Databases

1 A Codebase of... Codebase?

Create a data file of all Codebase members who joined this semester in any format and title it `members`. Then, from that file create a table that includes the following information:

- Name
- Year
- Team
- Role

Submit: the contents of your `members` data file, the `CREATE TABLE` and `INSERT` (or `.import`) statements used to build the table, and a screenshot of the populated `members` table.

2 Projects

Create a `projects` table (at least 6 rows) with the following columns: `project_id`, `project_name`, `status`.

The `project_name` column should use the same values as `members.Team` (e.g. SonySideUp, Big Ds) so that it serves as the join key between the two tables.

Write the following queries:

(a) Perform a `LEFT JOIN` from `members` to `projects` on `members.Team = projects.project_name`. Return each member's `Name`, `Role`, and `project_name`, ordered by `Name`.

(b) Group by `project_name` and return the team name, the number of members on that team, and the number of projects with that name. Use `LEFT JOIN` in both directions so that teams with no projects (and projects with no members) still appear.

(c) Return the `project_name` values whose corresponding team has exactly 6 members. You may not hardcode any team names—use a subquery or `HAVING` clause to derive the threshold dynamically.

Submit: the `INSERT` statements used to populate `projects`, all three queries, and a screenshot of each result.

3 Events

Codebase runs many events each semester (socials, workshops, hackathons, etc.), and members can attend multiple events. Model this many-to-many relationship with two new tables.

(a) Write a `CREATE TABLE` statement for an `events` table with the following columns:

- A primary key column
- `event_name` (must be `NOT NULL` and `UNIQUE`)
- `event_date`
- `event_type` (e.g., social, workshop, hackathon)

For each constraint you add, write 1–2 sentences justifying why it is necessary.

(b) Write a `CREATE TABLE` statement for an `attendance` junction table that:

- Has foreign keys referencing both `members` and `events`
- Uses a composite primary key on `(member_id, event_id)` to prevent duplicate attendance records
- Includes a `checked_in_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP` column

(c) Write two `INSERT` statements that are intentionally rejected by the database:

- One that violates the `NOT NULL` or `UNIQUE` constraint on `events`
- One that violates the composite primary key constraint on `attendance`

Paste the full error message returned by your database for each.

Submit: the `CREATE TABLE` statements for both tables, the two failing `INSERT` statements with their error messages, and a screenshot of your schema.

4 Graduation

At the end of each semester, graduating seniors are removed from the club roster. Both their row in `members` and all of their rows in `attendance` must be deleted together—leaving orphaned attendance records would violate referential integrity and corrupt the event history.

(a) Write a transaction that performs both deletions atomically. Your transaction must use `BEGIN`, a `SAVEPOINT` created after the first `DELETE`, and either `COMMIT` or `ROLLBACK`.

(b) Simulate a failure between the two `DELETE` statements (for example, by attempting to delete from a table that does not exist). Show the state of the database:

- Before the transaction begins
- After the error occurs (mid-transaction)
- After the `ROLLBACK`

Provide a screenshot for each of the three states.

(c) Answer the following questions in 2–4 sentences each:

- **Atomicity:** Why is it important that both the `members` deletion and the `attendance` deletion either both succeed or both fail? What could go wrong if only one of the two deletions were committed?
- **Isolation:** If another user queries the `members` table while your transaction is running—after the first `DELETE` but before the `COMMIT`—what will they see, and why?

Submit: the SQL for parts (a) and (b) with accompanying screenshots, and written answers for part (c).

Contributors

- Daniel Wang