Performance Analysis and Optimization Report

This section shows the main bottlenecks that we had encountered during the development of our StudyBuddy database and website in phase 3.

Jacob Craig

1. Data Loading

- In phase II we switched the clean data used for course_resources which allowed for more visually appealing data in the database for the resource table
- This new data allowed for the second version of CourseResourceCleaner
    - Normalizes text via functions strip_weird and normalize snippet
    - Writes a cleaned CSV

2. Query Optimization

- SearchCourseSmart: Originally was doing full table scans on courses
    - Added a CREATE INDEX idx_courses_code_name
    - MySQL range scans on course_code and course_name rather than the entire table
    - Complexity improved from O(N) to O(Log N + k) where k is the the number of matching rows
- Group Membership & Join Requests: Membership had no constraints regarding making owners, ensuring that users could not join a group if the max capacity was reached, dealing with multiple users trying to join at the same time, etc. Duplicates on StartDirectConversation started occurring a lot in tests, there needed to be a way to stop duplicate direct messages.
    - Added a CREATE INDEX idx_gm_group_user
    - Added a CREATE INDEX idx_jr_group_user_status
    - Procedures such as JoinGroupWithLock, KickGroupMember, RequestJoinPublicGroup, and ApproveJoinRequest now use index lookups instead of scanning by group
    - Membership tests (WHERE group_id = ? AND user_id = ?) are effectively O(log N) scans instead of O(N) scans as the number of members/requests grow
- Direct Messaging + Inbox: loading direct message conversations and messages required sorting direct_message tables by time. If the direct_messages table became large, this would be slow.
    - Added a CREATE INDEX idx_dm_convo_time

- Added a CREATE INDEX idx_dc_users
- GetDirectMessages and GetDmInboxForUser tables now do range scans on conversation_id, sent_at) instead of sorting in mem
- StartDirectConversation table now can check if a conversation already exists using idx_dc_users, stopping duplicates from happening with an easy index lookup

3. Matching Algorithm

- The first version of the StudyBuddy Match tried to get compatibility with study groups. Since the StudyBuddy schema cannot tell exactly what course a user is in, the original used study groups to see a users course. This made it impossible for a user to be matched with another user without joining a study group.
    - Made it such that students can select up to five courses they want to match on in the match_profile
    - Added more preferences to Match_Profile table to allow for more added to the scoring algorithm
    - Added a Match_Profile_Course table for many-to-many mapping between users and courses they want to match on
    - GetStudyBuddyMatches (Procedure)
        - Reduces candidate set via WITH (CTEs)
            - Only allows matches for users who share at least one course
            - candidate_base aggregates shared course count and other data
        - Score_candidates used formula based on college, style, meeting preference, goals, time, noise, shared courses, and age proximity
    - Based on the new tables and GetStudyBuddyMatches procedure
        - The in depth work happens on a much smaller candidate pool (users who share courses instead of all users)
        - Primary key on match_profile_course (user_id, course_id) help MySQL used index based joins for user/course relationships
        - Instead of O(U * G) scans over users and groups, we now have O(C * Log C) over a candidate pool C

4. Concurrency in Groups

- Original join group logic used (check count then insert)
    - Vulnerable when two users clicking at the same time could possibly get in if the study group is only one away to being full
    - JoinGroupWithLock procedure uses FOR UPDATE to lock relevant rows until transactions complete
        - This allows for the join operation to be atomic even under concurrent requests
        - Group overfill is avoided
        - Complexity checks are the same as before with O(Log N) thanks to idx_gm_group_user, but with isolation

5. Message and Request

- In early versions of DMs it was possible to
    - Send multiple messages without the target user accepting the message request
        - We wanted it such that a user can send one message and then the other user must accept the message_request (sending the one message is not mandatory)
        - This lead to multiple overlapping requests between the same pair.
    - Direct_Conversation and Direct_Message tables were added with constraints
        - uq_message_request_pair and uq_dc_pair to make sure the conversations and requests were unique pairs
    - Procedures:
        - StartDirectConversation
            - Normalized pairs such that u1, u2 is the same as u2, u1
            - Updated/Inserts a pending message_request
        - GetDmInboxForUser
            - Uses idx_dm_convo_time to find the latest message per conversation
            - Join Pending requests to show whether a conversation is still waiting to be accepted
        - RespondToMessageRequest
            - Accept and the conversation continues
            - Reject and the conversation is cleaned up as well as messages
    - The DM system is now consent based and the accept/reject works.

- Queries use composite indexes

Vivek Reddy Bhimavarapu

1. ACID CONSTRAINTS

ACID transactions were required because when creating a quiz with multiple questions and answers, if something goes wrong halfway through (like a network error), you don't want a partially created quiz. ACID transactions ensure everything succeeds together or everything fails together.

Implementation in StudyBuddy:
User submits form with: quiz title, description, questions, and answers.
`conn.start_transaction()` is used to ensure the entire quiz creation process is atomic.

Real-World Example:
If inserting quiz → success, question 1 → success, question 2 → fails, then the entire quiz is rolled back so nothing is partially saved.

2. Parameterized SQL Queries

SQL Injection is a security vulnerability where bad actors insert malicious SQL code into input fields.

Dangerous Example (DON'T DO THIS):
```
email = request.get_json()["email"]
```

Safe Example (WHAT WE DO):
```
email = request.get_json()["email"]
```

The database treats the input as **DATA**, not SQL code. Even if user enters `' OR '1'='1`, it's treated as a literal string to search for.

Where We Use This:

- Login:
  ```
  cursor.execute("SELECT ... WHERE email = %s", (email,))
  ```

- Quizzes:
  ```
  cursor.execute("INSERT INTO Quiz ... VALUES (%s, %s, %s, %s)",
  ```

```
(title, desc, course_id, creator_id))
```

- Flashcards:
```
cursor.execute("INSERT INTO flashcardset ... VALUES (%s, %s, %s, %s)", (title, desc, course_id, creator_id))
```

- All user data updates

3. Proper Error Handling

Transaction Error Handling Pattern:

```
try:
    # db ops
except:
    rollback
finally:
    close
```

What This Does:

- **try:** Attempts database operation
- **except:** If error occurs, rolls back all changes
- **finally:** Always closes connections (prevents resource leaks)

Example Flow:

- Quiz creation starts
- Insert quiz: Success ✓
- Insert question 1: Success ✓
- Insert question 2: FAILS ✗
- Exception caught

- Rollback executed → removes quiz and question 1
- Connection closed
- Error returned to user

4. Database Constraints

Foreign Key Constraints
NOT NULL Constraints
Data Type Validation:

- Points are numbers, not text
- IDs are integers
- Descriptions have max length

These prevent invalid data from entering the database and enforce referential integrity.

5. Summary of Benefits

ACID transactions, parameterized queries, error handling, and constraints make StudyBuddy reliable and safe for students to use.