Homework 6

Farnoosh Moshirfatemi

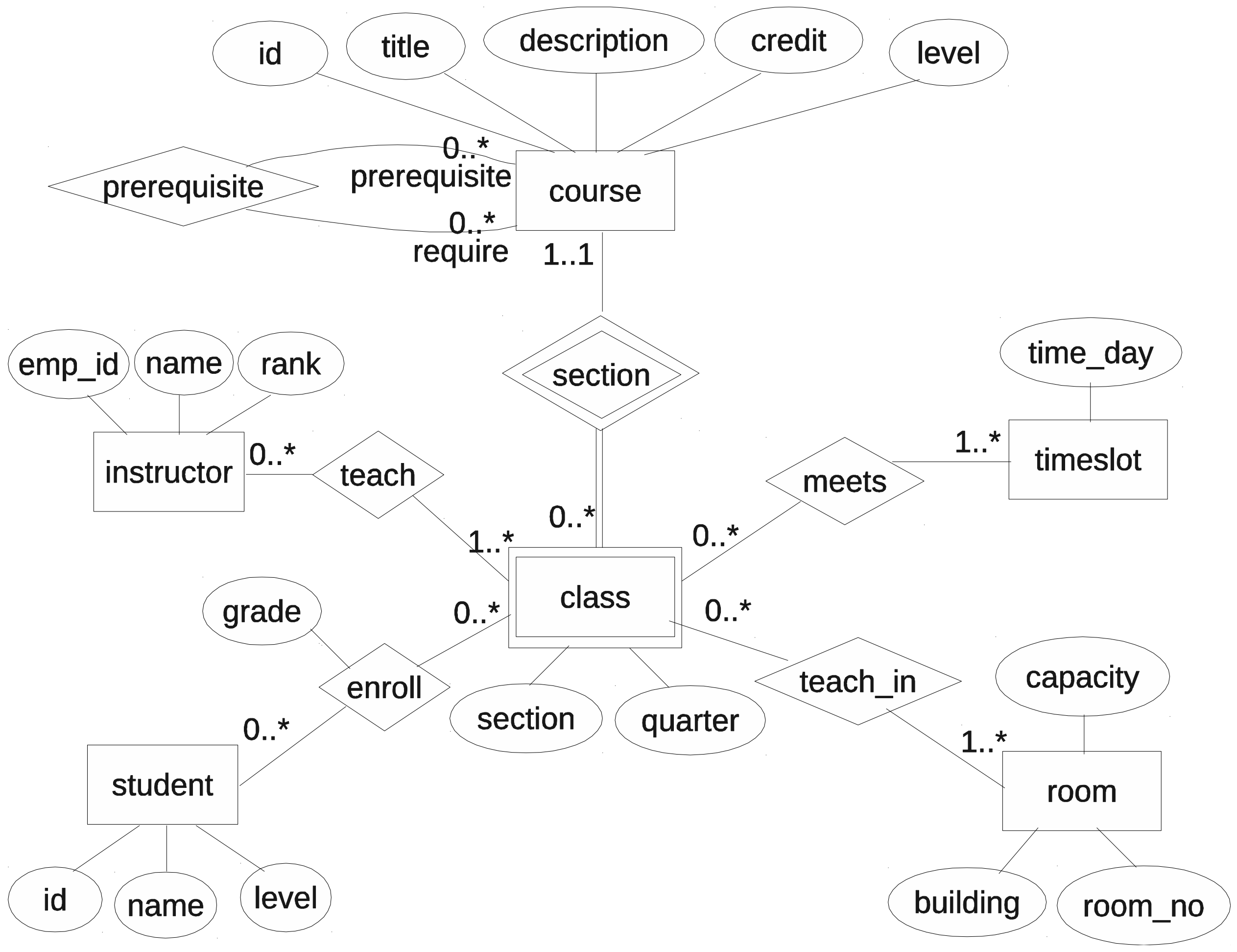
Dejun Qian

Question 1:

Database design problem

Answer:

The ER diagram of the database is shown below.



By translating the ER diagram, we get the following set of tables. The tables are described using create table statements.

Table 1. Create table statements for the database

table name	create table statement
course	CREATE TABLE course (id CHARACTER(22) PRIMARY KEY, title CHARACTER (20) NOT NULL, description CHARACTER (20), credit INTEGER, level CHARACTER(2));
prerequisit	CREATE TABLE prerequisite (course_id CHARACTER(22) REFERENCES course(id), precourse_id CHARACTER(22) REFERENCES course(id), PRIMARY KEY(course_id, precourse_id));
class	CREATE TABLE class (id CHARACTER(22) REFERENCES course(id), section INTEGER, quarter CHARACTER (22), PRIMARY KEY(id, section, quarter));
instructor	CREATE TABLE instructor (emp_id INTEGER PRIMARY KEY, name CHARACTER(20), rank INTEGER);
teach	CREATE TABLE teach (id CHARACTER(22), section INTEGER , quarter CHARACTER (22), emp_id INTEGER REFERENCES instructor(emp_id), PRIMARY KEY(id, section, quarter, emp_id), FOREIGN KEY(id, section, quarter) REFERENCES class(id, section, quarter));
student	CREATE TABLE student (id INTEGER PRIMARY KEY, name CHARACTER(22), level CHARACTER(1));

enroll	CREATE TABLE enroll (course_id CHARACTER(22), section INTEGER, quarter CHARACTER (22), student_id INTEGER REFERENCES student(id), grade CHARACTER(1), PRIMARY KEY(course_id, section, quarter, student_id), FOREIGN KEY(course_id, section, quarter) REFERENCES class(id, section, quarter));
room	CREATE TABLE room (building CHARACTER(20), room_no INTEGER, capacity INTEGER, PRIMARY KEY(building, room_no));
teach_in	CREATE TABLE teach_in (id CHARACTER(22), section INTEGER, quarter CHARACTER (22), room_no INTEGER, building CHARACTER(20), PRIMARY KEY(id, section, quarter, room_no, building), FOREIGN KEY(room_no, building) REFERENCES room(room_no, building), FOREIGN KEY(id, section, quarter) REFERENCES class(id, section, quarter));
timeslot	CREATE TABLE timeslot (time_day CHARACTER(20) PRIMARY KEY);
meets	CREATE TABLE meets (id CHARACTER(22), section INTEGER, quarter CHARACTER (22), time_day CHARACTER(22) REFERENCES timeslot(time_day), PRIMARY KEY(id, section, quarter, time_day), FOREIGN KEY(id, section, quarter) REFERENCES class(id, section, quarter)) ;

For the constraints listed, none of them could be fully supported by a foreign key constraint. Since we wanted the constraints to be enforced at all time, and not only before or after some triggering event, we implemented all of the constraints with assertions and not triggers. We preferred to use SQL statement instead of embedded SQL program, because this way we don't to write any C or Java code. SQL statement is at higher level as C or Java.

Therefore, we need to create assertions for each of them.

The results are shown below.

1-A room cannot be booked for more than one class (section) at the same time slot.

For this constraint, we wrote an assertion to check there are no rows in the table where the number of reservation for a room and a time slot is bigger than 1. The constraint is shown below.

```
CREATE ASSERTION question1 CHECK(NOT EXISTS
 (SELECT *
 FROM ( SELECT count(*) AS C
 FROM timeslot, meets, teach_in, room
 WHERE timeslot.time_day = meets.time_day AND
 meets.id = teach_in.id AND
 meets.section = teach_in.section AND
 meets.quarter = teach_in.quarter AND
 teach_in.building = room.building AND
 teach_in.room_no = room.room_no
 group by timeslot.time_day, room.building, room.room_no) x
 WHERE C > 1
 )
```

2-The number of students enrolled in a class cannot exceed the capacity of the room assigned to the class.

Similar to the previous constraint, we wrote an assertion to check there are no rows in the table where the number of students enrolled in a class exceeds the capacity of the room assigned to the class. The constraint is shown below.

```
CREATE ASSERTION question1 CHECK(NOT EXISTS
 (SELECT enroll.course_id, enroll.quarter, enroll.section, room.building, room.room_no
 FROM student, enroll, teach_in, room
 WHERE student.id = enroll.student_id AND
 enroll.course_id = teach_in.id AND
 enroll.section = teach_in.section AND
 enroll.quarter = teach_in.quarter AND
 teach_in.building = room.building AND
 teach_in.room_no = room.room_no
 group by enroll.course_id, enroll.quarter, enroll.section, room.building, room.room_no
 having count(*)>room.capacity)
 )
```

3- Students may not take more than 21 credits in any one quarter. (PSU rules may be more flexible.)

For this constraint, we wrote an assertion to check there are no rows in the table where students take more than 21 credits in any one quarter. The SQL statement is shown below.

```
CREATE ASSERTION question1 CHECK(NOT EXISTS
 (SELECT class.quarter, enroll.student_id
 FROM  enroll, class, course
 WHERE
 enroll.course_id = class.id AND
 enroll.section = class.section AND
 enroll.quarter = class.quarter AND
 class.id = course.id
 group by class.quarter, enroll.student_id
 having sum(course.credit)>21)
 )
```

4-A student can't enroll in two sections at the same time slot in a quarter.

To enforce this constraint, we wrote an assertion to enforce there are no rows in the table where a student enrolls in more than one section at the same time slot in a quarter. The constraint is shown below.

```
CREATE ASSERTION question1 CHECK(NOT EXISTS
 (SELECT count(*) AS C
 FROM timeslot, meets, enroll, student
 WHERE timeslot.time_day = meets.time_day AND
 meets.id = enroll.course_id AND
 meets.section = enroll.section AND
 meets.quarter = enroll.quarter AND
 enroll.student_id = student.id
 group by student.id, timeslot.time_day, enroll.quarter
 having count(*)>1)
 )
```

5-A student can't enroll in two sections of the same class in a quarter.

This constraint is done by writing an assertion to enforce there are no rows in the table where a student enrolls in more than one section of the same class in a quarter. The SQL statement is shown below.

```
CREATE ASSERTION question1 CHECK(NOT EXISTS
 (SELECT enroll.student_id, class.id, class.quarter
 FROM  enroll, class
 WHERE class.id = enroll.course_id AND
 class.section = enroll.section AND
 class.quarter = enroll.quarter
 group by enroll.student_id, class.id, class.quarter
 having count(*)>1)
 )
```

6- A student can't enroll in a class unless they have earned a grade of C or higher in the prerequisite courses.

We enforce this constraint by writing an assertion to make sure there are no rows in the table where a student enrolls in a class when they have earned a grade less than C in the prerequisite courses. The following is the SQL statement for this constraint.

```
CREATE ASSERTION question1 CHECK(NOT EXISTS
 (SELECT *
 FROM  enroll e1, class, prerequisite, enroll e2
 WHERE class.id = e1.course_id AND
 class.section = e1.section AND
 class.quarter = e1.quarter AND
 class.id = prerequisite.course_id AND
 prerequisite.precourse_id = e2.course_id AND
 e2.grade < 'C')
 )
```

7- A student can't take the same course more than three times.

This constraint is done by writing an assertion to enforce there are no rows in the table where a student takes the same course more than three times. The SQL statement is shown below.

```
CREATE ASSERTION question1 CHECK(NOT EXISTS
  (SELECT count(*)
 FROM  enroll
 GROUP BY enroll.student_id, enroll.course_id
 having count(*)>3)
 )
```

Note: As it is shown, all of the constraints are written as assertion statements. Therefore, they are guaranteed to be true at all time

The following is the answer for the query questions.

Q1: Write a query that will show the class roster for each class.

The query statement is shown below.

```
SELECT enroll.course_id, enroll.section, enroll.quarter, student.id, student.name
 FROM  enroll, student
 where enroll.student_id = student.id
 GROUP BY enroll.course_id, enroll.section, enroll.quarter, student.name, student.id
ORDER BY enroll.course_id, enroll.section, enroll.quarter
```

Q2: Write a query that will show the transcript for each student.

The query statement is shown below.

```
SELECT student.id, enroll.course_id, enroll.section, enroll.quarter, enroll.grade
 FROM  enroll, student
 where enroll.student_id = student.id
 GROUP BY student.id, enroll.course_id, enroll.section, enroll.quarter, enroll.grade
```

Question 2:

Normalization problem

Answer:

1- Identify all of the non-trivial FDs that hold (based on the application).

mission_id -> mission_name

mission_id -> access_id

mission_id -> team_id

mission_id -> mission_status

mission_id -> team_name

mission_id -> meeting_frequency

team_id -> team_name

team_id -> meeting_frequency

2- Identify the key(s) for this table.

mission_id is the key for this new table.

3- Identify any "troublesome" FDs that prevent this table from being in BCNF.

A table is in BCNF if the left side of every nontrivial FD be a superkey. Therefore, in this table the troublesome FDs are:

team_id -> team_name

team_id -> meeting_frequency

4- Describe one insert anomaly, one update anomaly, and one delete anomaly that can arise with this table.

Insert anomaly:

If we want to insert different missions for one team, we need to know the descriptive information for that team (team_id, team_name, team_meeting_frequency) and make sure this information is in consistence with the other rows with the same team_id wich are already in the table.

Also, we have to insert the same information for all the rows that you want to be inserted with the same team_id which is providing the redundant information.

Also, if we want to insert a team information we can't until there is at least one mission assigned to that team.

Update anomaly:

If we want to change the name of a specific team, we have to change the name as many times as it is used in the rows of the table.

Delete anomaly:

If one team has only one mission, by deleting the mission the team information would also be deleted.

Also, if we want to delete a specific team from the table, we have to delete all the rows that have that team_id.

5- Given that the system designers have decided to use this table and knowing that there are redundancies, describe (in English) the triggers that would need to be implemented in order to correctly manage the redundancy. You want to make sure that a given piece of information (if it is represented redundantly) is always consistent. That is, you want to make sure that all of the copies of any redundant information have the same, most up-to-date value. You want to make sure that if information is deleted, all of the copies are deleted, etc.

Basically, we have to have triggers for all the mentioned anomalies.

<u>Trigger on update:</u>

If a team information (team_name or team_meeting_freq) wants to be updated in the table, the trigger should be executed after the update and it should execute the same update for all rows of the table with the same team_id.

<u>Trigger on insert:</u>

The trigger should be executed before the insertion and make sure that the desired information to be inserted are in consistence with the previous information in the table and if this is not the case avoid the insertion.

<u>Trigger on delete:</u>

In order to delete a team from the table, a trigger should be executed after the deletion and delete all other rows in the table which have the same team_id as the deleted team.