Week 12 Manipulating Databases (2)

Assignment Solutions

Question 1

Given the relations **Students(id, name, major)**, **Courses(id, title, instructor)**, and **Enrolls(sID,cID,term, grade)**, if we declare that Enrolls.cID is a foreign key that references Courses.id, then what types of database manipulations can possibly violate this foreign key constraint? Check all that apply.

*A: INSERT INTO Enrolls

*B: DELETE FROM Courses

*C: UPDATE Enrolls

*D: UPDATE Courses

E: INSERT INTO Courses

F: DELETE FROM Enrolls

Solution: ABCD

Explanation: Foreign key constraints are directional. Here the constraint is on Enrolls referencing Courses. You can use the notion of pointers to visualize it and ask: What situations may have an invalid pointer that reference an undefined target? For instance, a newly inserted Enrolls.cID might not have a corresponding record in Courses, and deleting a Course.id record can create a dangling pointer in Enrolls. The same goes for updating either table. However, inserting new Course tuples or deleting Enrolls tuples will not violate this constraint, which follows from how the pointers are directed in our constraint declaration.

Question 2

Given the relations **Students(id, name, major)**, **Courses(id, title, instructor)**, and **Enrolls(sID,cID,term, grade)**, how can we use CHECK() to enforce the foreign key constraint on Enrolls.sID referencing Students.id? Does this provide the same level of referential integrity guarantee as using FOREIGN KEY declaration?

*A: Add **CHECK(sID IN (SELECT id FROM Students))** to the declaration of the sID attribute when creating the Enrolls table. No, FOREIGN KEY declaration provides more thorough referential integrity guarantee than CHECK() does.

B: Add **CHECK(sID IN (SELECT id FROM Students))**to the declaration of the sID attribute when creating the Enrolls table. No, CHECK() provides more thorough referential integrity guarantee than FOREIGN KEY declaration does.

C: Add **CHECK(sID IN (SELECT id FROM Students))** to the declaration of the sID attribute when creating the Enrolls table. Yes, the two approaches provide the same level of referential integrity guarantee.

Solution: A

Explanation: CHECK() is activated only when the Enrolls table itself is modified (via INSERT or UPDATE). However, when the Students table is modified, which can potentially cause a foreign key constraint violation, CHECK() is not aware of the changes taking place outside the Enrolls table where it is declared, and hence will not be run.

Question 3

Given the relations **Students(id, name, major)**, **Courses(id, title, instructor)**, and **Enrolls(sID,cID,term, grade)**, when we are creating the Courses table, how can you use CHECK() to enforce the UNIQUE (or PRIMARY KEY) constraint on the Courses.id attribute? Answer the question with the following assumptions:

1. the SQL system fully supports subqueries in a CHECK condition statement;
2. the SQL system has been designed to allow the table that is being created (i.e. Courses) to be used in the CHECK condition for that table's own attributes;
3. a CHECK is performed *after* an INSERT or UPDATE modification involving the id attribute is applied.

Select all that apply.

*A: Add **CHECK((SELECT COUNT(*) FROM Courses) = (SELECT COUNT(DISTINCT id) FROM Courses))** to the declaration of the id attribute when creating the Courses table.

B: Add **CHECK(id NOT IN (SELECT id FROM Courses))** to the declaration of the id attribute when creating the Courses table.

C: With the assumptions, there is no reliable way to enforce the UNIQUE constraint using CHECK().

Solution: A

Explanation: For A, Checking whether the number of distinct values of the key is equal to the total number of tuples-- after a modification is applied-- is essentially checking if each key value remains unique in the table, which indeed is the UNIQUE constraint.

For B, Since the check is run after a modification is applied, the check condition will always evaluate to false, and thus any INSERT or UPDATE involving the id attribute will always be rejected! This works only if the check happens BEFORE the table is modified, which violates the third assumption.

Question 4

Given the relations **Students(id, name, major)**, **Courses(id, title, instructor)**, and **Enrolls(sID,cID,term, grade)**, how can you use ASSERTION to enforce the foreign key constraint on Enrolls.sID referencing Students.id?

*A:

CREATE ASSERTION FKConstraint CHECK (NOT EXISTS SELECT * FROM Enrolls WHERE sID NOT IN (SELECT id FROM Students))

B:

CREATE ASSERTION FKConstraint CHECK (NOT EXISTS SELECT * FROM Students WHERE id NOT IN (SELECT sID FROM Enrolls))

Solution: A

Explanation: For A, The foreign key constraint we want to impose is on Enrolls.sID referencing Students.id; that is, for every sID in Enrolls, there must be a corresponding id in Students. However, using SQL we must check that the violation (sID NOT IN…) does not happen (NOT EXISTS), since the SQL syntax does not support the expression of "for every sID in Enrolls, it exists in Students".

For B, This asserts the foreign key constraint in the opposite direction to what we declared, and hence it does not meet the requirement.

Question 5

Given the relations **Students(id, name, major)**, **Courses(id, title, instructor)**, and **Enrolls(sID,cID,term, grade)**, how can you define a trigger to implement the foreign key constraint on Enrolls.sID referencing Students.id, by cascading the DELETE events that violate the constraint? Select all triggers that meet the requirement.

*A:

CREATE TRIGGER FKConstraint AFTER DELETE ON Students REFERENCING OLD ROW AS DeletedStudent FOR EACH ROW DELETE FROM Enrolls WHERE sID = DeletedStudent.id

*B:

CREATE TRIGGER FKConstraint AFTER DELETE ON Students REFERENCING OLD TABLE AS AllDeletedStudents DELETE FROM Enrolls WHERE sID IN (SELECT id FROM AllDeletedStudents)

C:

CREATE TRIGGER FKConstraint AFTER DELETE ON Enrolls REFERENCING OLD ROW AS DeletedEnroll FOR EACH ROW DELETE FROM Students WHERE id = DeletedEnroll.sID

Solution: AB

Explanation: For A, The delete events that can violate the constraint happens only in the Students table, not in the Enrolls table. Since we allow such deletions, we choose the event timing to be AFTER and, by referencing each deleted row in Students, we can remove the corresponding rows in Enrolls whose sIDs matches the deleted Students.id

For B, We can use a statement-level trigger by referencing the entire set of deleted rows (as OLD TABLE), and delete every corresponding row in Enrolls that matches a Student by ID in the old table of deleted Students.

For C, This places FK constraint on Students.id referencing Enrolls.sID (enforcing that every student must take at least one course), which is not the requirement.

Question 6

Given the relations **Students(id, name, major), Courses(id, title, instructor), and Enrolls(sID, cID, term, grade)**, we have created a view **CS411F16Students(sID, grade)**, which shows all the students (by sID) who took "CS411" in "Fall 2016", along with their grades. Now we intend to update a particular student's grade using this view only, e.g., via **UPDATE CS411F16Students SET grade = 4.0 WHERE sID = "ab12"**; however, as we have learned, a view is not materialized and thus its "modification" must be properly translated into the modification of the underlying base table(s). How can you define a trigger to make this translation work?

*A:

CREATE TRIGGER CS411F16ChangeGrade INSTEAD OF UPDATE OF grade ON CS411F16Students REFERENCING OLD ROW AS Old NEW ROW AS New FOR EACH ROW BEGIN UPDATE Enrolls SET grade = New.grade WHERE sID = Old.sID AND cID = "CS411" AND term = "Fall 2016" END

B:

CREATE TRIGGER CS411F16ChangeGrade INSTEAD OF UPDATE OF grade ON CS411F16Students REFERENCING OLD ROW AS Old NEW ROW AS New FOR EACH ROW BEGIN UPDATE Enrolls SET grade = New.grade WHERE sID = Old.sID AND cID = Old.cID AND term = Old.term END

C:

CREATE TRIGGER CS411F16ChangeGrade BEFORE UPDATE OF grade ON CS411F16Students REFERENCING OLD ROW AS Old NEW ROW AS New FOR EACH ROW BEGIN UPDATE Enrolls SET grade = New.grade WHERE sID = Old.sID AND cID = Old.cID AND term = Old.term END

Solution: A

Explanation: For A, A trigger for a view should be activated using the INSTEAD OF event timing keyword, and since the underlying base table of CS411F16Students is Enrolls, we need to translate updates of grade on the view into updates on Enrolls. Note that the view implicitly has the cID attribute held constant as "CS411", and the term attribute held constant as "Fall 2016". Therefore, in the action part of the trigger, we need to make sure that we select only the Enrolls tuples about CS411 and Fall 2016 only.

For B, The view has only sID and grade, without the course cID and term fields (and that's one of the benefits of using the view instead of Enrolls). Therefore, when referring to an OLD ROW, we do not have access to its cID or term attribute.

For C, In addition to mistakenly accessing Old.cID and Old.term, the event timing keyword "BEFORE" is not suitable for declaring a trigger for view updates, because BEFORE / AFTER will still allow an update attempt on the view to occur, which would be illegal.