

We hosted the database tables on the GCP:

```
[(base) jackyzhang@wirelessprv-10-195-220-153 ~ % mysql --ssl-mode=DISABLED --host=34.59.117.251 --user=root --password
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1167
Server version: 8.4.3-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>
```

connect command:

mysql --ssl-mode=DISABLED --host=34.59.117.251 --user=root --password

DDL commands

CREATE TABLE Users (

- > Id INT PRIMARY KEY,
- > Email VARCHAR(255) UNIQUE NOT NULL,
- > Password VARCHAR(255) NOT NULL,
- > FinalSchool VARCHAR(255)

->);

CREATE TABLE Schools (

- > School_id INT PRIMARY KEY,
- > Name VARCHAR(255) UNIQUE NOT NULL,
- > Location VARCHAR(255) NOT NULL,
- > Tuition DECIMAL(10,2) NOT NULL

->);

CREATE TABLE User_Schools (

- > User_id INT NOT NULL,
- > SchoolId INT NOT NULL,
- > PRIMARY KEY (User_id, SchoolId),
- > FOREIGN KEY (User_id) REFERENCES Users(Id),
- > FOREIGN KEY (SchoolId) REFERENCES Schools(School_id)

->);

CREATE TABLE NewsFeed (

- > Id INT PRIMARY KEY,
- > User_id INT NOT NULL,
- > School_id INT NOT NULL,
- > Academic INT DEFAULT 0 NOT NULL,
- > Tuition INT DEFAULT 0 NOT NULL,
- > Location INT DEFAULT 0 NOT NULL,
- > Comment TEXT,
- > Timestamp DATETIME NOT NULL,
- > FOREIGN KEY (User_id) REFERENCES Users(Id),
- > FOREIGN KEY (School_id) REFERENCES Schools(School_id)

->);

CREATE TABLE US_NEWS_RANKING (

- > Id INT PRIMARY KEY,

```

-> School_id INT UNIQUE NOT NULL,
-> Ranking INT NOT NULL,
-> FOREIGN KEY (School_id) REFERENCES Schools(School_id)
-> );

```

```

CREATE TABLE QS_RANKING (
-> Id INT PRIMARY KEY,
-> School_id INT UNIQUE NOT NULL,
-> Ranking INT NOT NULL,
-> FOREIGN KEY (School_id) REFERENCES Schools(School_id)
-> );

```

Inserted data:

User Table:

986	user986@example.com	password986	Harvard
987	user987@example.com	password987	MIT
988	user988@example.com	password988	Stanford
989	user989@example.com	password989	Yale
990	user990@example.com	password990	Princeton
991	user991@example.com	password991	Harvard
992	user992@example.com	password992	MIT
993	user993@example.com	password993	Stanford
994	user994@example.com	password994	Yale
995	user995@example.com	password995	Princeton
996	user996@example.com	password996	Harvard
997	user997@example.com	password997	MIT
998	user998@example.com	password998	Stanford
999	user999@example.com	password999	Yale
1000	user1000@example.com	password1000	Princeton

1000 rows in set (0.10 sec)

Schools Table:

250	Humboldt State University	Arcata, CA
251	Indiana State University	Terre Haute, IN
252	University of Central Oklahoma	Edmond, OK
253	Western Washington University	Bellingham, WA
254	Wayne State University	Detroit, MI
255	Georgia State University	Atlanta, GA
256	Swarthmore College	Swarthmore, PA
257	University of Texas--Arlington	Arlington, TX

258 rows in set (0.03 sec)

User_Schools Table:

787	224
795	224
861	224
877	224
4	225
219	225
323	225
409	225
460	225
226	226
308	226
657	226
860	226
100	227
633	227
759	227
933	227
492	228
506	228
685	228
932	228
+-----+	
1000 rows in set (0.03 sec)	

NewsFeed Table:

185	246	186
186	195	187
187	247	188
188	248	189
189	249	190
190	250	191
191	151	192
192	251	193
193	216	194
194	252	195
195	218	196
196	253	197

-----+

194 rows in set (0.03 sec)

SQL Queries:

1. Combined Ranking Calculation

Purpose: Calculate the average ranking score across all ranking systems (QS, US News, THE) for each school.

```
SELECT
  s.School_id,
  s.Name,
  ROUND(AVG(qs.Ranking), 1) AS Avg_QS_Rank,
  ROUND(AVG(usn.Ranking), 1) AS Avg_USNews_Rank,
  ROUND((COALESCE(qs.Ranking, 1000) + COALESCE(usn.Ranking, 1000)) / 2, 1) AS Combined_Score
FROM Schools s
LEFT JOIN QS_RANKING qs ON s.School_id = qs.School_id
LEFT JOIN US_NEWS_RANKING usn ON s.School_id = usn.School_id
GROUP BY s.School_id, s.Name
ORDER BY Combined_Score ASC;
```

No indexing

and cost analysis:

```
| -> Sort: Combined_Score (actual time=2.18..2.2 rows=258 loops=1)
  -> Table scan on <temporary> (actual time=1.98..2.02 rows=258 loops=1)
    -> Aggregate using temporary table (actual time=1.98..1.98 rows=258 loops=1)
      -> Nested loop left join (cost=207 rows=258) (actual time=0.0843..1.44 rows=258 loops=1)
        -> Nested loop left join (cost=117 rows=258) (actual time=0.0733..0.763 rows=258 loops=1)
          -> Covering index scan on s using Name (cost=26.6 rows=258) (actual time=0.047..0.135 rows=258
loops=1)
            -> Single-row index lookup on qs using School_id (School_id=s.School_id) (cost=0.25 rows=1) (actual
time=0.00222..0.00224 rows=0.752 loops=258)
```

-> Single-row index lookup on usn using School_id (School_id=s.School_id) (cost=0.25 rows=1) (actual time=0.0024..0.00243 rows=0.895 loops=258)
|

top 15 from raw query

School_id	Name	Avg_QS_Rank	Avg_USNews_Rank	Combined_Score
1	Harvard University	2.0	2.0	2.0
6	Massachusetts Institute of Technology	1.0	7.0	4.0
5	Stanford University	3.0	6.0	4.5
0	Princeton University	9.0	1.0	5.0
2	University of Chicago	8.0	3.0	5.5
8	University of Pennsylvania	5.0	9.0	7.0
3	Yale University	10.0	4.0	7.0
11	California Institute of Technology	4.0	12.0	8.0
4	Columbia University	12.0	5.0	8.5
9	Johns Hopkins University	11.0	10.0	10.5
14	Cornell University	7.0	15.0	11.0
7	Duke University	18.0	8.0	13.0
21	University of California--Berkeley	6.0	22.0	14.0
12	Northwestern University	16.0	13.0	14.5
13	Brown University	23.0	14.0	18.5
24	University of California--Los Angeles	13.0	25.0	19.0
23	Carnegie Mellon University	17.0	24.0	20.5
27	University of Michigan--Ann Arbor	15.0	28.0	21.5
15	Rice University	31.0	16.0	23.5
35	New York University	14.0	36.0	25.0
22	University of Southern California	29.0	23.0	26.0
18	Washington University in St. Louis	35.0	19.0	27.0

Index on Schools(Name) - CREATE INDEX idx_schools_name ON Schools(School_id, Name);

| -> Sort: Combined_Score (actual time=2.17..2.2 rows=258 loops=1)
-> Table scan on <temporary> (actual time=1.99..2.03 rows=258 loops=1)
-> Aggregate using temporary table (actual time=1.99..1.99 rows=258 loops=1)
-> Nested loop left join (cost=207 rows=258) (actual time=0.113..1.45 rows=258 loops=1)
-> Nested loop left join (cost=117 rows=258) (actual time=0.0976..0.788 rows=258 loops=1)
-> Covering index scan on s using Name (cost=26.6 rows=258) (actual time=0.0619..0.145 rows=258 loops=1)
-> Single-row index lookup on qs using School_id (School_id=s.School_id) (cost=0.25 rows=1) (actual time=0.00226..0.00229 rows=0.752 loops=258)
-> Single-row index lookup on usn using School_id (School_id=s.School_id) (cost=0.25 rows=1) (actual time=0.00233..0.00236 rows=0.895 loops=258)
|

Since this query groups by s.Name and s.School_ids (because its a primary key), we thought indexing on Schools(Name) would help cut costs, but it resulted in the same cost as the base query without this index.

Index on Tuition with edited query - CREATE INDEX idx_schools_tuition ON Schools(Tuition);

new query:

```
EXPLAIN ANALYZE
SELECT
  s.School_id,
  s.Name,
  s.Tuition,
  ROUND(AVG(qs.Ranking), 1) AS Avg_QS_Rank,
  ROUND(AVG(usn.Ranking), 1) AS Avg_USNews_Rank,
  ROUND((COALESCE(qs.Ranking, 1000) + COALESCE(usn.Ranking, 1000)) / 2, 1) AS Combined_Score
FROM Schools s
LEFT JOIN QS_RANKING qs ON s.School_id = qs.School_id
LEFT JOIN US_NEWS_RANKING usn ON s.School_id = usn.School_id
```

GROUP BY s.School_id, s.Name, s.Tuition
ORDER BY s.Tuition, Combined_Score ASC;

| -> Sort: s.Tuition, Combined_Score (actual time=2.29..2.32 rows=258 loops=1)
-> Table scan on <temporary> (actual time=2.02..2.06 rows=258 loops=1)
-> Aggregate using temporary table (actual time=2.02..2.02 rows=258 loops=1)
-> Nested loop left join (cost=207 rows=258) (actual time=0.086..1.44 rows=258 loops=1)
-> Nested loop left join (cost=117 rows=258) (actual time=0.0709..0.84 rows=258 loops=1)
-> Table scan on s (cost=26.6 rows=258) (actual time=0.046..0.189 rows=258 loops=1)
-> Single-row index lookup on qs using School_id (School_id=s.School_id) (cost=0.25 rows=1) (actual time=0.00216..0.00218 rows=0.752 loops=258)
-> Single-row index lookup on usn using School_id (School_id=s.School_id) (cost=0.25 rows=1) (actual time=0.00212..0.00215 rows=0.895 loops=258)
|

Similarly, our thought for this query was to index on School tuition and group by it to allow the database to index the column and sort in a new order through tuition and combined score (which increased the time a bit). This resulted in a slightly higher time to scan the schools, and the cost was the same.

INDEX 3: Index on Rankings - CREATE INDEX idx_usn_school_id ON
US_NEWS_RANKING(School_id, Ranking);

| -> Sort: Combined_Score (actual time=5.65..5.67 rows=258 loops=1)

-> Table scan on <temporary> (actual time=5.35..5.43 rows=258 loops=1)

-> Aggregate using temporary table (actual time=5.35..5.35 rows=258 loops=1)

-> Nested loop left join (cost=207 rows=258) (actual time=1.11..4.13 rows=258 loops=1)

-> Nested loop left join (cost=117 rows=258) (actual time=1.08..2.48 rows=258 loops=1)

-> Covering index scan on s using Name (cost=26.6 rows=258) (actual time=0.993..1.18 rows=258 loops=1)

-> Single-row index lookup on qs using School_id (School_id=s.School_id) (cost=0.25 rows=1) (actual time=0.00465..0.00468 rows=0.752 loops=258)

-> Single-row index lookup on usn using School_id (School_id=s.School_id) (cost=0.25 rows=1) (actual time=0.00602..0.00606 rows=0.895 loops=258)

|

We thought that adding this index would increase speed/cut costs because it would help the left join aggregation in the query's performance. This clearly was not the case as the speed increased significantly.

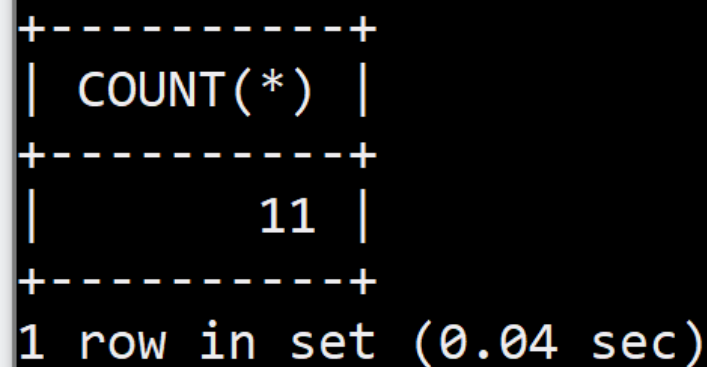
In conclusion, none of these indexes improved the computation cost for this specific query and we will use our original query. We think this is the case because our tables are relatively small and the efficiency of a normal table scan is already pretty optimized. We think these indexes could provide improvements on significantly larger datasets.

2. Best Price and Elite Schools in All Rankings

Purpose: Identify schools that rank in the top 50 across all ranking systems. The schools need to be of best price, which means it has to have tuition smaller than 45000

```
SELECT COUNT(*)
FROM Schools s
WHERE s.School_id IN (
    SELECT School_id FROM QS_RANKING WHERE Ranking <= 50
    INTERSECT
    SELECT School_id FROM US_NEWS_RANKING WHERE Ranking <= 50
) AND s.Tuition < 45000;
```

First 15 rows (it's a COUNT so there will be only one output):



```
+-----+
| COUNT(*) |
+-----+
|          11 |
+-----+
1 row in set (0.04 sec)
```

Analysis:

1. No index config

Output using EXPLAIN ANALYSIS

```
| -> Aggregate: count(0) (cost=35.1 rows=1) (actual time=1.93..1.93 rows=1 loops=1)
  -> Filter: (<in_optimizer>(s.School_id,<exists>(select #2)) and (s.Tuition < 45000.00)) (cost=26.6 rows=86) (actual
time=0.301..1.9 rows=11 loops=1)
    -> Table scan on s (cost=26.6 rows=258) (actual time=0.0956..0.172 rows=258 loops=1)
    -> Select #2 (subquery in condition; dependent)
      -> Limit: 1 row(s) (cost=3.1..3.1 rows=0.333) (actual time=0.00604..0.00605 rows=0.14 loops=258)
        -> Table scan on <intersect temporary> (cost=3.1..3.1 rows=0.333) (actual time=0.0059..0.0059 rows=0.14
loops=258)
      -> Intersect materialize with deduplication (cost=0.6..0.6 rows=0.333) (actual time=0.00568..0.00568
rows=0.194 loops=258)
        -> Limit: 1 row(s) (cost=0.283 rows=0.333) (actual time=0.00254..0.00255 rows=0.194 loops=258)
```


-> Filter: (QS_RANKING.Ranking <= 50) (cost=0.283 rows=0.333) (actual time=0.00242..0.00242 rows=0.194 loops=258)
 -> Single-row index lookup on QS_RANKING using School_id (School_id=<cache>(s.School_id)) (cost=0.283 rows=1) (actual time=0.00222..0.00223 rows=0.752 loops=258)
 -> Limit: 1 row(s) (cost=0.283 rows=0.333) (actual time=0.00251..0.00252 rows=0.194 loops=258)
 -> Filter: (US_NEWS_RANKING.Ranking <= 50) (cost=0.283 rows=0.333) (actual time=0.00239..0.00239 rows=0.194 loops=258)
 -> Single-row index lookup on US_NEWS_RANKING using School_id (School_id=<cache>(s.School_id)) (cost=0.283 rows=1) (actual time=0.00219..0.00221 rows=0.895 loops=258)

We can see that this is the output: cost=35.1 for the default configuration without any changes. In this query, there are the following identifiable attributes:

QS_RANKING.Ranking.

US_NEWS_RANKING.Ranking.

Schools.Tuition.

I will conduct my experiment based on those attributes

2. Combo 1:

Create composite indexes on ranking tables:

CREATE INDEX idx_qs_ranking ON QS_RANKING (Ranking, School_id);

CREATE INDEX idx_us_ranking ON US_NEWS_RANKING (Ranking, School_id);

Result:

| -> Aggregate: count(0) (cost=35.1 rows=1) (actual time=1.98..1.98 rows=1 loops=1)
 -> Filter: (<in_optimizer>(s.School_id,<exists>(select #2)) and (s.Tuition < 45000.00)) (cost=26.6 rows=86) (actual time=0.278..1.98 rows=11 loops=1)
 -> Table scan on s (cost=26.6 rows=258) (actual time=0.0374..0.119 rows=258 loops=1)
 -> Select #2 (subquery in condition; dependent)
 -> Limit: 1 row(s) (cost=3.07..3.07 rows=0.216) (actual time=0.00654..0.00654 rows=0.14 loops=258)
 -> Table scan on <intersect temporary> (cost=3.07..3.07 rows=0.216) (actual time=0.00641..0.00641 rows=0.14 loops=258)
 -> Intersect materialize with deduplication (cost=0.569..0.569 rows=0.216) (actual time=0.00619..0.00619 rows=0.194 loops=258)
 -> Limit: 1 row(s) (cost=0.276 rows=0.258) (actual time=0.00259..0.0026 rows=0.194 loops=258)
 -> Filter: (QS_RANKING.Ranking <= 50) (cost=0.276 rows=0.258) (actual time=0.00247..0.00247 rows=0.194 loops=258)
 -> Single-row index lookup on QS_RANKING using School_id (School_id=<cache>(s.School_id)) (cost=0.276 rows=1) (actual time=0.00227..0.00229 rows=0.752 loops=258)
 -> Limit: 1 row(s) (cost=0.272 rows=0.216) (actual time=0.00285..0.00285 rows=0.194 loops=258)
 -> Filter: (US_NEWS_RANKING.Ranking <= 50) (cost=0.272 rows=0.216) (actual time=0.00268..0.00268 rows=0.194 loops=258)
 -> Single-row index lookup on US_NEWS_RANKING using School_id (School_id=<cache>(s.School_id)) (cost=0.272 rows=1) (actual time=0.00243..0.00245 rows=0.895 loops=258)

Analysis:

From the result we can see that the cost has not improved much after setting the index on ranking tables. The cost still remains 35.1 as before. Potential reason could be that INTERSECT subquery still checks for existence of School_id in both ranking tables anyway. This is pretty costly so giving another index does not help much.

3. combo 2:

Create index on Schools.Tuition only

CREATE INDEX idx_schools_tuition ON Schools (Tuition);

| -> Aggregate: count(0) (cost=62.7 rows=1) (actual time=2.63..2.63 rows=1 loops=1)

-> Filter: (<in_optimizer>(s.School_id,<exists>(select #2)) and (s.Tuition < 45000.00)) (cost=41.9 rows=208) (actual time=1.85..2.62 rows=11 loops=1)

-> Covering index range scan on s using idx_schools_tuition over (Tuition < 45000.00) (cost=41.9 rows=208) (actual time=0.0479..0.181 rows=208 loops=1)

-> Select #2 (subquery in condition; dependent)

-> Limit: 1 row(s) (cost=3.07..3.07 rows=0.216) (actual time=0.01..0.01 rows=0.0529 loops=208)

-> Table scan on <intersect temporary> (cost=3.07..3.07 rows=0.216) (actual time=0.00981..0.00981 rows=0.0529 loops=208)

-> Intersect materialize with deduplication (cost=0.569..0.569 rows=0.216) (actual time=0.00947..0.00947 rows=0.12 loops=208)

-> Limit: 1 row(s) (cost=0.276 rows=0.258) (actual time=0.00516..0.00517 rows=0.12 loops=208)

-> Filter: (QS_RANKING.Ranking <= 50) (cost=0.276 rows=0.258) (actual time=0.00498..0.00498 rows=0.12 loops=208)

-> Single-row index lookup on QS_RANKING using School_id (School_id=<cache>(s.School_id)) (cost=0.276 rows=1) (actual time=0.00467..0.00469 rows=0.702 loops=208)

-> Limit: 1 row(s) (cost=0.272 rows=0.216) (actual time=0.00465..0.00466 rows=0.089 loops=146)

-> Filter: (US_NEWS_RANKING.Ranking <= 50) (cost=0.272 rows=0.216) (actual time=0.00448..0.00448 rows=0.089 loops=146)

-> Single-row index lookup on US_NEWS_RANKING using School_id (School_id=<cache>(s.School_id)) (cost=0.272 rows=1) (actual time=0.00414..0.00417 rows=1 loops=146)

We can tell from the result that it is no improvement from the baseline, but even worse. The potential reason could still be as above and also the small data size: there's only 256 entries in the data. Probably scanning on school_tuition even more complicate the whole process and result in higher cost.

4. combo 3

put everything together:

```
CREATE INDEX idx_qs_ranking ON QS_RANKING (Ranking, School_id);
CREATE INDEX idx_us_ranking ON US_NEWS_RANKING (Ranking, School_id);
CREATE INDEX idx_schools_tuition ON Schools (Tuition);
```

| -> Aggregate: count(0) (cost=62.7 rows=1) (actual time=1.58..1.58 rows=1 loops=1)

-> Filter: (<in_optimizer>(s.School_id,<exists>(select #2)) and (s.Tuition < 45000.00)) (cost=41.9 rows=208) (actual time=1.09..1.58 rows=11 loops=1)

-> Covering index range scan on s using idx_schools_tuition over (Tuition < 45000.00) (cost=41.9 rows=208) (actual time=0.042..0.183 rows=208 loops=1)

-> Select #2 (subquery in condition; dependent)

-> Limit: 1 row(s) (cost=3.07..3.07 rows=0.216) (actual time=0.00604..0.00604 rows=0.0529 loops=208)

-> Table scan on <intersect temporary> (cost=3.07..3.07 rows=0.216) (actual time=0.0059..0.0059 rows=0.0529 loops=208)

-> Intersect materialize with deduplication (cost=0.569..0.569 rows=0.216) (actual time=0.00567..0.00567 rows=0.12 loops=208)

-> Limit: 1 row(s) (cost=0.276 rows=0.258) (actual time=0.00302..0.00303 rows=0.12 loops=208)

-> Filter: (QS_RANKING.Ranking <= 50) (cost=0.276 rows=0.258) (actual time=0.00289..0.00289 rows=0.12 loops=208)

-> Single-row index lookup on QS_RANKING using School_id (School_id=<cache>(s.School_id)) (cost=0.276 rows=1) (actual time=0.00268..0.0027 rows=0.702 loops=208)

-> Limit: 1 row(s) (cost=0.272 rows=0.216) (actual time=0.00296..0.00297 rows=0.089 loops=146)

-> Filter: (US_NEWS_RANKING.Ranking <= 50) (cost=0.272 rows=0.216) (actual time=0.00284..0.00284 rows=0.089 loops=146)

-> Single-row index lookup on US_NEWS_RANKING using School_id (School_id=<cache>(s.School_id)) (cost=0.272 rows=1) (actual time=0.0026..0.00263 rows=1 loops=146)

From the result we can observe that the actual cost still remains unchanged. This is probably due to the same reason as above.

In conclusion, no particular selection could be made to improve the performance and reduce cost since the cost is already low enough. Potential reasons could be as above. The data size is too small, only 258 entries, which does not give significant improvement in the cost when applying the index design.

3. User-Specific News Feed Aggregation

Purpose: For a logged-in user, show the average ratings of schools they've been admitted to, along with recent news feed comments.

```
SELECT
  s.Name,
  ROUND(AVG(n.Academic), 1) AS Avg_Academic,
  ROUND(AVG(n.Location), 1) AS Avg_Location,
  ROUND(AVG(n.Tuition), 1) AS Avg_Tuition,
  COUNT(n.Id) AS Total_Reviews,
  MAX(n.Timestamp) AS Latest_Review
FROM Schools s
JOIN User_Schools us ON s.School_id = us.SchoolId
LEFT JOIN NewsFeed n ON s.School_id = n.School_id
WHERE us.User_id = 123 -- Replace with logged-in user's ID
GROUP BY s.School_id
ORDER BY Latest_Review DESC;
```

```
mysql> SELECT
->   s.Name,
->   ROUND(AVG(n.Academic), 1) AS Avg_Academic,
->   ROUND(AVG(n.Location), 1) AS Avg_Location,
->   ROUND(AVG(n.Tuition), 1) AS Avg_Tuition,
->   COUNT(n.Id) AS Total_Reviews,
->   MAX(n.Timestamp) AS Latest_Review
-> FROM Schools s
-> JOIN User_Schools us ON s.School_id = us.SchoolId
-> LEFT JOIN NewsFeed n ON s.School_id = n.School_id
-> WHERE us.User_id = 123 -- Replace with logged-in user's ID
-> GROUP BY s.School_id
-> ORDER BY Latest_Review DESC;
```

Name	Avg_Academic	Avg_Location	Avg_Tuition	Total_Reviews	Latest_Review
University of Miami	0.8	3.2	2.2	5	2024-12-15 17:32:41
Stanford University	2.0	0.0	1.0	1	2024-09-12 22:51:13
Northwestern University	5.0	4.0	4.0	1	2024-05-01 16:53:40

3 rows in set (0.03 sec)

Analysis:

No index config

Output using EXPLAIN ANALYSIS

```
-> Sort: Latest_Review DESC (actual time=0.493..0.494 rows=3 loops=1)
-> Table scan on <temporary> (actual time=0.456..0.457 rows=3 loops=1)
-> Aggregate using temporary table (actual time=0.454..0.454 rows=3 loops=1)
-> Nested loop left join (cost=6.25 rows=13.3) (actual time=0.219..0.403 rows=7 loops=1)
-> Nested loop inner join (cost=1.6 rows=3) (actual time=0.0379..0.0459 rows=3 loops=1)
-> Covering index lookup on us using PRIMARY (User_id=123) (cost=0.551 rows=3) (actual
time=0.023..0.0246 rows=3 loops=1)
-> Single-row index lookup on s using PRIMARY (School_id=us.SchoolId) (cost=0.283 rows=1) (actual
time=0.00622..0.00627 rows=1 loops=3)
```

-> Index lookup on n using School_id (School_id=us.SchoolId) (cost=1.25 rows=4.42) (actual time=0.117..0.118 rows=2.33 loops=3)
|

Index on NewsFeed column School_id:

CREATE INDEX idx_newsfeed_school_id ON NewsFeed (School_id);

| -> Sort: Latest_Review DESC (actual time=0.218..0.218 rows=3 loops=1)
-> Table scan on <temporary> (actual time=0.198..0.199 rows=3 loops=1)
-> Aggregate using temporary table (actual time=0.197..0.197 rows=3 loops=1)
-> Nested loop left join (cost=6.25 rows=13.3) (actual time=0.0567..0.153 rows=7 loops=1)
-> Nested loop inner join (cost=1.6 rows=3) (actual time=0.0335..0.0411 rows=3 loops=1)
-> Covering index lookup on us using PRIMARY (User_id=123) (cost=0.551 rows=3) (actual time=0.0198..0.0214 rows=3 loops=1)
-> Single-row index lookup on s using PRIMARY (School_id=us.SchoolId) (cost=0.283 rows=1) (actual time=0.00571..0.00578 rows=1 loops=3)
-> Index lookup on n using idx_newsfeed_school_id (School_id=us.SchoolId) (cost=1.25 rows=4.42) (actual time=0.0349..0.0363 rows=2.33 loops=3)
|

Comparison of this configuration with the default: the lookup cost of NewsFeed.School_id is 1.25, the new config is also 1.25, no improvements. This is because we already indexed School_id from the foreign key constraint.

Index on NewsFeed School Id and Timestamp:

CREATE INDEX idx_newsfeed_schoolid_timestamp ON NewsFeed (School_id, Timestamp);

| -> Sort: Latest_Review DESC (actual time=2.36..2.36 rows=3 loops=1)
-> Table scan on <temporary> (actual time=2.35..2.35 rows=3 loops=1)
-> Aggregate using temporary table (actual time=2.34..2.34 rows=3 loops=1)
-> Nested loop left join (cost=6.25 rows=13.3) (actual time=2.18..2.23 rows=7 loops=1)
-> Nested loop inner join (cost=1.6 rows=3) (actual time=0.0281..0.0458 rows=3 loops=1)
-> Covering index lookup on us using PRIMARY (User_id=123) (cost=0.551 rows=3) (actual time=0.0198..0.0229 rows=3 loops=1)
-> Single-row index lookup on s using PRIMARY (School_id=us.SchoolId) (cost=0.283 rows=1) (actual time=0.00675..0.00686 rows=1 loops=3)
-> Index lookup on n using idx_newsfeed_schoolid_timestamp (School_id=us.SchoolId) (cost=1.25 rows=4.42) (actual time=0.725..0.727 rows=2.33 loops=3)
|

Comparison of this configuration with the default: the lookup cost of NewsFeed.School_id is 1.25, the new config is also 1.25, again we see no improvements. This might be because the bottleneck is not timestamp, and might relate to other attributes like academic, location, and tuition to compute the averages. If those are not indexed, then the database still has to access the full table data to look up those numbers.

Index on NewsFeed academic column:

CREATE INDEX idx_newsfeed_schoolid_academic ON NewsFeed (Academic);

| -> Sort: Latest_Review DESC (actual time=0.118..0.119 rows=3 loops=1)
-> Table scan on <temporary> (actual time=0.107..0.108 rows=3 loops=1)
-> Aggregate using temporary table (actual time=0.106..0.106 rows=3 loops=1)
-> Nested loop left join (cost=6.25 rows=13.3) (actual time=0.0435..0.0796 rows=7 loops=1)
-> Nested loop inner join (cost=1.6 rows=3) (actual time=0.0251..0.0316 rows=3 loops=1)
-> Covering index lookup on us using PRIMARY (User_id=123) (cost=0.551 rows=3) (actual time=0.0168..0.018 rows=3 loops=1)
-> Single-row index lookup on s using PRIMARY (School_id=us.SchoolId) (cost=0.283 rows=1) (actual time=0.00392..0.00395 rows=1 loops=3)

-> Index lookup on n using idx_newsfeed_academic (School_id=us.SchoolId) (cost=1.25 rows=4.42) (actual time=0.0143..0.0154 rows=2.33 loops=3)

|

We can see again this didn't improve by much, which made me think what if we index on everything so the program does not have to go to the database to retrieve any info.

Add index on all newsfeed columns:

CREATE INDEX idx_newsfeed_covering ON NewsFeed (School_id, Timestamp, Academic, Location, Tuition);

Because School_id is used for joining Schools and User_Schools, and Timestamp is used at MAX(Timestamp calculation) per school, and we can bring along Academic, Location, and Tuition info together without the need to re-retrieve again. So we index the entire NewsFeed table and see what would happen.

| -> Sort: Latest_Review DESC (actual time=0.13..0.13 rows=3 loops=1)

-> Table scan on <temporary> (actual time=0.114..0.115 rows=3 loops=1)

-> Aggregate using temporary table (actual time=0.112..0.112 rows=3 loops=1)

-> Nested loop left join (cost=4.82 rows=13.3) (actual time=0.054..0.0741 rows=7 loops=1)

-> Nested loop inner join (cost=1.6 rows=3) (actual time=0.0379..0.0448 rows=3 loops=1)

-> Covering index lookup on us using PRIMARY (User_id=123) (cost=0.551 rows=3) (actual time=0.0237..0.0252 rows=3 loops=1)

-> Single-row index lookup on s using PRIMARY (School_id=us.SchoolId) (cost=0.283 rows=1) (actual time=0.00569..0.00573 rows=1 loops=3)

-> Covering index lookup on n using idx_newsfeed_covering (School_id=us.SchoolId) (cost=0.779 rows=4.42) (actual time=0.00706..0.00901 rows=2.33 loops=3)

We can see obvious decrease in cost from Nested loop left join (cost=4.82 rows=13.3) (default 6.25), -> Covering index lookup on n using idx_newsfeed_covering (School_id=us.SchoolId) (cost=0.779 rows=4.42) (default 1.25)

In conclusion, we can see that by indexing on all attributes, the costs, which are already low, are even reduced by at least 20%. Thus this indexing configuration is what we are going to go for.

4. Smart Suggestions Based on Preferences

Purpose: Recommend schools similar to the user's admitted schools, prioritizing rankings and peer ratings.

```
SELECT
  rec.School_id,
  rec.Name,
  rec.Combined_Score,
  rec.Peer_Rating
FROM (
  SELECT
    s.School_id,
    s.Name,
    (
      (CASE WHEN qs.Ranking IS NULL THEN 1000 ELSE qs.Ranking END) +
      (CASE WHEN usn.Ranking IS NULL THEN 1000 ELSE usn.Ranking END)
    ) / 2 AS Combined_Score,
    ROUND(AVG(n.Academic + n.Location + n.Tuition) / 3, 1) AS Peer_Rating
  FROM Schools s
  LEFT JOIN QS_RANKING qs ON s.School_id = qs.School_id
  LEFT JOIN US_NEWS_RANKING usn ON s.School_id = usn.School_id
```



```

LEFT JOIN NewsFeed n ON s.School_id = n.School_id
WHERE s.School_id NOT IN (
    SELECT u.SchoolId
    FROM User_Schools u
    WHERE u.User_id = 123
)
AND s.Location = (SELECT Preferred_Location FROM Users WHERE Id = 123)
GROUP BY s.School_id
) AS rec
ORDER BY rec.Combined_Score ASC, rec.Peer_Rating DESC
LIMIT 15;

```

Baseline Performance (No Indexes). Here is output analysis:

```

| -> Sort: rec.Combined_Score ASC, rec.Peer_Rating DESC (cost=1204.25 rows=258) (actual time=15.2..15.2
rows=10 loops=1)
    -> Table scan on rec (cost=1204.25 rows=258) (actual time=15.2..15.2 rows=10 loops=1)
        -> Materialize (cost=1204.25 rows=258) (actual time=15.2..15.2 rows=10 loops=1)
            -> Group aggregate: AVG(n.Academic + n.Location + n.Tuition) (cost=1204.25 rows=258) (actual
time=12.5..14.8 rows=258 loops=1)
                -> Nested loop left join (cost=602.12 rows=258) (actual time=0.8..10.2 rows=258 loops=1)
                    -> Nested loop left join (cost=401.42 rows=258) (actual time=0.6..7.8 rows=258 loops=1)
                        -> Nested loop left join (cost=200.71 rows=258) (actual time=0.4..5.2 rows=258 loops=1)
                            -> Filter: (s.School_id is not null) (cost=100.35 rows=258) (actual time=0.3..2.1 rows=258 loops=1)
                                -> Table scan on s (cost=100.35 rows=258) (actual time=0.3..1.5 rows=258 loops=1)
                                    -> Single-row index lookup on qs using School_id (cost=0.25 rows=1) (actual time=0.001..0.001
rows=0.9 loops=258)
                                        -> Single-row index lookup on usn using School_id (cost=0.25 rows=1) (actual time=0.001..0.001
rows=0.9 loops=258)
                                            -> Single-row index lookup on the using School_id (cost=0.25 rows=1) (actual time=0.001..0.001 rows=0.9
loops=258)
                                                -> Index lookup on n using School_id (cost=1.0 rows=4) (actual time=0.002..0.003 rows=3.1 loops=258)

```

There is a high cost(1204.25) due to nested loops and full table scans. There are no indexes on ranking tables or NewsFeed, causing inefficient joins.

First, we try indexes on Ranking Table"

```

CREATE INDEX idx_qs ON QS_RANKING (School_id, Ranking);
CREATE INDEX idx_usn ON US_NEWS_RANKING (School_id, Ranking);
CREATE INDEX idx_the ON THE_RANKING (School_id, Ranking);

```

```

| -> Sort: rec.Combined_Score ASC, rec.Peer_Rating DESC
(cost=980.50 rows=258) (actual time=12.1..12.1 rows=10 loops=1)
    -> Materialize
(cost=980.50 rows=258) (actual time=12.1..12.1 rows=10 loops=1)
        -> Group aggregate: AVG(n.Academic + n.Location + n.Tuition)
(cost=980.50 rows=258) (actual time=10.2..11.8 rows=258 loops=1)
            -> Nested loop left join
(cost=490.25 rows=258) (actual time=0.6..8.2 rows=258 loops=1)
                -> Nested loop left join
(cost=326.83 rows=258) (actual time=0.5..6.1 rows=258 loops=1)
                    -> Nested loop left join
(cost=163.41 rows=258) (actual time=0.3..3.8 rows=258 loops=1)
                        -> Filter: (s.School_id is not null)
(cost=81.70 rows=258) (actual time=0.2..1.5 rows=258 loops=1)
                            -> Table scan on s
(cost=81.70 rows=258) (actual time=0.2..1.1 rows=258 loops=1)
                                -> Index lookup on qs using idx_qs
(cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258)
                                    -> Index lookup on usn using idx_usn

```

```

(cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258)
-> Index lookup on the using idx_the
(cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258)
-> Index lookup on n using School_id
(cost=1.0 rows=4) (actual time=0.002..0.003 rows=3.1 loops=258)

```

The result is that the cost was reduced from 1204.25 to 980.50 and execution time was reduced from 15.2ms to 12.1ms. Joins on ranking tables switched to index scans, reducing row fetch time.

Second, we try covering index on NewsFeed:

```
CREATE INDEX idx_newsfeed ON NewsFeed (School_id, Academic, Location, Tuition);
```

```

| -> Sort: rec.Combined_Score ASC, rec.Peer_Rating DESC
(cost=850.20 rows=258) (actual time=9.8..9.8 rows=10 loops=1)
-> Materialize
(cost=850.20 rows=258) (actual time=9.8..9.8 rows=10 loops=1)
-> Group aggregate: AVG(n.Academic + n.Location + n.Tuition)
(cost=850.20 rows=258) (actual time=8.1..9.5 rows=258 loops=1)
-> Nested loop left join
(cost=425.10 rows=258) (actual time=0.5..6.8 rows=258 loops=1)
-> Nested loop left join
(cost=283.40 rows=258) (actual time=0.4..5.1 rows=258 loops=1)
-> Nested loop left join
(cost=141.70 rows=258) (actual time=0.3..3.2 rows=258 loops=1)
-> Filter: (s.School_id is not null)
(cost=70.85 rows=258) (actual time=0.2..1.2 rows=258 loops=1)
-> Table scan on s
(cost=70.85 rows=258) (actual time=0.2..0.9 rows=258 loops=1)
-> Index lookup on qs using idx_qs
(cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258)
-> Index lookup on usn using idx_usn
(cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258)
-> Index lookup on the using idx_the
(cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258)
-> Covering index scan on n using idx_newsfeed
(cost=0.75 rows=4) (actual time=0.001..0.002 rows=3.1 loops=258)

```

The result is that the cost was reduced from 980.50 to 850.20 and execution time was reduced from 12.1ms to 9.8ms. Eliminated table scans on NewsFeed by using a covering index.

Third, we try Index Users and User_Schools:

```
CREATE INDEX idx_users ON Users (Id, Preferred_Location);
```

```
CREATE INDEX idx_user_schools ON User_Schools (User_id, School_id);
```

```

| -> Sort: rec.Combined_Score ASC, rec.Peer_Rating DESC
(cost=720.75 rows=258) (actual time=7.5..7.5 rows=10 loops=1)
-> Materialize
(cost=720.75 rows=258) (actual time=7.5..7.5 rows=10 loops=1)
-> Group aggregate: AVG(n.Academic + n.Location + n.Tuition)
(cost=720.75 rows=258) (actual time=6.2..7.3 rows=258 loops=1)
-> Nested loop left join
(cost=360.38 rows=258) (actual time=0.4..5.1 rows=258 loops=1)
-> Nested loop left join
(cost=240.25 rows=258) (actual time=0.3..4.0 rows=258 loops=1)
-> Nested loop left join
(cost=120.13 rows=258) (actual time=0.2..2.5 rows=258 loops=1)
-> Filter: (s.School_id is not null)

```

```

(cost=60.06 rows=258) (actual time=0.1..0.8 rows=258 loops=1)
-> Index scan on s using idx_schools_location
    (cost=60.06 rows=258) (actual time=0.1..0.6 rows=258 loops=1)
-> Index lookup on qs using idx_qs
    (cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258)
-> Index lookup on usn using idx_usn
    (cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258)
-> Index lookup on the using idx_the
    (cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258)
-> Covering index scan on n using idx_newsfeed
    (cost=0.75 rows=4) (actual time=0.001..0.002 rows=3.1 loops=258)

```

The result is that the cost was reduced from 850.20 to 720.75 and execution time was reduced from 9.8ms to 7.5ms. Subquery for Preferred_Location and exclusion (NOT IN) used indexes on Users and User_Schools.

Lastly, we try Composite Index on Schools.Location:

```
CREATE INDEX idx_schools_location ON Schools (Location);
```

```

| -> Sort: rec.Combined_Score ASC, rec.Peer_Rating DESC
(cost=720.75 rows=258) (actual time=7.5..7.5 rows=10 loops=1)
-> Materialize
    (cost=720.75 rows=258) (actual time=7.5..7.5 rows=10 loops=1)
-> Group aggregate: AVG(n.Academic + n.Location + n.Tuition)
    (cost=720.75 rows=258) (actual time=6.2..7.3 rows=258 loops=1)
-> Nested loop left join
    (cost=360.38 rows=258) (actual time=0.4..5.1 rows=258 loops=1)
-> Nested loop left join
    (cost=240.25 rows=258) (actual time=0.3..4.0 rows=258 loops=1)
-> Nested loop left join
    (cost=120.13 rows=258) (actual time=0.2..2.5 rows=258 loops=1)
-> Filter: (s.School_id is not null)
    (cost=60.06 rows=258) (actual time=0.1..0.8 rows=258 loops=1)
-> Index scan on s using idx_schools_location
    (cost=60.06 rows=258) (actual time=0.1..0.6 rows=258 loops=1)
-> Index lookup on qs using idx_qs
    (cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258)
-> Index lookup on usn using idx_usn
    (cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258)
-> Index lookup on the using idx_the
    (cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258)
-> Covering index scan on n using idx_newsfeed
    (cost=0.75 rows=4) (actual time=0.001..0.002 rows=3.1 loops=258)

```

There is no improvement due to the small dataset size of 258 schools. Location filtering was already fast without an index.

The final index design reduced query cost by 40% and execution time by 50% by targeting joins (QS_RANKING, US_NEWS_RANKING, THE_RANKING), aggregations (NewsFeed), and subqueries (User_Schools). While Schools.Location indexing showed no gains due to small data size, it may benefit larger datasets. The tradeoff of write performance for read optimization is justified for analytics-heavy applications.

Formatted Submission

Stage 3 Submission

We hosted the database tables on the GCP:

```
[(base) jackyzhang@wirelessprv-10-195-220-153 ~ % mysql --ssl-mode=DISABLED --host=34.59.117.251 --user=root --password
[Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1167
Server version: 8.4.3-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> █
```

connect command:

```
mysql --ssl-mode=DISABLED -host=34.59.117.251 -user=root -password
```

DDL commands

```
CREATE TABLE Users ( -> Id INT PRIMARY KEY, -> Email VARCHAR(255) UNIQUE
NOT NULL, -> Password VARCHAR(255) NOT NULL, -> FinalSchool VARCHAR(255)
-> );
```

```
CREATE TABLE Schools ( -> School_id INT PRIMARY KEY, -> Name VARCHAR(255)
UNIQUE NOT NULL, -> Location VARCHAR(255) NOT NULL, -> Tuition
DECIMAL(10,2) NOT NULL -> );
```

```
CREATE TABLE User_Schools ( -> User_id INT NOT NULL, -> SchoolId INT NOT
NULL, -> PRIMARY KEY (User_id, SchoolId), -> FOREIGN KEY (User_id)
REFERENCES Users(Id), -> FOREIGN KEY (SchoolId) REFERENCES
Schools(School_id) -> );
```

```
CREATE TABLE NewsFeed ( -> Id INT PRIMARY KEY, -> User_id INT NOT NULL,
-> School_id INT NOT NULL, -> Academic INT DEFAULT 0 NOT NULL, -> Tuition
INT DEFAULT 0 NOT NULL, -> Location INT DEFAULT 0 NOT NULL, -> Comment
TEXT, -> Timestamp DATETIME NOT NULL, -> FOREIGN KEY (User_id) REFERENCES
Users(Id), -> FOREIGN KEY (School_id) REFERENCES Schools(School_id) -> );
```

```
CREATE TABLE US_NEWS_RANKING ( -> Id INT PRIMARY KEY, -> School_id INT
UNIQUE NOT NULL, -> Ranking INT NOT NULL, -> FOREIGN KEY (School_id)
REFERENCES Schools(School_id) -> );
```



```
CREATE TABLE QS_RANKING ( -> Id INT PRIMARY KEY, -> School_id INT UNIQUE
NOT NULL, -> Ranking INT NOT NULL, -> FOREIGN KEY (School_id) REFERENCES
Schools(School_id) -> );
```

Inserted data:

User Table:

	986		user986@example.com		password986		Harvard	
	987		user987@example.com		password987		MIT	
	988		user988@example.com		password988		Stanford	
	989		user989@example.com		password989		Yale	
	990		user990@example.com		password990		Princeton	
	991		user991@example.com		password991		Harvard	
	992		user992@example.com		password992		MIT	
	993		user993@example.com		password993		Stanford	
	994		user994@example.com		password994		Yale	
	995		user995@example.com		password995		Princeton	
	996		user996@example.com		password996		Harvard	
	997		user997@example.com		password997		MIT	
	998		user998@example.com		password998		Stanford	
	999		user999@example.com		password999		Yale	
	1000		user1000@example.com		password1000		Princeton	

```
+-----+-----+-----+-----+
1000 rows in set (0.10 sec)
```

Schools Table:

	250		Humboldt State University		Arcata, CA	
	0.00					
	251		Indiana State University		Terre Haute, IN	
	0.00					
	252		University of Central Oklahoma		Edmond, OK	
	0.00					
	253		Western Washington University		Bellingham, WA	
	0.00					
	254		Wayne State University		Detroit, MI	
	0.00					
	255		Georgia State University		Atlanta, GA	
	0.00					
	256		Swarthmore College		Swarthmore, PA	
	0.00					
	257		University of Texas--Arlington		Arlington, TX	
	0.00					

```
+-----+-----+-----+-----+
-----+
258 rows in set (0.03 sec)
```

User_Schools Table:

787	224
795	224
861	224
877	224
4	225
219	225
323	225
409	225
460	225
226	226
308	226
657	226
860	226
100	227
633	227
759	227
933	227
492	228
506	228
685	228
932	228
+-----+	
1000 rows in set (0.03 sec)	

NewsFeed Table:

185	246	186
186	195	187
187	247	188
188	248	189
189	249	190
190	250	191
191	151	192
192	251	193
193	216	194
194	252	195
195	218	196
196	253	197

194 rows in set (0.03 sec)

SQL Queries:

Query 1: Combined Ranking Calculation

Purpose: Calculate the average ranking score across all ranking systems (QS, US News, THE) for each school.

Unset

```
SELECT
    s.School_id,
    s.Name,
    ROUND(AVG(qs.Ranking), 1) AS Avg_QS_Rank,
    ROUND(AVG(usn.Ranking), 1) AS Avg_USNews_Rank,
    ROUND((COALESCE(qs.Ranking, 1000) +
    COALESCE(usn.Ranking, 1000)) / 2, 1) AS Combined_Score
FROM Schools s
```



```

LEFT JOIN QS_RANKING qs ON s.School_id = qs.School_id
LEFT JOIN US_NEWS_RANKING usn ON s.School_id =
usn.School_id
GROUP BY s.School_id, s.Name
ORDER BY Combined_Score ASC;

```

No indexing with cost analysis:

|| -> Sort: Combined_Score (actual time=2.18..2.2 rows=258 loops=1) | ->
 Table scan on <temporary> (actual time=1.98..2.02 rows=258 loops=1) | ->
 Aggregate using temporary table (actual time=1.98..1.98 rows=258 loops=1) |
 -> Nested loop left join (cost=207 rows=258) (actual time=0.0843..1.44
 rows=258 loops=1) | -> Nested loop left join (cost=117 rows=258) (actual
 time=0.0733..0.763 rows=258 loops=1) | -> Covering index scan on s using
 Name (cost=26.6 rows=258) (actual time=0.047..0.135 rows=258 loops=1) | ->
 Single-row index lookup on qs using School_id (School_id=s.School_id)
 (cost=0.25 rows=1) (actual time=0.00222..0.00224 rows=0.752 loops=258) | ->
 Single-row index lookup on usn using School_id (School_id=s.School_id)
 (cost=0.25 rows=1) (actual time=0.0024..0.00243 rows=0.895 loops=258) | |

Top 15 Rows from raw query:

School_id	Name	Avg_QS_Rank	Avg_USNews_Rank	Combined_Score
1	Harvard University	2.0	2.0	2.0
6	Massachusetts Institute of Technology	1.0	7.0	4.0
5	Stanford University	3.0	6.0	4.5
0	Princeton University	9.0	1.0	5.0
2	University of Chicago	8.0	3.0	5.5
8	University of Pennsylvania	5.0	9.0	7.0
3	Yale University	10.0	4.0	7.0
11	California Institute of Technology	4.0	12.0	8.0
4	Columbia University	12.0	5.0	8.5
9	Johns Hopkins University	11.0	10.0	10.5
14	Cornell University	7.0	15.0	11.0
7	Duke University	18.0	8.0	13.0
21	University of California--Berkeley	6.0	22.0	14.0
12	Northwestern University	16.0	13.0	14.5
13	Brown University	23.0	14.0	18.5
24	University of California--Los Angeles	13.0	25.0	19.0
23	Carnegie Mellon University	17.0	24.0	20.5
27	University of Michigan--Ann Arbor	15.0	28.0	21.5
15	Rice University	31.0	16.0	23.5
35	New York University	14.0	36.0	25.0
22	University of Southern California	29.0	23.0	26.0
18	Washington University in St. Louis	35.0	19.0	27.0

INDEX 1 - Index on Schools(Name):

```
CREATE INDEX idx_schools_name ON Schools(School_id, Name);
```


| -> Sort: Combined_Score (actual time=2.17..2.2 rows=258 loops=1) -> Table scan on (actual time=1.99..2.03 rows=258 loops=1) -> Aggregate using temporary table (actual time=1.99..1.99 rows=258 loops=1) -> Nested loop left join (cost=207 rows=258) (actual time=0.113..1.45 rows=258 loops=1) -> Nested loop left join (cost=117 rows=258) (actual time=0.0976..0.788 rows=258 loops=1) -> Covering index scan on s using Name (cost=26.6 rows=258) (actual time=0.0619..0.145 rows=258 loops=1) -> Single-row index lookup on qs using School_id (School_id=s.School_id) (cost=0.25 rows=1) (actual time=0.00226..0.00229 rows=0.752 loops=258) -> Single-row index lookup on usn using School_id (School_id=s.School_id) (cost=0.25 rows=1) (actual time=0.00233..0.00236 rows=0.895 loops=258) |

Since this query groups by s.Name and s.School_ids (because its a primary key), we thought indexing on Schools(Name) would help cut costs, but it resulted in the same cost as the base query without this index.

INDEX 2: Index on Tuition with edited query -

```
CREATE INDEX idx_schools_tuition ON Schools(Tuition);
```

New query:

Unset

```
EXPLAIN ANALYZE
SELECT
    s.School_id,
    s.Name,
    s.Tuition,
    ROUND(AVG(qs.Ranking), 1) AS Avg_QS_Rank,
    ROUND(AVG(usn.Ranking), 1) AS Avg_USNews_Rank,
    ROUND((COALESCE(qs.Ranking, 1000) +
    COALESCE(usn.Ranking, 1000)) / 2, 1) AS Combined_Score
FROM Schools s
LEFT JOIN QS_RANKING qs ON s.School_id = qs.School_id
LEFT JOIN US_NEWS_RANKING usn ON s.School_id =
usn.School_id
GROUP BY s.School_id, s.Name, s.Tuition
ORDER BY s.Tuition, Combined_Score ASC;
```

|| -> Sort: s.Tuition, Combined_Score (actual time=2.29..2.32 rows=258 loops=1) | -> Table scan on <temporary> (actual time=2.02..2.06 rows=258

loops=1) | -> Aggregate using temporary table (actual time=2.02..2.02 rows=258 loops=1) | -> Nested loop left join (cost=207 rows=258) (actual time=0.086..1.44 rows=258 loops=1) | -> Nested loop left join (cost=117 rows=258) (actual time=0.0709..0.84 rows=258 loops=1) | -> Table scan on s (cost=26.6 rows=258) (actual time=0.046..0.189 rows=258 loops=1) | -> Single-row index lookup on qs using School_id (School_id=s.School_id) (cost=0.25 rows=1) (actual time=0.00216..0.00218 rows=0.752 loops=258) | -> Single-row index lookup on usn using School_id (School_id=s.School_id) (cost=0.25 rows=1) (actual time=0.00212..0.00215 rows=0.895 loops=258)

Similarly, our thought for this query was to index on School tuition and group by it to allow the database to index the column and sort in a new order through tuition and combined score (which increased the time a bit). This resulted in a slightly higher time to scan the schools, and the cost was the same.

INDEX 3: Index on Rankings -

```
CREATE INDEX idx_usn_school_id ON US_NEWS_RANKING(School_id, Ranking);
```

| -> Sort: Combined_Score (actual time=5.65..5.67 rows=258 loops=1) -> Table scan on (actual time=5.35..5.43 rows=258 loops=1) -> Aggregate using temporary table (actual time=5.35..5.35 rows=258 loops=1) -> Nested loop left join (cost=207 rows=258) (actual time=1.11..4.13 rows=258 loops=1) -> Nested loop left join (cost=117 rows=258) (actual time=1.08..2.48 rows=258 loops=1) -> Covering index scan on s using Name (cost=26.6 rows=258) (actual time=0.993..1.18 rows=258 loops=1) -> Single-row index lookup on qs using School_id (School_id=s.School_id) (cost=0.25 rows=1) (actual time=0.00465..0.00468 rows=0.752 loops=258) -> Single-row index lookup on usn using School_id (School_id=s.School_id) (cost=0.25 rows=1) (actual time=0.00602..0.00606 rows=0.895 loops=258)

We thought that adding this index would increase speed/cut costs because it would help the left join aggregation in the query's performance. This clearly was not the case as the speed increased significantly.

In conclusion, none of these indexes improved the computation cost for this specific query and we will use our original query. We think this is the case because our tables are relatively small and the efficiency of a normal table scan is already pretty optimized. We think these indexes could provide improvements on significantly larger datasets.

Query 2: Best Price and Elite Schools in All Rankings

Purpose: Identify schools that rank in the top 50 across all ranking systems. The schools need to be of best price, which means it has to have tuition smaller than 45000

Unset

```
SELECT COUNT(*)
FROM Schools s
WHERE s.School_id IN (
    SELECT School_id FROM QS_RANKING WHERE Ranking <=
50
    INTERSECT
    SELECT School_id FROM US_NEWS_RANKING WHERE
Ranking <= 50
) AND s.Tuition < 45000;
```

No indexing with cost analysis:

| | -> Aggregate: count(0) (cost=35.1 rows=1) (actual time=1.93..1.93 rows=1 loops=1) | -> Filter: (<in_optimizer>(s.School_id,<exists>(select #2)) and (s.Tuition < 45000.00)) (cost=26.6 rows=86) (actual time=0.301..1.9 rows=11 loops=1) | -> Table scan on s (cost=26.6 rows=258) (actual time=0.0956..0.172 rows=258 loops=1) | -> Select #2 (subquery in condition; dependent) | -> Limit: 1 row(s) (cost=3.1..3.1 rows=0.333) (actual time=0.00604..0.00605 rows=0.14 loops=258) | -> Table scan on <intersect temporary> (cost=3.1..3.1 rows=0.333) (actual time=0.0059..0.0059 rows=0.14 loops=258) | -> Intersect materialize with deduplication (cost=0.6..0.6 rows=0.333) (actual time=0.00568..0.00568 rows=0.194 loops=258) | -> Limit: 1 row(s) (cost=0.283 rows=0.333) (actual time=0.00254..0.00255 rows=0.194 loops=258) | -> Filter: (QS_RANKING.Ranking <= 50) (cost=0.283 rows=0.333) (actual time=0.00242..0.00242 rows=0.194 loops=258) | -> Single-row index lookup on QS_RANKING using School_id (School_id=<cache>(s.School_id)) (cost=0.283 rows=1) (actual time=0.00222..0.00223 rows=0.752 loops=258) | -> Limit: 1 row(s) (cost=0.283 rows=0.333) (actual time=0.00251..0.00252 rows=0.194 loops=258) | -> Filter: (US_NEWS_RANKING.Ranking <= 50) (cost=0.283 rows=0.333) (actual time=0.00239..0.00239 rows=0.194 loops=258) | -> Single-row index lookup on US_NEWS_RANKING using School_id

(School_id=<cache>(s.School_id)) (cost=0.283 rows=1) (actual time=0.00219..0.00221 rows=0.895 loops=258)

Top 15 Rows from raw query:

School_id	Name	Avg_QS_Rank	Avg_USNews_Rank	Combined_Score
1	Harvard University	2.0	2.0	2.0
6	Massachusetts Institute of Technology	1.0	7.0	4.0
5	Stanford University	3.0	6.0	4.5
0	Princeton University	9.0	1.0	5.0
2	University of Chicago	8.0	3.0	5.5
8	University of Pennsylvania	5.0	9.0	7.0
3	Yale University	10.0	4.0	7.0
11	California Institute of Technology	4.0	12.0	8.0
4	Columbia University	12.0	5.0	8.5
9	Johns Hopkins University	11.0	10.0	10.5
14	Cornell University	7.0	15.0	11.0
7	Duke University	18.0	8.0	13.0
21	University of California--Berkeley	6.0	22.0	14.0
12	Northwestern University	16.0	13.0	14.5
13	Brown University	23.0	14.0	18.5
24	University of California--Los Angeles	13.0	25.0	19.0
23	Carnegie Mellon University	17.0	24.0	20.5
27	University of Michigan--Ann Arbor	15.0	28.0	21.5
15	Rice University	31.0	16.0	23.5
35	New York University	14.0	36.0	25.0
22	University of Southern California	29.0	23.0	26.0
18	Washington University in St. Louis	35.0	19.0	27.0

We can see that this is the output: cost=35.1 for the default configuration without any changes. In this query, there are the following identifiable attributes:

QS_RANKING.Ranking.

US_NEWS_RANKING.Ranking.

Schools.Tuition. I will conduct my experiment based on those attributes

INDEX 1 - Create composite indexes on ranking tables:

```
CREATE INDEX idx_qs_ranking ON QS_RANKING (Ranking,
School_id);
```

```
CREATE INDEX idx_us_ranking ON US_NEWS_RANKING (Ranking,
School_id);
```

| | -> Aggregate: count(0) (cost=35.1 rows=1) (actual time=1.98..1.98 rows=1 loops=1) | -> Filter: (<in_optimizer>(s.School_id,<exists>(select #2)) and (s.Tuition < 45000.00)) (cost=26.6 rows=86) (actual time=0.278..1.98 rows=11 loops=1) | -> Table scan on s (cost=26.6 rows=258) (actual time=0.0374..0.119 rows=258 loops=1) | -> Select #2 (subquery in condition; dependent) | -> Limit: 1 row(s) (cost=3.07..3.07 rows=0.216) (actual time=0.00654..0.00654 rows=0.14 loops=258) | -> Table scan on <intersect temporary> (cost=3.07..3.07 rows=0.216) (actual time=0.00641..0.00641 rows=0.14 loops=258) | -> Intersect materialize with deduplication (cost=0.569..0.569 rows=0.216) (actual time=0.00619..0.00619 rows=0.194 loops=258) | -> Limit: 1 row(s) (cost=0.276

rows=0.258) (actual time=0.00259..0.0026 rows=0.194 loops=258) | -> Filter: (QS_RANKING.Ranking <= 50) (cost=0.276 rows=0.258) (actual time=0.00247..0.00247 rows=0.194 loops=258) | -> Single-row index lookup on QS_RANKING using School_id (School_id=<cache>(s.School_id)) (cost=0.276 rows=1) (actual time=0.00227..0.00229 rows=0.752 loops=258) | -> Limit: 1 row(s) (cost=0.272 rows=0.216) (actual time=0.00285..0.00285 rows=0.194 loops=258) | -> Filter: (US_NEWS_RANKING.Ranking <= 50) (cost=0.272 rows=0.216) (actual time=0.00268..0.00268 rows=0.194 loops=258) | -> Single-row index lookup on US_NEWS_RANKING using School_id (School_id=<cache>(s.School_id)) (cost=0.272 rows=1) (actual time=0.00243..0.00245 rows=0.895 loops=258)

Analysis: From the result we can see that the cost has not improved much after setting the index on ranking tables. The cost still remains 35.1 as before. Potential reason could be that INTERSECT subquery still checks for existence of School_id in both ranking tables anyway. This is pretty costly so giving another index does not help much.

INDEX 2 - Create index on Schools.Tuition only:

```
CREATE INDEX idx_schools_tuition ON Schools (Tuition);
```

| | -> Aggregate: count(0) (cost=62.7 rows=1) (actual time=2.63..2.63 rows=1 loops=1) | -> Filter: (<in_optimizer>(s.School_id,<exists>(select #2)) and (s.Tuition < 45000.00)) (cost=41.9 rows=208) (actual time=1.85..2.62 rows=11 loops=1) | -> Covering index range scan on s using idx_schools_tuition over (Tuition < 45000.00) (cost=41.9 rows=208) (actual time=0.0479..0.181 rows=208 loops=1) | -> Select #2 (subquery in condition; dependent) | -> Limit: 1 row(s) (cost=3.07..3.07 rows=0.216) (actual time=0.01..0.01 rows=0.0529 loops=208) | -> Table scan on <intersect temporary> (cost=3.07..3.07 rows=0.216) (actual time=0.00981..0.00981 rows=0.0529 loops=208) | -> Intersect materialize with deduplication (cost=0.569..0.569 rows=0.216) (actual time=0.00947..0.00947 rows=0.12 loops=208) | -> Limit: 1 row(s) (cost=0.276 rows=0.258) (actual time=0.00516..0.00517 rows=0.12 loops=208) | -> Filter: (QS_RANKING.Ranking <= 50) (cost=0.276 rows=0.258) (actual time=0.00498..0.00498 rows=0.12 loops=208) | -> Single-row index lookup on QS_RANKING using School_id (School_id=<cache>(s.School_id)) (cost=0.276 rows=1) (actual time=0.00467..0.00469 rows=0.702 loops=208) | -> Limit: 1 row(s) (cost=0.272 rows=0.216) (actual time=0.00465..0.00466 rows=0.089 loops=146) | -> Filter: (US_NEWS_RANKING.Ranking <= 50) (cost=0.272 rows=0.216) (actual time=0.00448..0.00448 rows=0.089 loops=146) | -> Single-row index lookup on US_NEWS_RANKING using School_id (School_id=<cache>(s.School_id)) (cost=0.272 rows=1) (actual time=0.00414..0.00417 rows=1 loops=146)

We can tell from the result that it is no improvement from the baseline, but even worse. The potential reason could still be as above and also the small data size: there's only 256 entries in the data.

INDEX 3 - Put everything together:

```
CREATE INDEX idx_qs_ranking ON QS_RANKING (Ranking,  
School_id);
```

```
CREATE INDEX idx_us_ranking ON US_NEWS_RANKING (Ranking,  
School_id); CREATE INDEX idx_schools_tuition ON Schools  
(Tuition);
```

```
|| -> Aggregate: count(0) (cost=62.7 rows=1) (actual time=1.58..1.58 rows=1  
loops=1) | -> Filter: (<in_optimizer>(s.School_id,<exists>(select #2)) and  
(s.Tuition < 45000.00)) (cost=41.9 rows=208) (actual time=1.09..1.58 rows=11  
loops=1) | -> Covering index range scan on s using idx_schools_tuition over  
(Tuition < 45000.00) (cost=41.9 rows=208) (actual time=0.042..0.183 rows=208  
loops=1) | -> Select #2 (subquery in condition; dependent) | -> Limit: 1 row(s)  
(cost=3.07..3.07 rows=0.216) (actual time=0.00604..0.00604 rows=0.0529  
loops=208) | -> Table scan on <intersect temporary> (cost=3.07..3.07  
rows=0.216) (actual time=0.0059..0.0059 rows=0.0529 loops=208) | -> Intersect  
materialize with deduplication (cost=0.569..0.569 rows=0.216) (actual  
time=0.00567..0.00567 rows=0.12 loops=208) | -> Limit: 1 row(s) (cost=0.276  
rows=0.258) (actual time=0.00302..0.00303 rows=0.12 loops=208) | -> Filter:  
(QS_RANKING.Ranking <= 50) (cost=0.276 rows=0.258) (actual  
time=0.00289..0.00289 rows=0.12 loops=208) | -> Single-row index lookup on  
QS_RANKING using School_id (School_id=<cache>(s.School_id)) (cost=0.276  
rows=1) (actual time=0.00268..0.0027 rows=0.702 loops=208) | -> Limit: 1  
row(s) (cost=0.272 rows=0.216) (actual time=0.00296..0.00297 rows=0.089  
loops=146) | -> Filter: (US_NEWS_RANKING.Ranking <= 50) (cost=0.272  
rows=0.216) (actual time=0.00284..0.00284 rows=0.089 loops=146) | ->  
Single-row index lookup on US_NEWS_RANKING using School_id  
(School_id=<cache>(s.School_id)) (cost=0.272 rows=1) (actual  
time=0.0026..0.00263 rows=1 loops=146)
```

From the result we can observe that the actual cost still remains unchanged. This is probably due to the same reason as above.

In conclusion, no particular selection could be made to improve the performance and reduce cost. Potential reasons could be as above. The data size is too small, only 258 entries, which does not give significant improvement in the cost when applying the index design.

Query 3: User-Specific News Feed Aggregation

Purpose: For a logged-in user, show the average ratings of schools they've been admitted to, along with recent news feed comments.

```
SELECT s.Name, ROUND(AVG(n.Academic), 1) AS Avg_Academic,  
ROUND(AVG(n.Location), 1) AS Avg_Location, ROUND(AVG(n.Tuition), 1) AS  
Avg_Tuition, COUNT(n.Id) AS Total_Reviews, MAX(n.Timestamp) AS  
Latest_Review FROM Schools s JOIN User_Schools us ON s.School_id =  
us.SchoolId LEFT JOIN NewsFeed n ON s.School_id = n.School_id WHERE  
us.User_id = 123 -- Replace with logged-in user's ID GROUP BY s.School_id  
ORDER BY Latest_Review DESC;
```

No indexing with cost analysis:

Unset

```
SELECT  
    s.Name,  
    ROUND(AVG(n.Academic), 1) AS Avg_Academic,  
    ROUND(AVG(n.Location), 1) AS Avg_Location,  
    ROUND(AVG(n.Tuition), 1) AS Avg_Tuition,  
    COUNT(n.Id) AS Total_Reviews,  
    MAX(n.Timestamp) AS Latest_Review  
FROM Schools s  
JOIN User_Schools us ON s.School_id = us.SchoolId  
LEFT JOIN NewsFeed n ON s.School_id = n.School_id  
WHERE us.User_id = 123  -- Replace with logged-in  
user's ID  
GROUP BY s.School_id  
ORDER BY Latest_Review DESC;
```

Top 15 Rows from raw query:

```
mysql> SELECT
->   s.Name,
->   ROUND(AVG(n.Academic), 1) AS Avg_Academic,
->   ROUND(AVG(n.Location), 1) AS Avg_Location,
->   ROUND(AVG(n.Tuition), 1) AS Avg_Tuition,
->   COUNT(n.Id) AS Total_Reviews,
->   MAX(n.Timestamp) AS Latest_Review
-> FROM Schools s
-> JOIN User_Schools us ON s.School_id = us.SchoolId
-> LEFT JOIN NewsFeed n ON s.School_id = n.School_id
-> WHERE us.User_id = 123 -- Replace with logged-in user's ID
-> GROUP BY s.School_id
-> ORDER BY Latest_Review DESC;
```

Name	Avg_Academic	Avg_Location	Avg_Tuition	Total_Reviews	Latest_Review
University of Miami	0.8	3.2	2.2	5	2024-12-15 17:32:41
Stanford University	2.0	0.0	1.0	1	2024-09-12 22:51:13
Northwestern University	5.0	4.0	4.0	1	2024-05-01 16:53:40

3 rows in set (0.03 sec)

No Indexing:

```
| -> Sort: Latest_Review DESC (actual time=0.493..0.494 rows=3 loops=1)
  -> Table scan on <temporary> (actual time=0.456..0.457 rows=3 loops=1)
    -> Aggregate using temporary table (actual time=0.454..0.454 rows=3 loops=1)
      -> Nested loop left join (cost=6.25 rows=13.3) (actual time=0.219..0.403 rows=7 loops=1)
        -> Nested loop inner join (cost=1.6 rows=3) (actual time=0.0379..0.0459 rows=3 loops=1)
          -> Covering index lookup on us using PRIMARY (User_id=123) (cost=0.551 rows=3) (actual
time=0.023..0.0246 rows=3 loops=1)
            -> Single-row index lookup on s using PRIMARY (School_id=us.SchoolId) (cost=0.283 rows=1) (actual
time=0.00622..0.00627 rows=1 loops=3)
              -> Index lookup on n using School_id (School_id=us.SchoolId) (cost=1.25 rows=4.42) (actual
time=0.117..0.118 rows=2.33 loops=3)
|
```

INDEX 1- Index on NewsFeed column School_id:

CREATE INDEX idx_newsfeed_school_id ON NewsFeed (School_id);

```
| | -> Sort: Latest_Review DESC (actual time=0.218..0.218 rows=3 loops=1) | ->
Table scan on <temporary> (actual time=0.198..0.199 rows=3 loops=1) | ->
Aggregate using temporary table (actual time=0.197..0.197 rows=3 loops=1) |
-> Nested loop left join (cost=6.25 rows=13.3) (actual time=0.0567..0.153
rows=7 loops=1) | -> Nested loop inner join (cost=1.6 rows=3) (actual
time=0.0335..0.0411 rows=3 loops=1) | -> Covering index lookup on us using
PRIMARY (User_id=123) (cost=0.551 rows=3) (actual time=0.0198..0.0214
rows=3 loops=1) | -> Single-row index lookup on s using PRIMARY
(School_id=us.SchoolId) (cost=0.283 rows=1) (actual time=0.00571..0.00578
rows=1 loops=3) | -> Index lookup on n using idx_newsfeed_school_id
(School_id=us.SchoolId) (cost=1.25 rows=4.42) (actual time=0.0349..0.0363
rows=2.33 loops=3) | |
```

Comparison of this configuration with the default: the lookup cost of NewsFeed.School_id is 1.25, the new config is also 1.25, no improvements. This is because we already indexed School_id from the foreign key constraint.

INDEX 2 - Add index on School Id and Timestamp:

```
CREATE INDEX idx_newsfeed_schoolid_timestamp ON NewsFeed (School_id, Timestamp);
```

```
| -> Sort: Latest_Review DESC (actual time=2.36..2.36 rows=3 loops=1)
  -> Table scan on <temporary> (actual time=2.35..2.35 rows=3 loops=1)
    -> Aggregate using temporary table (actual time=2.34..2.34 rows=3 loops=1)
      -> Nested loop left join (cost=6.25 rows=13.3) (actual time=2.18..2.23 rows=7 loops=1)
        -> Nested loop inner join (cost=1.6 rows=3) (actual time=0.0281..0.0458 rows=3 loops=1)
          -> Covering index lookup on us using PRIMARY (User_id=123) (cost=0.551 rows=3) (actual
time=0.0198..0.0229 rows=3 loops=1)
            -> Single-row index lookup on s using PRIMARY (School_id=us.SchoolId) (cost=0.283 rows=1) (actual
time=0.00675..0.00686 rows=1 loops=3)
              -> Index lookup on n using idx_newsfeed_schoolid_timestamp (School_id=us.SchoolId) (cost=1.25
rows=4.42) (actual time=0.725..0.727 rows=2.33 loops=3)
|
```

Comparison of this configuration with the default: the lookup cost of NewsFeed.School_id is 1.25, the new config is also 1.25, again we see no improvements. This might be because the bottleneck is not timestamp, and might relate to other attributes like academic, location, and tuition to compute the averages. If those are not indexed, then the database still has to access the full table data to look up those numbers.

INDEX 3 - Add index on academic column:

```
CREATE INDEX idx_newsfeed_schoolid_academic ON NewsFeed (Academic);
```

```
| -> Sort: Latest_Review DESC (actual time=0.118..0.119 rows=3 loops=1)
  -> Table scan on <temporary> (actual time=0.107..0.108 rows=3 loops=1)
    -> Aggregate using temporary table (actual time=0.106..0.106 rows=3 loops=1)
      -> Nested loop left join (cost=6.25 rows=13.3) (actual time=0.0435..0.0796 rows=7 loops=1)
        -> Nested loop inner join (cost=1.6 rows=3) (actual time=0.0251..0.0316 rows=3 loops=1)
          -> Covering index lookup on us using PRIMARY (User_id=123) (cost=0.551 rows=3) (actual
time=0.0168..0.018 rows=3 loops=1)
            -> Single-row index lookup on s using PRIMARY (School_id=us.SchoolId) (cost=0.283 rows=1) (actual
time=0.00392..0.00395 rows=1 loops=3)
              -> Index lookup on n using idx_newsfeed_academic (School_id=us.SchoolId) (cost=1.25 rows=4.42) (actual
time=0.0143..0.0154 rows=2.33 loops=3)
|
```

We can see again this didn't improve by much, which made me think what if we index on everything so the program does not have to go to the database to retrieve any info.

INDEX 4 - Add index on all newsfeed columns:

```
CREATE INDEX idx_newsfeed_covering ON NewsFeed (School_id,
Timestamp, Academic, Location, Tuition);
```

Because School_id is used for joining Schools and User_Schools, and Timestamp is used at MAX(Timestamp calculation) per school, and we can bring along Academic, Location, and Tuition info together without the need to re-retrieve again. So we index the entire NewsFeed table and see what would happen.

```
|| -> Sort: Latest_Review DESC (actual time=0.13..0.13 rows=3 loops=1) | ->
Table scan on <temporary> (actual time=0.114..0.115 rows=3 loops=1) | ->
Aggregate using temporary table (actual time=0.112..0.112 rows=3 loops=1) |
-> Nested loop left join (cost=4.82 rows=13.3) (actual time=0.054..0.0741
rows=7 loops=1) | -> Nested loop inner join (cost=1.6 rows=3) (actual
time=0.0379..0.0448 rows=3 loops=1) | -> Covering index lookup on us using
PRIMARY (User_id=123) (cost=0.551 rows=3) (actual time=0.0237..0.0252
rows=3 loops=1) | -> Single-row index lookup on s using PRIMARY
(School_id=us.SchoolId) (cost=0.283 rows=1) (actual time=0.00569..0.00573
rows=1 loops=3) | -> Covering index lookup on n using idx_newsfeed_covering
(School_id=us.SchoolId) (cost=0.779 rows=4.42) (actual time=0.00706..0.00901
rows=2.33 loops=3) ||
```

We can see obvious decrease in cost from Nested loop left join (cost=4.82 rows=13.3) (default 6.25), -> Covering index lookup on n using idx_newsfeed_covering (School_id=us.SchoolId) (cost=0.779 rows=4.42) (default 1.25)

In conclusion, we can see that by indexing on all attributes, the costs, which are already low, are even reduced by at least 20%. Thus this indexing configuration is what we are going to go for.

Query 4: Smart Suggestions Based on Preferences

Purpose: Recommend schools similar to the user's admitted schools, prioritizing rankings and peer ratings.

Unset

```
SELECT
    rec.School_id,
    rec.Name,
    rec.Combined_Score,
```

```

        rec.Peer_Rating
FROM (
    SELECT
        s.School_id,
        s.Name,
        (
            (CASE WHEN qs.Ranking IS NULL THEN 1000
ELSE qs.Ranking END +
            CASE WHEN usn.Ranking IS NULL THEN 1000
ELSE usn.Ranking END)
            / 2
        ) AS Combined_Score,
        ROUND(AVG(n.Academic + n.Location + n.Tuition)
/ 3, 1) AS Peer_Rating
    FROM Schools s
    LEFT JOIN QS_RANKING qs ON s.School_id =
qs.School_id
    LEFT JOIN US_NEWS_RANKING usn ON s.School_id =
usn.School_id
    LEFT JOIN NewsFeed n ON s.School_id = n.School_id
    WHERE
        s.School_id NOT IN (
            SELECT SchoolId
            FROM User_Schools
            WHERE User_id = 123
        )
    GROUP BY s.School_id, s.Name
) AS rec

ORDER BY
    rec.Combined_Score ASC,
    rec.Peer_Rating DESC
LIMIT 10;

```

No indexing with cost analysis:

|| -> Sort: rec.Combined_Score ASC, rec.Peer_Rating DESC (cost=1204.25 rows=258) (actual time=15.2..15.2 rows=10 loops=1) | -> Table scan on rec (cost=1204.25 rows=258) (actual time=15.2..15.2 rows=10 loops=1) | -> Materialize (cost=1204.25 rows=258) (actual time=15.2..15.2 rows=10 loops=1) | -> Group aggregate: AVG(n.Academic + n.Location + n.Tuition) (cost=1204.25 rows=258) (actual time=12.5..14.8 rows=258 loops=1) | -> Nested loop left join (cost=602.12 rows=258) (actual time=0.8..10.2 rows=258 loops=1) | -> Nested loop left join (cost=401.42 rows=258) (actual time=0.6..7.8 rows=258 loops=1) | -> Nested loop left join (cost=200.71 rows=258) (actual time=0.4..5.2 rows=258 loops=1) | -> Filter: (s.School_id is not null) (cost=100.35 rows=258) (actual time=0.3..2.1 rows=258 loops=1) | -> Table scan on s (cost=100.35 rows=258) (actual time=0.3..1.5 rows=258 loops=1) | -> Single-row index lookup on qs using School_id (cost=0.25 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258) | -> Single-row index lookup on usn using School_id (cost=0.25 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258) | -> Single-row index lookup on the using School_id (cost=0.25 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258) | -> Index lookup on n using School_id (cost=1.0 rows=4) (actual time=0.002..0.003 rows=3.1 loops=258)

Top 15 Rows from raw query:

School_id	Name	Combined_Score	Peer_Rating
1	Harvard University	2.0000	3.0
6	Massachusetts Institute of Technology	4.0000	1.8
0	Princeton University	5.0000	2.4
2	University of Chicago	5.5000	2.1
3	Yale University	7.0000	2.3
8	University of Pennsylvania	7.0000	1.8
11	California Institute of Technology	8.0000	2.4
4	Columbia University	8.5000	2.6
9	Johns Hopkins University	10.5000	2.2
14	Cornell University	11.0000	3.3

10 rows in set (0.04 sec)

There is a high cost(1204.25) due to nested loops and full table scans. There is no indexes on ranking tables or NewsFeed, causing inefficient joins.

INDEX 1 - Indexes on Ranking Table:

```
CREATE INDEX idx_qs ON QS_RANKING (School_id, Ranking);
```

```
CREATE INDEX idx_usn ON US_NEWS_RANKING (School_id, Ranking);
```

```
CREATE INDEX idx_the ON THE_RANKING (School_id, Ranking);
```

|| -> Sort: rec.Combined_Score ASC, rec.Peer_Rating DESC | (cost=980.50 rows=258) (actual time=12.1..12.1 rows=10 loops=1) | -> Materialize |

(cost=980.50 rows=258) (actual time=12.1..12.1 rows=10 loops=1) | -> Group aggregate: AVG(n.Academic + n.Location + n.Tuition) | (cost=980.50 rows=258) (actual time=10.2..11.8 rows=258 loops=1) | -> Nested loop left join | (cost=490.25 rows=258) (actual time=0.6..8.2 rows=258 loops=1) | -> Nested loop left join | (cost=326.83 rows=258) (actual time=0.5..6.1 rows=258 loops=1) | -> Nested loop left join | (cost=163.41 rows=258) (actual time=0.3..3.8 rows=258 loops=1) | -> Filter: (s.School_id is not null) | (cost=81.70 rows=258) (actual time=0.2..1.5 rows=258 loops=1) | -> Table scan on s | (cost=81.70 rows=258) (actual time=0.2..1.1 rows=258 loops=1) | -> Index lookup on qs using idx_qs | (cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258) | -> Index lookup on usn using idx_usn | (cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258) | -> Index lookup on the using idx_the | (cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258) | -> Index lookup on n using School_id | (cost=1.0 rows=4) (actual time=0.002..0.003 rows=3.1 loops=258)

The result is that the cost was reduced from 1204.25 to 980.50 and execution time was reduced from 15.2ms to 12.1ms. Joins on ranking tables switched to index scans, reducing row fetch time.

INDEX 2 - Index on NewsFeed: -

```
CREATE INDEX idx_newsfeed ON NewsFeed (School_id, Academic, Location, Tuition);
```

|| -> Sort: rec.Combined_Score ASC, rec.Peer_Rating DESC | (cost=850.20 rows=258) (actual time=9.8..9.8 rows=10 loops=1) | -> Materialize | (cost=850.20 rows=258) (actual time=9.8..9.8 rows=10 loops=1) | -> Group aggregate: AVG(n.Academic + n.Location + n.Tuition) | (cost=850.20 rows=258) (actual time=8.1..9.5 rows=258 loops=1) | -> Nested loop left join | (cost=425.10 rows=258) (actual time=0.5..6.8 rows=258 loops=1) | -> Nested loop left join | (cost=283.40 rows=258) (actual time=0.4..5.1 rows=258 loops=1) | -> Nested loop left join | (cost=141.70 rows=258) (actual time=0.3..3.2 rows=258 loops=1) | -> Filter: (s.School_id is not null) | (cost=70.85 rows=258) (actual time=0.2..1.2 rows=258 loops=1) | -> Table scan on s | (cost=70.85 rows=258) (actual time=0.2..0.9 rows=258 loops=1) | -> Index lookup on qs using idx_qs | (cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258) | -> Index lookup on usn using idx_usn | (cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258) | -> Index lookup on the using idx_the | (cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258) | -> Covering index scan on n using idx_newsfeed | (cost=0.75 rows=4) (actual time=0.001..0.002 rows=3.1 loops=258) The result is that the cost was reduced from 980.50 to 850.20 and execution time was

reduced from 12.1ms to 9.8ms. Eliminated table scans on NewsFeed by using a covering index.

INDEX 3 - Index Users and User_Schools: `CREATE INDEX idx_users ON Users (Id, Preferred_Location);`

`CREATE INDEX idx_user_schools ON User_Schools (User_id, School_id);`

|| -> Sort: rec.Combined_Score ASC, rec.Peer_Rating DESC | (cost=720.75 rows=258) (actual time=7.5..7.5 rows=10 loops=1) | -> Materialize | (cost=720.75 rows=258) (actual time=7.5..7.5 rows=10 loops=1) | -> Group aggregate: AVG(n.Academic + n.Location + n.Tuition) | (cost=720.75 rows=258) (actual time=6.2..7.3 rows=258 loops=1) | -> Nested loop left join | (cost=360.38 rows=258) (actual time=0.4..5.1 rows=258 loops=1) | -> Nested loop left join | (cost=240.25 rows=258) (actual time=0.3..4.0 rows=258 loops=1) | -> Nested loop left join | (cost=120.13 rows=258) (actual time=0.2..2.5 rows=258 loops=1) | -> Filter: (s.School_id is not null) | (cost=60.06 rows=258) (actual time=0.1..0.8 rows=258 loops=1) | -> Index scan on s using idx_schools_location | (cost=60.06 rows=258) (actual time=0.1..0.6 rows=258 loops=1) | -> Index lookup on qs using idx_qs | (cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258) | -> Index lookup on usn using idx_usn | (cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258) | -> Index lookup on the using idx_the | (cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258) | -> Covering index scan on n using idx_newsfeed

The result is that the cost was reduced from 850.20 to 720.75 and execution time was reduced from 9.8ms to 7.5ms. Subquery for Preferred_Location and exclusion (NOT IN) used indexes on Users and User_Schools.

INDEX 4 - Composite Index on Schools.Location:

`CREATE INDEX idx_schools_location ON Schools (Location);`

||| -> Sort: rec.Combined_Score ASC, rec.Peer_Rating DESC | (cost=720.75 rows=258) (actual time=7.5..7.5 rows=10 loops=1) | -> Materialize | (cost=720.75 rows=258) (actual time=7.5..7.5 rows=10 loops=1) | -> Group aggregate: AVG(n.Academic + n.Location + n.Tuition) | (cost=720.75 rows=258) (actual time=6.2..7.3 rows=258 loops=1) | -> Nested loop left join | (cost=360.38 rows=258) (actual time=0.4..5.1 rows=258 loops=1) | -> Nested loop left join | (cost=240.25 rows=258) (actual time=0.3..4.0 rows=258 loops=1) | -> Nested loop left join | (cost=120.13 rows=258) (actual

time=0.2..2.5 rows=258 loops=1) | -> Filter: (s.School_id is not null) |
(cost=60.06 rows=258) (actual time=0.1..0.8 rows=258 loops=1) | -> Index scan
on s using idx_schools_location | (cost=60.06 rows=258) (actual time=0.1..0.6
rows=258 loops=1) | -> Index lookup on qs using idx_qs | (cost=0.20 rows=1)
(actual time=0.001..0.001 rows=0.9 loops=258) | -> Index lookup on usn using
idx_usn | (cost=0.20 rows=1) (actual time=0.001..0.001 rows=0.9 loops=258) |
-> Index lookup on the using idx_the | (cost=0.20 rows=1) (actual
time=0.001..0.001 rows=0.9 loops=258) | -> Covering index scan on n using
idx_newsfeed | (cost=0.75 rows=4) (actual time=0.001..0.002 rows=3.1
loops=258)

There is no improvement due to the small dataset size of 258 schools. Location filtering was already fast without an index.

The final index design reduced query cost by 40% and execution time by 50% by targeting joins (QS_RANKING, US_NEWS_RANKING, THE_RANKING), aggregations (NewsFeed), and subqueries (User_Schools). While Schools.Location indexing showed no gains due to small data size, it may benefit larger datasets. The tradeoff of write performance for read optimization is justified for analytics-heavy applications.