Exercise 3.6 What is a foreign key constraint? Why are such constraints important? What is referential integrity?

Answer 3.6 A *foreign key* constraint is a statement of the form that one or more fields of a relation, say R, together *refer* to a second relation, say S. That is, the values in these fields of a tuple in R are either *null*, or uniquely identify some tuple in S. Thus, these fields of R should be a (candidate or primary) key. For example, a student, uniquely identified by an *sid*, enrolled in a class must also be registered in the school's student database (say, in a relation called Students). Therefore, the *sid* of a legal entry in the Class_Enrollment relation must match an existing *sid* in the Students relation.

Foreign key constraints are important because they provide safeguards for insuring the integrity of data. Users are alerted/thwarted when they try to do something that does not make sense. This can help minimize errors in application programs or in data-entry.

Referential integrity means all foreign key constraints are enforced.

Exercise 3.7 Consider the relations Students, Faculty, Courses, Rooms, Enrolled, Teaches, and Meets_In defined in Section **??**.

1. List all the foreign key constraints among these relations.
2. Give an example of a (plausible) constraint involving one or more of these relations that is not a primary key or foreign key constraint.

Answer 3.7 There is no reason for a foreign key constraint (FKC) on the Students, Faculty, Courses, or Rooms relations. These are the most basic relations and must be free-standing. Special care must be given to entering data into these base relations.

In the Enrolled relation, *sid* and *cid* should both have FKCs placed on them. (Real students must be enrolled in real courses.) Also, since real teachers must teach real courses, both the *fid* and the *cid* fields in the Teaches relation should have FKCs. Finally, Meets_In should place FKCs on both the *cid* and *rno* fields.

It would probably be wise to enforce a few other constraints on this DBMS: the length of *sid*, *cid*, and *fid* could be standardized; checksums could be added to these identification numbers; limits could be placed on the size of the numbers entered into the credits, capacity, and salary fields; an enumerated type should be assigned to the grade field (preventing a student from receiving a grade of *G*, among other things); etc.

Exercise 3.8 Answer each of the following questions briefly. The questions are based on the following relational schema:

Emp(*eid:* `integer`, *ename:* `string`, *age:* `integer`, *salary:* `real`)
Works(*eid:* `integer`, *did:* `integer`, *pcttime:* `integer`)
Dept(*did:* `integer`, *dname:* `string`, *budget:* `real`, *managerid:* `integer`)

1. Give an example of a foreign key constraint that involves the Dept relation. What are the options for enforcing this constraint when a user attempts to delete a Dept tuple?
2. Write the SQL statements required to create the preceding relations, including appropriate versions of all primary and foreign key integrity constraints.
3. Define the Dept relation in SQL so that every department is guaranteed to have a manager.
4. Write an SQL statement to add John Doe as an employee with $eid = 101$, $age = 32$ and $salary = 15,000$.
5. Write an SQL statement to give every employee a 10 percent raise.
6. Write an SQL statement to delete the Toy department. Given the referential integrity constraints you chose for this schema, explain what happens when this statement is executed.

Answer 3.8 The answers are given below:

1. Consider the following example. It is natural to require that the *did* field of Works should be a foreign key, and refer to Dept.

```
CREATE TABLE Works ( eid     INTEGER NOT NULL ,
                     did     INTEGER NOT NULL ,
```

```
                                pcttime INTEGER,
                                PRIMARY KEY (eid, did),
                                UNIQUE (eid),
                                FOREIGN KEY (did) REFERENCES Dept )
```

When a user attempts to delete a Dept tuple, There are four options:

- Also delete all Works tuples that refer to it.
- Disallow the deletion of the Dept tuple if some Works tuple refers to it.
- For every Works tuple that refers to it, set the *did* field to the *did* of some (existing) 'default' department.
- For every Works tuple that refers to it, set the *did* field to *null*.

2.
```
CREATE TABLE Emp (    eid         INTEGER,
                      ename       CHAR(10),
                      age         INTEGER,
                      salary      REAL,
                      PRIMARY KEY (eid) )
CREATE TABLE Works (  eid         INTEGER,
                      did         INTEGER,
                      pcttime     INTEGER,
                      PRIMARY KEY (eid, did),
                      FOREIGN KEY (did) REFERENCES Dept,
                      FOREIGN KEY (eid) REFERENCES Emp,
                      ON DELETE CASCADE)
CREATE TABLE Dept (   did         INTEGER,
                      budget      REAL,
                      managerid   INTEGER ,
                      PRIMARY KEY (did),
                      FOREIGN KEY (managerid) REFERENCES Emp,
                      ON DELETE SET NULL)
```

3.
```
CREATE TABLE Dept (  did        INTEGER,
                     budget     REAL,
                     managerid INTEGER  NOT NULL ,
                     PRIMARY KEY (did),
                     FOREIGN KEY (managerid) REFERENCES Emp)
```

4.
```
INSERT
INTO    Emp     (eid, ename, age, salary)
VALUES          (101, 'John Doe', 32, 15000)
```

5.
```
UPDATE Emp E
```

```
        SET     E.salary = E.salary * 1.10
```

6.
```
        DELETE
        FROM    Dept D
                WHERE D.dname = 'Toy'
```

The did field in the Works relation is a foreign key and references the Dept relation. This is the referential integrity constraint chosen. By adding the action `ON DELETE CASCADE` to this, when a department record is deleted, the Works record associated with that Dept is also deleted.

The query works as follows: The Dept relation is searched for a record with name = 'Toy' and that record is deleted. The did field of that record is then used to look in the Works relation for records with a matching did value. All such records are then deleted from the Works relation.

sid	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.2 Students with $age < 18$ on Instance S

Exercise 3.12 Consider the scenario from Exercise 2.2, where you designed an ER diagram for a university database. Write SQL statements to create the corresponding relations and capture as many of the constraints as possible. If you cannot capture some constraints, explain why.

Answer 3.12 The SQL statements are as follows:

1. `CREATE TABLE` Teaches (ssn `CHAR(10)`,

```
                courseId INTEGER,
                semester CHAR(10),
                PRIMARY KEY (ssn, courseId, semester),
                FOREIGN KEY (ssn) REFERENCES Professor,
                FOREIGN KEY (courseId) REFERENCES Course )
                FOREIGN KEY (semester) REFERENCES Semester )
```

The table corresponding to Course is created using:

```
CREATE TABLE Course (   courseId INTEGER,
                        PRIMARY KEY (courseId) )
```

The tables for Professor and Semester can be created similarly.

2.
```
CREATE TABLE Teaches (  ssn      CHAR(10),
                        courseId INTEGER,
                        semester CHAR(10),
                        PRIMARY KEY (ssn, courseId),
                        FOREIGN KEY (ssn) REFERENCES Professor,
                        FOREIGN KEY (courseId) REFERENCES Course )
```

   Tables for Professor and Course can be created as before.

3. The tables created for the previous part to this question are the best we can do without using check constraints or assertions in SQL. The participation constraint cannot be captured using only primary and foreign key constraints because we cannot ensure that every entry in $Professor$ has an entry in $Teaches$.

4.
```
CREATE TABLE Professor_teaches ( ssn      CHAR(10),
                                 courseId INTEGER,
                                 semester CHAR(10),
                                 PRIMARY KEY (ssn),
                                 FOREIGN KEY (courseId)
                                         REFERENCES Course )
```

```
CREATE TABLE Course (   courseId INTEGER,
                        PRIMARY KEY (courseId) )
```

   Observe that we do not need a separate table for Professor.

5.
```
CREATE TABLE Professor_teaches ( ssn      CHAR(10),
                                 courseId INTEGER,
```

```
                    semester CHAR(10),
                    PRIMARY KEY (ssn),
                    FOREIGN KEY (courseId)
                            REFERENCES Course )
```

Observe that the table for Professor can be omitted as before. Interestingly, we do not need a table for Course either, because (i) every course must be taught, and (ii) the only attribute for Course is courseId, which is included in the Professor_teaches table. If Course had other attributes, we could need a separate table for Course, and would not be able to enforce the constraint that every course should be taught by some professor (without including all attributes of Course in Professor_teaches and dropping the Course table; a solution that leads to *redundancy* if several professors teach the same course.)

6.
```
CREATE TABLE Teaches (      gid       INTEGER,
                            courseId  INTEGER,
                            semester  CHAR(10),
                            PRIMARY KEY (gid, courseId),
                            FOREIGN KEY (gid) REFERENCES Group,
                            FOREIGN KEY (courseId) REFERENCES Course )

CREATE TABLE MemberOf (     ssn       CHAR(10),
                            gid       INTEGER,
                            PRIMARY KEY (ssn, gid),
                            FOREIGN KEY (ssn) REFERENCES Professor,
                            FOREIGN KEY (gid) REFERENCES Group )

CREATE TABLE Course (       courseId  INTEGER,
                            PRIMARY KEY (courseId) )

CREATE TABLE Group (        gid       INTEGER,
                            PRIMARY KEY (gid) )

CREATE TABLE Professor (    ssn       CHAR(10),
                            PRIMARY KEY (ssn) )
```

Exercise 3.18 Write SQL statements to create the corresponding relations to the ER diagram you designed for Exercise 2.8. If your translation cannot capture any constraints in the ER diagram, explain why.

Answer 3.18 The statements to create tables corresponding to entity sets Customer, Group, and Artist are straightforward and omitted. The other required tables can be created as follows:

1. `CREATE TABLE` Classify (title `CHAR(20)`,
   name `CHAR(20)`,
   `PRIMARY KEY` (title, name),

```
                        FOREIGN KEY (title) REFERENCES Artwork_Paints,
                        FOREIGN KEY (name) REFERENCES Group )
```

2. `CREATE TABLE` Like_Group (name `CHAR(20)`,
   cust_name `CHAR(20)`,
   `PRIMARY KEY` (name, cust_name),
   `FOREIGN KEY` (name) `REFERENCES` Group,
   `FOREIGN KEY` (cust_name) `REFERENCES` Customer)

3. `CREATE TABLE` Like_Artist (name `CHAR(20)`,
   cust_name `CHAR(20)`,
   `PRIMARY KEY` (name, cust_name),
   `FOREIGN KEY` (name) `REFERENCES` Artist,
   `FOREIGN KEY` (cust_name) `REFERENCES` Customer)

4. `CREATE TABLE` Artwork_Paints (title `CHAR(20)`,
   artist_name `CHAR(20)`,
   type `CHAR(20)`,
   price `INTEGER`,
   year `INTEGER`,
   `PRIMARY KEY` (title),
   `FOREIGN KEY` (artist_name)
   References Artist)