00\' and \'2024-05-15 23:00:00\') (cost=58.5 rows=63.3)\n   -> Table scan on r (cost=58.5 rows=570)\n -> Covering index lookup on a using ix_authorship_paper (paper_id = r.paper_id) (cost=0.281 rows=2.5)\n   -> Filter: (u.is_reviewer = true) (cost=0.25 rows=0.1)\n -> Single-row index lookup on u using PRIMARY (user_id = a.user_id) (cost=0.25 rows=1)\n'

The new index on Users.is_reviewer did not improve performance because the Users table is very small and the filter is not selective (about half the rows are reviewers.)

For such cases, MySQL finds it faster to read all users instead of using an index. This is why the cost and query plan look almost identical before and after indexing. The earlier index on Reviews(paper_id, review_timestamp) was already doing the main optimization work, and adding another index on a low-selectivity column brought no extra benefit. // again this is because less rows filtered

4.

Query 4:

'-> Limit: 15 row(s)\n   -> Sort: review_count DESC, last_review_at DESC, limit input to 15 row(s) per chunk\n   -> Table scan on <temporary>\n   -> Aggregate using temporary table\n   -> Nested loop left join (cost=85.4 rows=149)\n   -> Nested loop inner join (cost=33.3 rows=70)\n   -> Covering index lookup on a using PRIMARY (user_id = \'U010\') (cost=8.82 rows=70)\n   -> Single-row index lookup on p using PRIMARY (paper_id = a.paper_id) (cost=0.251 rows=1)\n   -> Index lookup on r using idx_reviews_paper (paper_id = a.paper_id) (cost=0.535 rows=2.13)\n'

We didn't add index:

When MySQL ran this query, it already had indexes on the primary and foreign key columns created automatically by the DDL. Every table's primary key automatically becomes an index, and every foreign key also creates an index on its referenced column if one doesn't already exist. These default indexes are necessary for enforcing referential integrity and for speeding up joins between related tables. In this query, the joins use columns such as user_id (from Authorship) and paper_id (from Papers and Reviews). Because these are foreign keys, MySQL already had indexes available for them.

Since the query filters on a specific user (user_id = 'U010') and joins through foreign keys, MySQL used these built-in indexes to perform covering index lookups and single-row index lookups, which are already efficient. That is why the cost stayed moderate (around 85.4) even without adding new indexes. If we tried to add another custom index on these same key columns, it would not reduce the cost because MySQL is already using the existing indexes created for referential constraints. In short, the query's performance was already optimized by default indexes that came from primary and foreign key definitions, so there was no measurable gain from adding new ones manually.

Local Connection with mySql using Terminal:

```
hardiklad10 — mysql --local-infile=1 -u root -p — 124×35
Last login: Thu Oct 30 21:31:09 on ttys001
(base) hardiklad10@Hardiks-MacBook-Pro ~ % mysql --local-infile=1 -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 45
Server version: 9.5.0 MySQL Community Server - GPL

Copyright (c) 2000, 2025, Oracle and/or its affiliates.

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

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

mysql> Use PapersDB
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> Select Count(*) from papers;
+----------+
| Count(*) |
+----------+
|     4000 |
+----------+
1 row in set (0.00 sec)
```