**1-First PART- Database Design Problem:**

CREATE TABLE course

(

id CHARACTER(22) PRIMARY KEY,

title character(20) NOT NULL,

description CHARACTER (20),

credit INTEGER,

level CHARACTER(2)

);

CREATE TABLE prerequisit

(

course\_id CHARACTER(22) REFERENCES course(id),

precourse\_id CHARACTER(22) REFERENCES course(id),

PRIMARY KEY(course\_id, precourse\_id)

);

CREATE TABLE class

(

id CHARACTER(22) REFERENCES course(id),

section INTEGER,

quarter CHARACTER (22),

PRIMARY KEY(id, section, quarter)

);

CREATE TABLE instructor

(

emp\_id INTEGER PRIMARY KEY,

name CHARACTER(20),

rank INTEGER

);

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)

);

CREATE TABLE student

(

id INTEGER PRIMARY KEY,

name CHARACTER(22),

level CHARACTER(1)

);

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)

);

CREATE TABLE room

(

building CHARACTER(20),

room\_no INTEGER,

capacity INTEGER,

PRIMARY KEY(building, room\_no)

);

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)

);

CREATE TABLE timeslot

(

time\_day CHARACTER(20) PRIMARY KEY

);

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)

)

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

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.**

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.)**

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.**

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.**

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.**

CREATE ASSERTION question1 CHECK(NOT EXISTS

(SELECT \*

FROM enroll e1, class, prerequisit, enroll e2

WHERE class.id = e1.course\_id AND

class.section = e1.section AND

class.quarter = e1.quarter AND

class.id = prerequisit.course\_id AND

prerequisit.precourse\_id = e2.course\_id AND

e2.grade < 'C')

)

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

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

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

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

??????Query2:

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

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

**2- SECOND PART. Normalization problem:**

**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 (team\_id, team\_name, team\_meeting\_frequency) and have to insert the same information for all the inserted rows which is redundant.

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.

**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.

Trigger on update:

If a team information (team\_name or team\_meeting\_freq) wants to be updated somewhere in the table, the trigger should executed after the update and execute the same update for all rows of the table with the same team\_id.

Trigger on insert:

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.

Trigger on delete:

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.