SQL Programming: Triggers, Functions, and Stored Procedures

CISC637, Lecture #13

Ben Carterette

Database Design and Application Development

- Database design is about how to define relational schema to:
  - Capture as many requirements as possible through schema/key definition
  - Minimize redundancy in data
  - Maximize storage and query efficiency
- Not always possible or desirable to do all three
- Sometimes better (or just easier) to address things at the application layer
  - Applications connect to the database (either locally or remotely) to retrieve or input data
  - You can write e.g. Java code to retrieve any data from a database and check it for validity or anything else

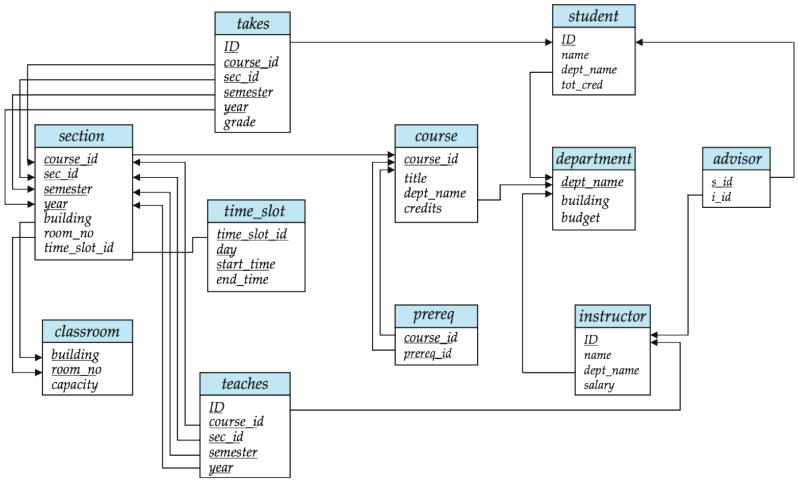
Database Design and Application Development

- There's a middle layer in between the schema and the applications that connect to the database:
  - SQL program layer
  - Full programs stored in the database in order to do things that cannot be achieved with schema definition alone
- SQL is actually a full, Turing-complete programming language
  - Supports conditionals, iteration, recursion, temporary storage, etc
  - Not much fun to program in though…

Triggers

- A **trigger** is a procedure that automatically runs when there is a change to the database
- Triggers consist of two parts:
  - **Activation event**, the change that triggers the procedure
  - **Action** to execute when the trigger is activated and a given test condition is true

Trigger Example



Trigger Example

- Using triggers to keep tot_cred up to date

```
CREATE TRIGGER increase_creds AFTER INSERT ON Takes
FOR EACH ROW
BEGIN
  UPDATE Student SET tot_cred = tot_cred + (SELECT credits FROM
   Course WHERE course_id=NEW.course_id) WHERE ID=NEW.ID;
END


CREATE TRIGGER update_creds AFTER UPDATE ON Course
FOR EACH ROW
BEGIN
  UPDATE Student SET tot_cred = tot_cred - OLD.credits +
   NEW.credits WHERE ID IN (SELECT ID FROM Takes WHERE
   course_id=OLD.course_id);
END
```


Trigger Example

- tot_cred depends on course grades

```
CREATE TRIGGER credits_earned AFTER UPDATE ON Takes
FOR EACH ROW
BEGIN
  IF NEW.grade != 'F' AND NEW.grade IS NOT NULL
     AND (OLD.grade = 'F' OR OLD.grade IS NULL)
  THEN
    UPDATE Student SET tot_cred = tot_cred +
       (SELECT credits FROM Course WHERE
        course.course_id = NEW.course_id) WHERE
       Student.ID = NEW.ID;
  END IF;
END
```


Trigger Example

- Disallow deletions from Takes if the student ID still exists in Student

```
CREATE TRIGGER takes_delete_student_check BEFORE DELETE ON Takes
FOR EACH ROW
BEGIN
  IF OLD.ID IN (SELECT ID FROM Student) THEN
    UPDATE `Error:  student ID still exists` SET x=1;
  END IF;
END
```


Triggers

- `SHOW TRIGGERS;`
- `DROP TRIGGER x;`

- No way to change a defined trigger

Why Use Triggers?

- Keep database logic with the database
  - The alternative is putting database logic in high-level application code
    - Poor modularity
    - There may be multiple applications accessing the same database
      - Desktop client, web app, mobile app
- Keep derived fields up-to-date
  - tot_cred for example
    - If the field didn't exist, computing total credits "on the fly" would require an expensive GROUP BY
    - When the field does exist, it needs to be kept up-to-date
    - Using a trigger to keep it up-to-date avoids GROUP BY
- Help with DBA tasks
  - For example, you could write a trigger that automatically sends someone an email to notify them of high-priority events

SQL Programming Language

- Supports standard programming language constructs
  - Local variables:
    - DECLARE [variable] [type] [DEFAULT [value]];
  - Variable assignment from expressions:
    - SET [variable] = [expression];
  - Conditionals:
    - IF [expression] THEN … END IF;
  - Conditional loops:
    - WHILE [expression] DO … END WHILE;
- "Global" variables @x
  - A trigger/procedure/function can modify @x
  - After execution, SELECT @x will return modified value

Cursors

- Use cursors to iterate over results of a SELECT

```
DECLARE ID INT;
DECLARE name VARCHAR(20);
DECLARE dept_name VARCHAR(60);
DECLARE tot_cred INT;
DECLARE cur1 CURSOR FOR SELECT * FROM Student;

OPEN cur1;
LOOP
  FETCH FROM cur1 INTO ID, name, dept_name, tot_cred;
  [do something with variables];
END LOOP;
CLOSE cur1;
```


Stored Procedures and Functions

- Logic stored in the database and executed within the DBMS process space
- They allow:
  - Encapsulation of application logic
  - Reuse of application logic by different higher-level applications and users

Defining a Function

```
CREATE FUNCTION fname ([var1] [type1], [var2]
  [type2], …) RETURNS [type]
```

- Functions must return some value of the given type
- Function body may consist of compound statements, which should be in a BEGIN .. END block
- Functions can be used in SQL statements
  - As a field to be selected, as a condition in a WHERE clause

Function Example

```
CREATE FUNCTION countInst (dept
  VARCHAR(40)) RETURNS INT
BEGIN
  DECLARE count INT DEFAULT 0;
  SELECT COUNT(*) INTO count FROM
    Instructor WHERE dept_name = dept;
  RETURN count;
END
```


Defining a Stored Procedure

```
CREATE PROCEDURE proc_name ([var1]
  [type1], [var2] [type2], …)
```

- Procedures do not return values, but they can take variables designated for output
  - Use IN, OUT, INOUT (optionally) before parameters to indicate input, output, or both

Procedure Example

```
CREATE PROCEDURE countInst (IN dept
  VARCHAR(40), OUT count INT)
BEGIN
   SELECT COUNT(*) INTO count FROM
    Instructor WHERE dept_name=dept;
END
```

Functions and stored procedures are useful for executing expensive queries and saving results for use in other queries or procedures

Useful Statements

- `DROP FUNCTION [IF EXISTS], DROP PROCEDURE [IF EXISTS]`
- `SHOW FUNCTION STATUS, SHOW PROCEDURE STATUS`
- `SHOW CREATE FUNCTION [fname], SHOW CREATE PROCEURE [fname]`

Why Use Functions vs Procedures?

- Functions return a value, which means they can be used in SELECT statements
  - Think aggregation functions like COUNT(), SUM() or string functions like SUBSTR()
  - Procedures cannot be used in SELECTs
- Procedures can modify data
  - They can execute UPDATE/INSERT/DELETE
  - Functions cannot
- Procedures are pre-compiled in the database
  - Like a compiled language, helps with faster execution
  - Functions are not, they are run like scripting languages

CALL fail()

- Hack MySQL to support CHECK constraints
  - CHECK is part of the SQL standard
  - Defined in a CREATE TABLE statement to further limit the data that can be put in the table
  - Useful for enforcing constraints that can't be enforced in SQL schema
  - But MySQL doesn't support it– so let's hack it in
- First define an in-memory table called Error:

```
CREATE TABLE Error (
  Message VARCHAR(128),
  PRIMARY KEY (Message)) ENGINE=MEMORY;
```

- Then define a fail() procedure:

```
DELIMITER $$
CREATE PROCEDURE fail(message VARCHAR(128))
BEGIN
  INSERT INTO Error VALUES (message);
  INSERT INTO Error VALUES (message);
END $$
DELIMITER ;
```


Using a Trigger to Emulate CHECK

- Suppose GPA added as field for students
- Trigger to check value of GPA in defined range

```
CREATE TRIGGER gpa_check BEFORE INSERT ON
  Student
FOR EACH ROW
BEGIN
    IF NEW.gpa < 0 OR NEW.gpa > 4 THEN
        CALL fail('GPA not in allowed range.');
    END IF;
END
```


Using a Trigger to Emulate CHECK

- Simple example:
  - Suppose CISC437 is only offered in the Spring

```
CREATE TRIGGER course_check BEFORE INSERT ON Section
FOR EACH ROW
BEGIN
    IF NEW.course_id = 'CISC437' AND
     NEW.semester != 'Spring' THEN
        CALL fail('CISC437 only offered in Spring');
    END IF;
END
```


Enforcing FDs With Triggers

- Requirements: a student can have multiple advisors, but only one advisor per department
  - Previously we did this using a 3NF table: DeptAdvisor(s_ID, dept_name, i_ID)
  - Let's do it with BCNF tables and a trigger instead
    - StudentDept(s_ID, dept_name)
    - Advisor(s_ID, i_ID)


```
CREATE TRIGGER adv_dept_check BEFORE INSERT ON Advisor
FOR EACH ROW
BEGIN
    DECLARE stu_dept VARCHAR(40);
    DECLARE adv_dept VARCHAR(40);
    DECLARE inst_dept VARCHAR(40);
    DECLARE cur CURSOR FOR
        SELECT dept_name FROM StudentDept WHERE ID = NEW.s_ID;
    DELCARE advcur CURSOR FOR
        SELECT dept_name FROM Advisor A JOIN Instructor I ON A.i_ID=I.ID WHERE A.s_ID=NEW.s_ID;
    DECLARE match BOOL DEFAULT 0;

    SELECT dept_name INTO inst_dept FROM Instructor WHERE ID = NEW.i_ID;

    OPEN advcur;
        LOOP
            FETCH advcur INTO adv_dept;
            IF adv_dept = inst_dept THEN
                CALL fail(CONCAT('student already has an advisor in ', inst_dept));
            END IF;
        END LOOP;
    CLOSE advcur;

    OPEN cur;
        cloop: LOOP
            FETCH cur INTO stu_dept;
            IF stu_dept = inst_dept THEN
                SET match = 1;
                LEAVE cloop;
            END IF;
        END LOOP;
    CLOSE cur;

    IF match = 0 THEN
        CALL fail('student and instructor are not in the same department');
    END IF;
END
```

Enforcing FDs With Triggers

- Total participation constraints are sometimes difficult to enforce in 3NF SQL schema
  - For example "every instructor advises at least one student" would be difficult to enforce in schema
- Can be enforced using triggers though
  - Define a trigger that automatically assigns new instructors to new students for example
  - Define a trigger for when all of an instructor's advisees have graduated to assign them new students