Assignment-7 Manipulating Databases

Solutions CS411 Summer ‘18

----

1 - The INSERT statement

Given the relations Students(id, name, major), Courses(id, title, instructor), and Enrolls(sID, cID, term, grade), for all the students who have taken "CS411" and received a grade of 4.0, we want to use INSERT to enroll those students in "CS511" for the "Spring 2018" term. Select the statement that meets the requirement.

- noFigure

✔

```
INSERT INTO Enrolls
SELECT id, "CS511", "Spring 2018", Null
FROM Students
WHERE id IN (
   SELECT sID
   FROM Enrolls
   WHERE cID = "CS411" AND grade = 4.0
)
```

- Inserted tuples must conform to the schema of Enrolls. Here, since we can't predetermine the grade for those future enrollments, we should use Null as a placeholder.

✖

```
INSERT INTO Enrolls
SELECT id, "CS511", "Spring 2018"
FROM Students
WHERE id IN (
   SELECT sID
   FROM Enrolls
   WHERE cID = "CS411" AND grade = 4.0
)
```

- Inserted tuples must conform to the schema of Enrolls. Here, the constructed tuples are missing the value for "grade".

✖

```
INSERT ON Enrolls
SELECT id, "CS511", "Spring 2018", Null
FROM Students
WHERE id IN (
   SELECT sID
   FROM Enrolls
   WHERE cID = "CS411" AND grade = 4.0
)
```

- INSERT INTO Table is the correct syntax. (INSERT ON Table is used in Triggers.)

---

2 - The downside of indexes

In one of the video lectures, Prof. Chang talked about a feature of databases called indexes. Indexes are used to speed up querying. In essence, indexes facilitate the retrieval of a subset of a table's tuples and attributes *without* the need to traverse every single record in the table. However, in the 'Food for Thought' question, he suggests that creating an index for every single attribute in the table is a bad idea. Which of the following statements is/are true in the context of indexes? You don't need to know balanced-trees (B-trees) to answer this question.

- noFigure

✔ Everytime a row is updated, all the indexes on the column(s) affected need to be modified as well. As we have higher number of indexes, the load at the server rises to keep all the schema objects up-to-date and this tends to slow things down.

- Indexes are a special type of table. Therefore, every modification on the main table should be written through into the index table. Naturally, this data structure needs to be stored on the disk. As indexes put extra load on the servers, the servers need extra care from the administrator to manage system utilization and disk space. In general, it is common to have neither too many indexes nor too few. This is taken care of by the database administrator.

✖ Columns that are rarely used to retrieve data should be indexed more and columns with higher activity should be left out of indexing.

- Indexes are a special type of table. Therefore, every modification on the main table should be written through into the index table. Naturally, this data structure needs to be stored on the disk. As indexes put extra load on the servers, the servers need extra care from the administrator to manage system utilization and disk space. This is taken care of by the database administrator.

✔ Columns that are frequently used to retrieve data should be indexed more and columns with lesser activity should be left out of indexing.

- Indexes are a special type of table. Therefore, every modification on the main table should be written through into the index table. Naturally, this data structure needs to be stored on the disk. As indexes put extra load on the servers, the servers need extra care from the administrator to manage system utilization and disk space. This is taken care of by the database administrator.

---

3 - The UPDATE statement

Given the relations Students(id, name, major), Courses(id, title, instructor), and Enrolls(sID, cID, term, grade), for each Mathematics major who has taken CS473 and received a grade of 4.0 (possibly among multiple grades if the student has taken the course more than once), use the UPDATE statement to change the major of those students to CS. Select all the queries that meet the requirement.

- noFigure

✔

```
UPDATE Students
SET major = "CS"
WHERE major = "Mathematics" and 4.0 = ANY (
   SELECT grade
   FROM Enrolls
   WHERE id = sID AND cID = "CS473"
)
```

- A student can possibly take the same course multiple times (in different semesters), and it counts as long as the student received one grade of 4.0. Therefore, the subquery can return multiple grade values, and we use ANY to check if at least one grade equals 4.0.

✔

```
UPDATE Students
SET major = "CS"
WHERE major = "Mathematics" AND  EXISTS (
   SELECT sID
   FROM Enrolls
   WHERE id = sID AND cID = "CS473" AND grade = 4.0
```

)

- The subquery finds the set of students, by ID, who have taken the course and got a grade of 4.0, by joining Enrolls and Students (i.e. Students.id = Enrolls.sID). Note that since the student can take the course more than once, the set can have multiple sIDs. We just need to check if the set is empty or not.

✖

```
UPDATE Students
SET major = "CS"
WHERE major = "Mathematics" AND  EXISTS (
   SELECT sID
   FROM Enrolls
   WHERE cID = "CS473" AND grade = 4.0
)
```

- In the WHERE clause in the subquery, the equi-join between Students.id and Enrolls.sID is missing.

---

4 - Views

Given the relations Students(id, name, major), Courses(id, title, instructor), and Enrolls(sID, cID, term, grade), how can you create a view, named "CS411Students", for all the students (by ID) who have taken CS411, along with the terms and grades? Now using that view, how can you find all the Bioengineering majors in the view (by ID and name) who got a grade of 4.0 in CS411?

- noFigure

✔

```
CREATE VIEW CS411Students AS
SELECT sID, term, grade
FROM Enrolls
WHERE cID = "CS411"


SELECT S.id, S.name
FROM Students S, CS411Students S411
WHERE S.id = S411.sID AND S.major = "Bioengineering" AND S411.grade = 4.0
```

- The view essentially extracts every tuple from Enrolls whose cID attribute is "CS411". This view can then be used for queries as if it were a real table, and we no longer have to specify the course in the join condition, because it is a constant value "CS411" in this view.

✖

```
CREATE VIEW (
SELECT sID, term, grade
FROM Enrolls
WHERE cID = "CS411"
) AS CS411Students


SELECT S.id, S.name
FROM Students S, CS411Students S411
WHERE S.id = S411.sID AND S.major = "Bioengineering" AND S411.grade = 4.0
```

- The correct syntax for view creation is CREATE VIEW ViewName AS (...Subquery...). The answer wrongly uses the "(...Subquery...) AS Name" syntax in the FROM clause.

✖

```
CREATE VIEW CS411Students AS
SELECT sID, term, grade
FROM Enrolls
WHERE cID = "CS411"


SELECT S.id, S.name
FROM Students S, CS411Students S411
WHERE S.major = "Bioengineering" AND S411.grade = 4.0
```

- The second SQL SELECT is missing equi-join between Students and the view.

---

5 - Cascading update

Consider the following schema called Tournament consisting of the following tables:.

Player(player_name, team_name, age, num_matches_played)
Captain(capt_name, team_name, age, position, height, weight)

We create the tables using the commands:

```
CREATE TABLE Player
( player_name VARCHAR(50) PRIMARY KEY,
  team_name VARCHAR(20) NOT NULL,
  age INT,
  num_matches_played INT
);
```

```
CREATE TABLE Captain
( capt_name VARCHAR(50) NOT NULL,
   team_name VARCHAR(20) PRIMARY KEY,
   age INT,
   position VARCHAR(15),
   height INT,
   weight INT,
   FOREIGN KEY (capt_name)
   REFERENCES Player (player_name)
);
```

Here, we have created a foreign key on the Captain table which references the Player table on the player_name field. This is because the captain is a player himself. Here, the table Captain is called the referencing table and the table Player is called the referenced table.

Now, the tournament authorities realize that there is a typo in the name of one of the players, Messi. Unfortunately, it is spelt as Mesi instead of Messi. So, they hire a database engineer Mary to fix the problem. She tries to fix the issue by running the following commands on the Player and Captain tables:

```
UPDATE Player
SET player_name = ‘Messi’
WHERE player_name = ‘Mesi’;
```

```
UPDATE Captain
SET capt_name = ‘Messi’
WHERE capt_name = ‘Mesi’;
```

Which of the following is/are true?

- noFigure

✖ Both the updates succeed as the referenced table is updated first followed by the referencing table. We don’t need any more operations to fix the typo in the name.
✖ The update is reflected in the referencing table only but not in the referenced table. We need to delete the existing foreign key and add a new one that includes the ON UPDATE CASCADE clause using ALTER TABLE on the referenced table to have the change reflected on both tables.
✖ The update is reflected in the referenced table only but not in the referencing table. We need to delete the existing foreign key and add a new one that includes the ON UPDATE CASCADE clause using ALTER TABLE on the referencing table to have the change reflected on both tables.
✔ Neither the referencing table nor the referenced table are updated successfully. We need to delete the existing foreign key and add a new one that includes the ON UPDATE CASCADE clause. Now, we can perform an UPDATE on the referenced table which will be propagated by the server to all the referencing tables.

- We need to execute the following commands for successful updates:

-
  ```
  ALTER TABLE Captain
  DROP FOREIGN KEY ;
  ```
- Note: fk_symbol is generated internally when the foreign key is created. This symbol can be found out by using the command SHOW CREATE TABLE Captain;

-
  ```
  ALTER TABLE Captain
  ADD FOREIGN KEY (capt_name)
  REFERENCES Player (player_name)
  ON UPDATE CASCADE;
  ```
- Now we can update in the referenced table using:

-
  ```
  UPDATE Player
  SET player_name = 'messi'
  WHERE player_name = 'mesi';
  ```
- This will execute such that the changes get reflected in both the Player table and the Captain table. This process is called cascading update. Note that the cascading update propagates from the referenced to the referencing table and not the other way around.

---

6 - Check Vs. Foreign Key Constraint

Consider the relations Students(id, name, major), Courses(id, title, instructor), and Enrolls(sID, cID, term, grade).

Now, suppose we use CHECK() to enforce a foreign key constraint on Enrolls.sID referencing Students.id by adding CHECK(sID IN (SELECT id FROM Students)) to the declaration of the sID attribute when creating the Enrolls table. Does this provide the same level of referential integrity guarantee as using FOREIGN KEY declaration?

- noFigure

✔ No, FOREIGN KEY declaration provides more thorough referential integrity guarantee than CHECK() does.

- 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, and hence will not be run.

✖ No, CHECK() provides more thorough referential integrity guarantee than FOREIGN KEY declaration does.

✖ Yes, the two approaches provide the same level of referential integrity guarantee.

---

7 - Check for Unique Constraint

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.

- noFigure

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

- Checking whether the number of distinct values of the key is equal to the total number of tuples after the modification is applied, is essentially checking if each key attribute's value remains unique in the table, and that indeed is the UNIQUE constraint.

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

- Since the check is run after the 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,

✖ With the two assumptions above, there is no reliable way to enforce the UNIQUE constraint using CHECK().

---

8 - Assertions

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?

- noFigure

✔

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

- 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). Alternatively, we can use set difference to assert that the set of sIDs must be a subset of the IDs in Students, using the fact that R - S is the empty set iff R is a subset of S.

✖

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

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

---

9 - Triggers

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 DELETE events that violate the constraint? Select all triggers that meet the requirement.

- noFigure

✔

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

- 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 sID matches Students.id.

✖

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

- We use a statement-level trigger by referencing the entire set of deleted rows (as OLD TABLE), and delete every corresponding row from the old table of deleted Students, which is not our requirement.

✖

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

- 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.

---

10 - Triggers and Assertions

For regulating databases, one could use general constraint mechanisms such as checks or assertions. However, checks and assertions fall short on several counts. So, a better solution is to use triggers. Which of the following issues of checks are overcome by using triggers?

- noFigure

✔ Checks and assertions can slow down databases since they are expensive operations. Triggers help overcome this by providing an event-based mechanism to enforce constraints.

- The database programmer can specify events such as :

```
AFTER
UPDATE OF <attribute> ON <table>
```

✔ Checks lack flexibility as they are limited to attribute or tuple-based constraints. Triggers support any SQL boolean-valued expression and allows us to refer to the scope of a row or for a statement.

The database programmer can specify conditions in triggers such as:

```
REFERENCING
OLD ROW AS old
NEW ROW AS new
FOR EACH ROW
WHEN (new.price > old.price + 5.00)
```

✔ Triggers were designed to handle view updates which is something checks and assertions cannot accomplish.

A special type of trigger, called instead-of trigger, allows one to essentially intercept insert, update, and delete statements against a view, and write custom code to incorporate the changes.
