COL 362/632 Assignment 2
Database Design

7 Feb, 2024 (version 1.1)

Version History

Revision	Date	Author(s)	Description
1.0	07 Feb 2024	Instructor/TA	Initial Version
1.1	11 Feb 2024	Instructor/TA	Fixed FK-PK joins and Q4 in section 2.1 [text in blue color]
1.2	13 Feb 2024	Instructor/TA	Fixed credit limit in question 2.2.3 and Q4 in section 2.1 [text in red color]

Deadline

The assignment is due on **16 Feb 2024, 11:59 PM**. All submissions are to be made on Moodle.

General Instructions

Follow all instructions. Submissions not following these instructions will not be evaluated and will be given Zero marks.

1. Kindly ensure that this assignment is completed independently. Collaboration with any external entities, including individuals, AI agents, websites, discussion forums, etc., is strictly prohibited. You are free to discuss and post questions on the piazza to seek clarification.
2. You must use **PostgreSQL 15.x** and use **PLPGSQL** for writing functions and procedures for this assignment.
3. **Marking Scheme:** The assignment will be evaluated using a test set of DDL statements, for which your database design must uphold all the specified constraints. Each test set carries different weights based on the perceived difficulty level of the usecase. These weights are not revealed to you (**intentionally**). The weights will be released after evaluation.
4. Note that we will evaluate your assignment automatically, so take special care of following folder names, file names, and directory structure adhering to instructions. Also, we will do binary binary grading for each test case; there will be no partial grading.
5. **Note that there will be no deadline extensions.** But, you can use your three skip days. Refer to lecture notes on course organization for skip-day rules.
6. A single zip has to be submitted. The zip should be structured such that
   - Upon deflating, a single submission file should be under a directory with the student's registration number. For, suppose a student's registration number is 22XXCSXX123. In that case, the zip submission should be named 22XXCSXX123.zip, and upon deflating **a single `.sql` file** should be under the directory named `./22XXCSXX123` only (names should be in uppercase) - your submission might be rejected and not be evaluated if you do not adhere to these specifications.
   - In this directory, create a single `.sql` file with the name **`ASSIGNMENT_2.sql`** (name of file should be in uppercase). NOTE: All the SQL queries (DDL statements) should be written in the same single file.
   - The final directory structure will look like below:

```
22XXCSXX123
└── ASSIGNMENT_2.sql
```

Designing a database for course administration

Your objective in this assignment is to design a database for course administration purposes. After going through the requirements, ER modeling, and discussions with the Institute management, it was decided that the database must have the following tables.

Table Name	Description
student	Stores the basic details of students
courses	Stores the details of courses offered in IITD
student_courses	Stores details about the courses taken by students
course_offers	Stores details about which semester offers which course
professor	Stores the basic details of professors
valid_entry	Stores the current intake cycle/ useful for giving entry_id to students
department	Stores the details of departments in IITD

Table 2: All Tables

Column Name	Data Type	Description
first_name	VARCHAR(40)	
last_name	VARCHAR(40)	
student_id	CHAR(11)	Primary Key
address	VARCHAR(100)	
contact_number	CHAR(10)	
email_id	VARCHAR(50)	
tot_credits	INTEGER	
dept_id	ref Table 9	Foreign Key

Table 3: student (tot_credits represents total credits taken by a student and is not dependent on grade. Simply, it is the sum of credits in student_courses table grouped by student_id.)

Column Name	Data Type	Description
course_id	CHAR(6)	Primary Key
course_name	VARCHAR(20)	
course_desc	TEXT	
credits	NUMERIC	
dept_id	ref Table 9	Foreign Key

Table 4: courses

Column Name	Data Type	Description
student_id	ref Table 3	Foreign Key
course_id	ref Table 6	Foreign Key
session	ref Table 6	Foreign Key
semester	ref Table 6	Foreign Key
grade	NUMERIC	

Table 5: student_courses (Here [course_id, session, semester] combined act as a Foreign Key to Table 6)

Column Name	Data Type	Description
course_id	ref Table 4	Foreign Key, Primary Key
session	VARCHAR(9)	Primary Key
semester	INTEGER	Primary Key
professor_id	ref Table 7	Foreign Key
capacity	INTEGER	
enrollments	INTEGER	

Table 6: course_offers (Here [course_id, session, semester] combined act as a Primary Key)

Column Name	Data Type	Description
professor_id	VARCHAR(10)	Primary Key
professor_first_name	VARCHAR(40)	
professor_last_name	VARCHAR(40)	
office_number	VARCHAR(20)	
contact_number	CHAR(10)	
start_year	INTEGER	
resign_year	INTEGER	
dept_id	ref Table 9	Foreign Key

Table 7: professor

Column Name	Data Type	Description
dept_id	ref Table 9	Foreign Key
entry_year	INTEGER	
seq_number	INTEGER	

Table 8: valid_entry

Column Name	Data Type	Description
dept_id	CHAR(3)	Primary Key
dept_name	VARCHAR(40)	

Table 9: department

1 Single Table Constraints

Your first task as a database designer is to create an initial schema for all the tables with attributes and their abovementioned types. Adhere to the **Primary Key** and **Foreign Key** constraints, and use the same names for the tables and the columns. Additionally, add the database requirements calls for the following single table **Not NULL**, **Unique**, and **check** constraints:

1.1 Not NULL and Unique Constraints:

Column Name	Constraint
first_name	NOT NULL
student_id	NOT NULL
contact_number	NOT NULL, UNIQUE
email_id	UNIQUE
tot_credits	NOT NULL

Table 10: student

Column Name	Constraint
course_id	NOT NULL
course_name	NOT NULL, UNIQUE
credits	NOT NULL

Table 11: courses

Column Name	Constraint
grade	NOT NULL

Table 12: student_courses

Column Name	Constraint
semester	NOT NULL

Table 13: course_offers

Column Name	Constraint
professor_first_name	NOT NULL
professor_last_name	NOT NULL
contact_numner	NOT NULL

Table 14: professor

Column Name	Constraint
entry_year	NOT NULL
seq_number	NOT NULL

Table 15: valid_entry

Column Name	Constraint
dept_name	NOT NULL, UNIQUE

Table 16: department

1.2 Check Constraints:

1. **tot_credits** in the **student** table should be greater than or equal to 0.
2. **credits** in the **courses** table should be greater than 0.
3. **grade** in **student_courses** should be between 0 and 10 (inclusive).
4. **start_year** should be less than or equal to **resign_year** in the **professor** table.
5. **semester** should either be 1 or 2 in **course_offers** and **student_courses** tables.
6. **course_id** must have first 3 characters as some **dept_id** and next 3 characters must be digits.

2 Supporting Advanced Constraints

The requirement analysis also revealed certain behaviors and integrity the database must uphold when deployed. For this, you are required to create appropriate **views**, **functions and procedures**, and **triggers** that allow the database to behave as expected.

2.1 Modifications to Student Table

1. When a new student is registered, a unique **student_id** is assigned to each student. A **student_id** is a 10-digit unique code, with the first four digits being **entry_year**, the next three characters are **dept_id**, and the last three digits are **seq_number**. When a new student is registered, your schema must validate this **entry_number** with the below conditions:
   - The **entry_year** and **dept_id** in **student_id** should be a valid entry in **valid_entry** table.
   - The sequence number should start from 001 for each department (maintained in **valid_entry** table). Thus, the current sequence number is assigned when a new student is registered in a department.

   Create a trigger with the name of **validate_student_id** to validate the **student_id**. If the entry_number assigned to a student is not valid, then raise an **"invalid"** message; else, successfully insert the tuple in the table.

   Example: Below is an instance of the **valid_entry** and **student** table.

dept_id	entry_year	seq_number
CSZ	2020	1
CSY	2024	3

Table 17: Entry in **valid_entry** table

first_name	last_name	student_id	address	contact_number	email_id	tot_credits	dept_id
xxxx	xxxx	2024CSY001	xxxx	xxxxxxxxxx	2024CSY001@CSY.iitd.ac.in	0	CSY
xxxx	xxxx	2024CSY002	xxxx	xxxxxxxxxx	2024CSY002@CSY.iitd.ac.in	10	CSY

Table 18: Valid **student** table

Below are the valid 19 and invalid 20 tuples for the **student** table. The correct entry should be inserted in the table, while the invalid entry should return the ”invalid” message.

first_name	last_name	student_id	address	contact_number	email_id	tot_credits	dept_id
xxxx	xxxx	2020CSZ001	xxxx	xxxxxxxxxx	2020CSZ001@CSZ.iitd.ac.in	0	CSZ

Table 19: Valid entry in **student** table

first_name	last_name	student_id	address	contact_number	email_id	tot_credits	dept_id
xxxx	xxxx	2021CSZ001	xxxx	xxxxxxxxxx	2020CSZ001@CSZ.iitd.ac.in	0	CSZ
xxxx	xxxx	2020CSZ002	xxxx	xxxxxxxxxx	2020CSZ001@CSZ.iitd.ac.in	0	CSZ
xxxx	xxxx	2020CS001	xxxx	xxxxxxxxxx	2020CSZ001@CSZ.iitd.ac.in	0	CSZ
xxxx	xxxx	2020CSY005	xxxx	xxxxxxxxxx	2020CSZ001@CSZ.iitd.ac.in	0	CSZ

Table 20: Invalid entry in **student** table [incorrect **student_id**]

2. If the above **`student_id`** is a valid id, you add that student detail in the **`student`** table. But do not forget to increase the counter, i.e., **`seq_number`** in **`valid_entry`** table after each insert in the student table. Thus, create a trigger with the name, **`update_seq_number`**, which will update the **`seq_number`** in **`valid_entry`** table.

   Example: Once the valid student is inserted in the **`student`** table as shown in 19, there should be an update in **`valid_entry`** table. The correct update based on the above-given instance for **`valid_entry`** table is shown below:

dept_id	entry_year	seq_number
CSZ	2020	2
CSY	2024	3

Table 21: Valid update in **valid_entry** table

3. Assume that before we perform an insert operation on the **`student`** table, we need to verify if the student's **`email_id`** is correct or not. A correct **`email_id`** will be of the form '`YYYYABC123@ABC.iitd.ac.in`', i.e., it has two parts, one part before @ and other after it. The part before @ should match the **`student_id`** for example, '`YYYYABC123`', where the first four digits being **`entry_year`**, the next three characters are department ID (**`dept_id`**), and the last three digits are sequence number (**`seq_number`**). The second half of the email (after the '@') should start with the department ID of the **`student`** (`ABC` in this case). This should match the department ID, i.e., the three characters in the **`student_id`** column of the student as well as the department ID in the **`dept_id`** column and end with '`.iitd.ac.in`'. Validate if the student's email is correct or not. If the email is valid, continue with the insertion; otherwise, raise an **`"invalid"`** message.

   Example: Below are the valid and invalid entries for **`email_id`** for **`student`** table.

first_name	last_name	student_id	address	contact_number	email_id	tot_credits	dept
xxxx	xxxx	2017ABC001	xxxx	9876543210	2017ABC001@ABC.iitd.ac.in	0	ABC

Table 22: An example valid entry for **student** table

first_name	last_name	student_id	address	contact_number	email_id	tot_credits	dept
xxxx	xxxx	2017ABC001	xxxx	9876543210	2020ABC001@ABC.iitd.ac.in	0	ABC

Table 23: An example invalid entry for **student** table [incorrect **email_id**]

4. (To allow or to not allow change of branch) The Institute management also wants to study the branch change statistics. For this, your schema must include an additional table **`student_dept_change`** in your

schema that maintains a record of students that have changed their department consisting of **old_student_id**, **old_dept_id**, **new_dept_id**, and **new_student_id** (both **old_dept_id** and **new_dept_id** must be Foreign key referring to department table). Write a single trigger (name **log_student_dept_change**) that calls a function upon updating the student table. The function should do as follows: Before the update, if the update is changing the student's department, check if their department was updated before from **student_dept_change** table; if yes, raise an exception "`Department can be changed only once`" (every student can only change their department once). If the department has not changed before and the entry year (entry year can be extracted from **student_id**) is less than 2022, Raise an exception: "`Entry year must be >= 2022`". Only students who entered in 2022 or later can change their department. Further, check whether the average grade of the student is > 8.5 or not (from **student_courses** table) if the average grade of the student is $<= 8.5$ or the student has done no courses so far raise an exception "Low Grade". If all conditions are met, perform the update, and after the update, insert a row into the **student_dept_change** table.
Note: While assigning the new **student_id** you have to check the **seq_number** in the **valid_entry** table to assign the valid **student_id**. Also, do not forget to increase the counter, i.e., **seq_number** in **valid_entry** table after updating the **student_id**. Also, you have to update the corresponding valid **email_id** in the **student** table.

2.2 Modifications to student_courses table

1. Evaluating the recently concluded course is essential for planning and execution of the same course in the future. It is imperative to maintain a **view** which provides an average, min, and max grade for a particular course whenever there is a change (insert and update of a tuple) in **student_course** table. Such a view should contain the following columns and **must be up to date at all times**:
Note: The name of the view should be **course_eval**.

course_id	session	semester	number_of_students	average_grade	max_grade	min_grade

Table 24: Name of the columns in view (**course_eval**)

2. Create a trigger which updates the **student** table's **tot_credits** column each time an entry is made into the **student_courses** table. Each time an entry for a student pursuing any course is made in the **student_courses** table, the following is expected.
Given the entry that is to be inserted into the **student_courses** table, use the **course_id** and the **courses** table to get the number of credits for that course. Now that you know the credits for this course, update that particular student's **tot_credits** and add the credits for this new course in the **student** table.

3. Implement a trigger that ensures that a student is not enrolled in more than 5 courses simultaneously(in the same session and same semester) in the **student_courses** table. Also, check that while adding entries into **student_courses** table, the credit criteria for the student (maximum 60 total credits for every student) should not be exceeded. If the maximum course criteria or the maximum credit criteria are breached, raise an **"invalid"** exception; else, continue with the update.
Note: You can use the **tot_credits** column from table student.60 is the total credit limit for every student across all records, across all semesters and across all sessions. No student should surpass this limit.

4. Assume that we are trying to insert a record into the **student_courses** table. Write a trigger which uses **course_id** as the foreign key and makes sure that any course of 5 credits is taken up by the student in the student's first year only.(You can know the student's first year since the **student_id** begins with the year of their admission; compare this with the first four digits of the session of the course, which is usually of the form 2023-2024). If the entry is for a 5-credit course and is not in the first year of the student, Raise an **"invalid"** exception; else, insert the entry into the table. Any entry with a course with less than 5 credits should be added.

5. Create a view **student_semester_summary** from **student_courses** table which contains the **student_id**, **session**, **semester**, **sgpa**, **credits**. This view stores the students' details for a semester. Calculate sgpa (as done at IITD) as

$$\frac{\text{grade\_points\_secured\_in\_courses\_with\_grade\_greater\_than\_or\_equal\_to\_5.0}}{\text{earned\_credits\_in\_courses\_with\_grade\_greater\_than\_or\_equal\_to\_5.0}}$$

where courses and earned_credits should correspond to the semester and session. $grade\_points$ for a course is the product of grade secured in that course and credits of the course as calculated at IITD! You can interpret grades greater than or equal to 5 as pass grades. Ignore failed courses from sgpa calculation. The credits in the view corresponds to the credits completed (credits of courses with pass grade) in that semester. Whenever a new row is added to **student_courses** table update the **student_semester_summary** view, as well as **tot_credits** in **student** table. Also, add the course only if the **credit_count** doesn't exceed the limit of 26 per semester. When the grade for a course is updated in the **student_courses**, update the **sgpa** in the view. When a row is deleted from **student_courses** table, update the **credits** and **sgpa** in the view as well as update **tot_credits** in **student** table.

6. Write a single trigger on insert into **student_courses** table. Before insertion, check if the capacity of the course is full from the **course_offers** table; if yes raise an "course is full" exception; if it isn't full, perform the insertion, and after insertion, update the no. of enrollments in the course in **course_offers** table.

2.3 Modifications to course_offers table

1. Whenever a course is removed from the **course_offers** table, it should also update the **student_courses** table such that all the student entries for that course (in that particular session and semester) should be removed. The **tot_credits** in the **student** table for the removed students should also be updated. If the course is added, ensure that **course_id** is present in the **courses** table and **professor_id** is present in the **professor** table.

2. Given an entry that is to be inserted into the **course_offers** table, create a trigger that makes sure that a professor does not teach more than 4 courses in a session. Also make sure that the course is being offered before the associated professor resigns. If in any case the entry is not valid show an **"invalid"** message or else insert the entry into the table.

2.4 Modifications to department table

1. Write a single trigger using which on update on the department table, if **dept_id** is updated, updates all course ids of the courses belonging to that department according to the new **dept_id** in **course_offers**, **courses** and **student_courses** tables (i.e update the first three digits of the **course_id** according to new **dept_id**), also update it in **professor** and **student** tables. On delete, before deletion, check if there are no students in the department, if there are students show a "Department has students" message, else delete the department record and further delete all courses from **course_offers**, **courses** tables and professors in that department from **professor** table.