

## **LAB EXAM-04**

**NAME:CHANDA HARINI**

**ROLL NO.:2403A510E1**

**BATCH NO.:05**

**1)A student course registration system must enforce prerequisites and seat limits.**

**a) Design schema including constraints and relations.**

**PROMPT:**

```
You are an SQL expert. I am providing SQL code that creates database tables.
```

1. Check the syntax for errors.
2. Show the expected output messages after running it.
3. Display the structure of each table in a tabular format (columns, datatype, constraints).

## CODE:

```
> Users > Praneeeth Cheekati > Downloads > sunny > labtest4.1.sql > ...
1  CREATE TABLE Students (
2    student_id INT PRIMARY KEY,
3    student_name VARCHAR(100) NOT NULL
4 );
5
6
7  CREATE TABLE Courses (
8    course_id INT PRIMARY KEY,
9    course_name VARCHAR(100) NOT NULL,
10   seat_limit INT NOT NULL
11 );
12
13
14 CREATE TABLE CoursePrerequisites (
15   course_id INT,
16   prerequisite_id INT,
17   PRIMARY KEY (course_id, prerequisite_id),
18   FOREIGN KEY (course_id) REFERENCES Courses(course_id),
19   FOREIGN KEY (prerequisite_id) REFERENCES Courses(course_id)
20 );
21
22 CREATE TABLE Registrations (
23   reg_id INT PRIMARY KEY,
24   student_id INT,
25   course_id INT,
26   status VARCHAR(20) CHECK (status IN ('Confirmed', 'Waiting')),
27   FOREIGN KEY (student_id) REFERENCES Students(student_id),
28   FOREIGN KEY (course_id) REFERENCES Courses(course_id)
29 );
```

## OUTPUT:

```
sql

Table "Students" created successfully.
Table "Courses" created successfully.
Table "CoursePrerequisites" created successfully.
Table "Registrations" created successfully.
```

## OBSERVATION:

The SQL script successfully created four relational tables — **Students**, **Courses**, **CoursePrerequisites**, and **Registrations** — establishing the structure of a student course registration system.

The **primary keys**, **foreign keys**, and **CHECK constraints** were applied correctly, ensuring **data integrity** and **relationship enforcement**.

## b) Write AI-assisted SQL to list students waiting for enrolment confirmation

### PROMPT:

```
You are an SQL expert. Write an SQL query to list all students who are on the waiting list for course enrollment.  
Tables: Students(student_id, student_name), Courses(course_id, course_name), Registrations(reg_id, student_id, course_id, status).  
status = 'Waiting' means waiting list.  
Return student_name, course_name.
```

### CODE:

```
1   SELECT s.student_name, c.course_name  
2   FROM Students s  
3   JOIN Registrations r ON s.student_id = r.student_id  
4   JOIN Courses c ON r.course_id = c.course_id  
5   WHERE r.status = 'Waiting';  
6   |
```

### OUTPUT:

#### Sample Output

student_name	course_name
Aditya	Data Structures
Kiara	Machine Learning
Rohan	Operating Systems

## OBSERVATION:

---

### Observation

- The query joins `Students`, `Registrations`, and `Courses` to fetch names instead of IDs.
- Only records where `status = 'Waiting'` are selected, meaning:
  - These students have registered for the course
  - But their enrollment is **not yet confirmed due to seat limits or unmet prerequisites**.
- The output helps the administration identify pending registrations and allocate seats when available.

## 2) AI suggests denormalizing the database for fast reads.

### a) Evaluate pros & cons based on scenario.

### PROMPT:

You are a database optimization expert. Evaluate the pros and cons of denormalizing a student course registration system that currently has normalized tables (`Students`, `Courses`, `CoursePrerequisites`, and `Registrations`). The system must enforce prerequisites and seat limits but also needs fast read performance for dashboards and reports.

## CODE AND OUTPUT:

### Pros of Denormalization (Adding Count Columns to Courses):

- Significantly Faster Reads:** This is the primary advantage. To find out how many seats are available in a course, you can simply query `SELECT seat_limit - enrolled_count FROM Courses WHERE course_id = :id`. This is much faster than running an aggregate query like `SELECT COUNT(*) FROM Registrations WHERE course_id = :id AND status = 'Confirmed'`, especially as the `Registrations` table grows to millions of rows. Displaying a full course catalog with current enrollment numbers becomes trivial and extremely fast.
- Simplified Read Queries:** Queries for fetching course lists with their enrollment status become much simpler. Instead of a `JOIN` and `GROUP BY` across `Courses` and `Registrations`, it's a direct `SELECT` from the `Courses` table. This can make application-level code cleaner and less prone to errors.
- Reduced Database Load:** For read-heavy workloads (e.g., thousands of students browsing courses simultaneously), the database is spared from performing thousands of expensive aggregation calculations. This reduces CPU and I/O load, allowing the database to serve more users concurrently.

### Cons of Denormalization:

- Increased Write Complexity:** Every action that changes a registration status (enrolling, withdrawing, moving from waitlist to confirmed) now requires an additional `UPDATE` statement on the `Courses` table. This logic must be handled carefully by the application.
- High Risk of Data Inconsistency:** This is the most significant drawback. If the application successfully inserts a row into `Registrations` but fails to update the `Courses` table (due to a bug, network issue, or server crash), the counts will become incorrect. This can lead to critical failures, such as preventing a student from enrolling in a course that has open seats, or over-enrolling a course.
- Concurrency Bottlenecks:** During peak registration times, a single popular course might have hundreds of students trying to register simultaneously. Each registration would attempt to lock and update the same row in the `Courses` table, creating a "hot spot" and a potential performance bottleneck for write operations.
- Data Redundancy:** The count columns store information that can be derived from the `Registrations` table. This uses more storage, but more importantly, it creates two sources of information that can potentially contradict each other.

## OBSERVATION:

The AI correctly identifies the classic trade-off of denormalization: gaining read speed at the cost of write complexity and data integrity. It accurately applies this general concept to the specific scenario of the course registration system. The analysis is sound, highlighting that while fetching course availability would be faster, the risk of data becoming inconsistent is a major concern that could break the system's core business logic.

## b) Decide final approach and justify.

### PROMPT:

◊ Connect to MSSQL

Write SQL to maintain a normalized OLTP registration system and create a denormalized reporting table for fast reads. Include a procedure to refresh the reporting table periodically.

|

## CODE:

```
1  -- Connect to MSSQL
2  -- Denormalized Reporting Table
3  CREATE TABLE RegistrationReports AS
4  SELECT
5      r.reg_id,
6      s.student_name,
7      c.course_name,
8      r.status
9  FROM Registrations r
10 JOIN Students s ON r.student_id = s.student_id
11 JOIN Courses c ON r.course_id = c.course_id;
12
13 -- Procedure to refresh reporting table
14 CREATE OR REPLACE PROCEDURE RefreshRegistrationReports AS
15 BEGIN
16     DELETE FROM RegistrationReports;
17     INSERT INTO RegistrationReports
18     SELECT
19         r.reg_id,
20         s.student_name,
21         c.course_name,
22         r.status
23     FROM Registrations r
24     JOIN Students s ON r.student_id = s.student_id
25     JOIN Courses c ON r.course_id = c.course_id;
26 END;
27 /
```

## OUTPUT:

reg_id	student_name	course_name	status
101	Aditya	DBMS	Confirmed
102	Kiara	ML	Waiting
103	Rohan	OS	Confirmed

## OBSERVATION:

The AI's final recommendation is excellent and reflects a mature architectural decision.

1. **Correct Prioritization:** It correctly prioritizes data integrity over convenience, which is crucial for a transactional system like this.
2. **Robust Solution:** Instead of a simple "yes" or "no," it proposes a hybrid solution that addresses the cons of denormalization. Using a database trigger is the standard and most robust way to solve this exact problem. It guarantees that the `Courses` counts and the `Registrations` table data can never go out of sync.
3. **Complete Code:** The provided Oracle SQL trigger is comprehensive. It correctly handles all three database operations (`INSERT`, `DELETE`, `UPDATE`) and the different status transitions, ensuring the counts are always accurate. This is a production-quality solution to the problem.

This approach provides the best of both worlds: a normalized, consistent source of truth, and denormalized, pre-calculated values for high-performance reads.

