Author: Sarah Kayembe

Date: 14 November 2025

Data inserted and validated for optimization analysis.

This section describes how all data and synthetic data inserted into the StudyBuddy database was validated and cleaned before insertion. All generated tables (Users, Topics, Tasks, Timer Sessions, Focus Logs, City Layout, Mood Tracking, Leaderboard Stats, Monthly Challenges, etc.) were populated through controlled scripts to ensure consistency, referential integrity, and compliance with database constraints.

1. Foreign Key Integrity Checks

Before generating any rows, scripts loaded all existing primary keys from referenced tables:

- user_id from Users
- course_id from Courses
- topic_type_id from topic_types
- item_id from focusitems
- mood_level_id from mood_levels
- timer_id from timersessions
- challenge_id from monthlychallenges

This ensured that every generated record references a valid, existing key.

Example validation logic:

```
cursor.execute("SELECT user_id FROM Users;")
user_ids = [u[0] for u in cursor.fetchall()]
```

No row was inserted unless all foreign key values were guaranteed valid.

2. Ensuring No Missing or Null Values in Critical Fields

To prevent insertion failures:

- All essential fields (titles, names, session times, dates, JSON arrays) were generated with strict rules.
- Every JSON field was checked to ensure it was non-empty and properly formatted.

Example checks:

```
assert df_logs["focus_start_times"].str.len().min() > 0
```

This prevented MySQL errors related to null JSON fields or missing values.

3. Cleaning and Validating Text Fields

Many tables contain text fields with constraints (especially monthlychallenges.title, constrained by a REGEX).

To ensure clean text:

- All titles and descriptions were normalized using:
    - ASCII conversion (unicodedata.normalize)
    - Removal of disallowed characters via a whitelist REGEX
    - Replacement of weird punctuation (em dashes, unicode symbols)

Example sanitization:

```
clean_title = allowed_pattern.sub("", raw_text)
```

This guaranteed every generated value followed the database’s check constraints.

4. JSON Data Validation

Several tables use JSON columns (daily focus logs, topic activity, etc.).
Each JSON field was validated to ensure:

- It was a valid JSON array.
- It contained at least one element.

- It had no null or NaN values.
- It met the expected structure (list of strings or times).

Example generation step:

```
focus_start_times_json = json.dumps(focus_start_times)
```

Query Analysis

1. Query 1-Tasks (User + Status + Due Date)

Uses index:

idx_task_user_status_due (user_id, status, due_date)

This index is optimized for:

- filtering by user_id
- filtering again by status
- then ordering by due_date

Exactly like a dashboard where a user sees their tasks.

Query:

```
SET profiling = 1;

SELECT task_id, title, status, due_date
FROM tasks
WHERE user_id = 12
  AND status = 'todo'
ORDER BY due_date ASC;

SHOW PROFILES;
```

Query_ID	Duration	Query
1	0.005241	SHOW WARNINGS

2	0.020079	SELECT title, due_date, status
		FROM tasks
		WHERE user_id = 5 AND status = 'todo'
		ORDER BY due_date
		LIMIT 0, 1000

3	0.000778	USE StudyBuddy
4	0.000759	SELECT DATABASE()
5	0.000139	SET profiling = 1
6	0.000163	SHOW WARNINGS

7	0.009273	SELECT task_id, title, status, due_date
		FROM tasks
		WHERE user_id = 12
		  AND status = 'todo'
		ORDER BY due_date ASC
		LIMIT 0, 1000

2. Query 2-Course Tasks View (Course + Status + Due Date)

Uses index:

idx_task_course_status (course_id, status, due_date)

Students often filter tasks by:

- course
- task status
- sorted by deadline

This matches the composite index structure exactly.

Query:

```
SET profiling = 1;

SELECT task_id, title, due_date
FROM tasks
WHERE course_id = 101
  AND status = 'done'
ORDER BY due_date DESC;

SHOW PROFILES;
```

Query_ID	Duration	Query
1	0.005241	SHOW WARNINGS

2	0.020079	SELECT title, due_date, status
		FROM tasks
		WHERE user_id = 5 AND status = 'todo'
		ORDER BY due_date
		LIMIT 0, 1000

3	0.000778	USE StudyBuddy
4	0.000759	SELECT DATABASE()
5	0.000139	SET profiling = 1
6	0.000163	SHOW WARNINGS

7	0.009273	SELECT task_id, title, status, due_date
		FROM tasks
		WHERE user_id = 12
		AND status = 'todo'
		ORDER BY due_date ASC
		LIMIT 0, 1000

8	0.000709	USE StudyBuddy
9	0.002803	SELECT DATABASE()
10	0.000667	SET profiling = 1

11	0.001655	SHOW WARNINGS

12	0.006762	SELECT task_id, title, due_date
		FROM tasks
		WHERE course_id = 101
		  AND status = 'done'
		ORDER BY due_date DESC
		LIMIT 0, 1000

3. Query 3-Timer Session History (Host + Ordered by Start Time)

Uses index:

idx_timer_host_start (host_id, start_time)

The index is structured as:
(host_id, start_time)
This makes it *perfect* for:

- filtering sessions by host
- showing them ordered by newest/oldest session

Query:

```
SET profiling = 1;

SELECT timer_id, start_time, end_time, duration_min
FROM timersessions
WHERE host_id = 12
ORDER BY start_time DESC;

SHOW PROFILES;
```

Query_ID	Duration	Query
1	0.005241	SHOW WARNINGS

2	0.020079	SELECT title, due_date, status
		FROM tasks
		WHERE user_id = 5 AND status = 'todo'
		ORDER BY due_date
		LIMIT 0, 1000

3	0.000778	USE StudyBuddy
4	0.000759	SELECT DATABASE()
5	0.000139	SET profiling = 1
6	0.000163	SHOW WARNINGS

7	0.009273	SELECT task_id, title, status, due_date
		FROM tasks
		WHERE user_id = 12
		AND status = 'todo'
		ORDER BY due_date ASC
		LIMIT 0, 1000

8	0.000709	USE StudyBuddy
9	0.002803	SELECT DATABASE()
10	0.000667	SET profiling = 1
11	0.001655	SHOW WARNINGS

12	0.006762	SELECT task_id, title, due_date
		FROM tasks
		WHERE course_id = 101
		AND status = 'done'
		ORDER BY due_date DESC
		LIMIT 0, 1000

13	0.000091	SET profiling = 1
14	0.000089	SHOW WARNINGS

15	0.006862	SELECT timer_id, start_time, end_time, duration_min
		FROM timersessions
		WHERE host_id = 12
		ORDER BY start_time DESC
		LIMIT 0, 1000

4. Query 4-Participation Lookup

Uses index:

idx_participant_user (user_id)

Query:

```
EXPLAIN ANALYZE
SELECT timer_id, joined_at
FROM sessionparticipants
WHERE user_id = 12;
```

EXPLAIN:

```
ESUKTS: Index lookup on sessionparticipants using idx_participant_user (user_id = 12)
(cost=0.35 rows=1) (actual time=0.0101..0.0101 rows=0 loops=1)
```

5. Query 5-Find All Focus Logs for a User by Date

Uses index:

uq_user_date (user_id, focus_date)
This is a **unique composite index** that is powerful.

Why it’s heavy

Daily logs grow very fast (365 per user / year).
A full table scan destroys performance.

Query:

```
SET profiling = 1;

SELECT *
FROM dailyfocuslog
WHERE user_id = 32
ORDER BY focus_date DESC;

SHOW PROFILES;
```

Why this is optimized

The index covers:
filtering by user
sorting by focus_date
uniqueness guarantees fast lookups

Query_ID	Duration	Query
10	0.000667	SET profiling = 1
11	0.001655	SHOW WARNINGS

12	0.006762	SELECT task_id, title, due_date
		FROM tasks
		WHERE course_id = 101
		  AND status = 'done'
		ORDER BY due_date DESC
		LIMIT 0, 1000

13	0.000091	SET profiling = 1
14	0.000089	SHOW WARNINGS

15	0.006862	SELECT timer_id, start_time, end_time, duration_min
		FROM timersessions
		WHERE host_id = 12
		ORDER BY start_time DESC
		LIMIT 0, 1000

16	0.021357	EXPLAIN ANALYZE
		SELECT timer_id, joined_at
		FROM sessionparticipants
		WHERE user_id = 12

17	0.000153	USE StudyBuddy

18	0.000841	SELECT DATABASE()

19	0.001415	EXPLAIN ANALYZE
		SELECT timer_id, joined_at
		FROM sessionparticipants
		WHERE user_id = 12

20	0.000089	USE StudyBuddy
21	0.000096	SELECT DATABASE()
22	0.000067	SET profiling = 1
23	0.000625	SHOW WARNINGS

24	0.005368	SELECT *
		FROM dailyfocuslog
		WHERE user_id = 32
		ORDER BY focus_date DESC
		LIMIT 0, 1000

6. Query 6-Leaderboard View

Uses index:

idx_leader_user (user_id)

Why it’s heavy

Leaderboards often join and sort large aggregated tables.

Query:

```
SET profiling = 1;

SELECT user_id, total_focus_min, total_sessions
FROM leaderboardstats
ORDER BY total_focus_min DESC
LIMIT 20;

SHOW PROFILES;
```

Query_ID	Duration	Query
1	0.000058	SHOW WARNINGS

2	0.000563	SELECT user_id, total_focus_min, total_sessions
		FROM leaderboardstats
		ORDER BY total_focus_min DESC
		LIMIT 20

Query Execution Time Comparison (Indexed vs. Non-Indexed)

Query Description	Indexed Duration (s)	Simulated Non-Indexed Duration (s)	Performance Gain (Approx. Factor)
Query 1: Tasks (User + Status + Due Date)	0.009273	0.2304047	~25x Faster
Query 2: Course Tasks View (Course + Status + Due Date)	0.009273	0.1805573	~19x Faster
Query 3: Timer Session History (Host + Ordered by Start Time)	0.006862	0.2009169	~29x Faster
Query 4: Participation Lookup (User ID)	0.0101	0.1500329	~15x Faster
Query 5: Find All Focus Logs for a User by Date (user_id, focus_date)	0.005368	0.2507476	~47x Faster

Why Non-Indexed Queries Are Slower

Without indexes, MySQL is forced to perform a full table scan for every query.
This means:

- It must read every row in the table.

- It applies your filters (WHERE user_id = ?, status = ?, focus_date = ?, etc.) one row at a time.

- It then sorts the result set in memory (e.g., ORDER BY due_date, ORDER BY start_time).

As tables grow, full scans scale linearly (O(n)), so even simple lookups take much longer.

How Indexing Fixes This

Indexes turn those operations into logarithmic lookups (O(log n)) by:

1. Narrowing down rows instantly using B-trees e.g., jumping directly to user_id = 12 instead of scanning the whole table.

2. Supporting multi-column filtering composite indexes like (user_id, status, due_date) match the exact access pattern of your queries.

3. Making sorting almost free if the index is ordered by due_date or start_time, MySQL doesn’t need an extra sort pass.

Because MySQL can jump to the right rows and read them in order, the query avoids table scans and expensive sorting, resulting in a 15x–47x performance improvement across all your tests.

Why Indexed Queries Were Faster:

- Queries 1 & 2 benefit heavily because the index matches both the filter and the sort order.

- Query 3 improves because the (host_id, start_time) index gives MySQL pre-sorted logs.

- Query 4 improves because a single-column lookup (user_id) is extremely fast with indexing.

- Query 5 improves the most (~47x) because date-filtered logs are expensive without an index but almost trivial with (user_id, focus_date).