LECTURE 17 – TRIGGERS

CS2208
Information Storage and Management I

A TRADITION OF INDEPENDENT THINKING

UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Dr Harry Nguyen
<hn@cs.ucc.ie>

What We Saw Last Lecture

Integrity Constraints

Database and Software Development

Stored Procedures

INTEGRITY CONSTRAINTS – WHAT ARE THEY?

Integrity constraints ensure that changes made to databases do not cause any inconsistencies

These constraints are defined when a table is created

We can either write SQL to capture these constraints or we can use graphical interfaces in WorkBench

We can also add constraints using an alter statement but we should be careful if the table is populated with existing data

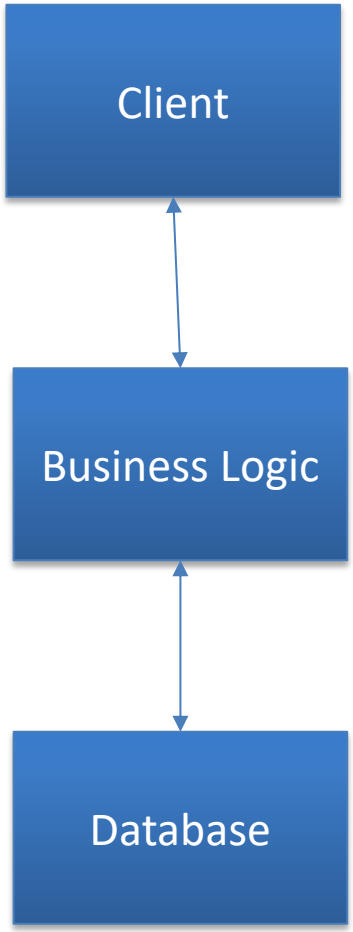
LIMITATIONS OF SQL

- SQL is generally a declarative language
  - We specify the information that we need or want to modify but we leave it to the DBMS to make the modifications

- Procedural languages, such as Python, Java, C/C++, etc allow us to specify a sequence of actions and how they will occur:
  - We sometimes need this degree of control to express processes that are used by an enterprise.

- As a result, we often see software architectures that are three or n tier – with the database at the lowest tier.

TYPICAL THREE TIER ARCHITECTURE



- **User Interfaces**

- **Business Logic**

- **Data management**

Databases and Software Development

- Typically we would use a library or framework to manipulate the database
  - JDBC in Java
  - ODBC in Visual C++ and Visual Basic
  - PyMySQL in Python

- Manipulation of the database is often driven from the code:
  - Some frameworks are explicitly organized that way, for example Django
  - In other cases, there are several arcs of software development, Webapp, iOS app, Android app, perhaps all communicating with the database directly.
    - In this case, the architecture can become fragmented and result in duplication of typical interactions with the database

Stored Procedures and Functions

- Most DBMS allow us to define both stored procedures and functions
- Each DBMS has its own syntax for doing this
- We can embed common actions in our database into SQL using functions
  - We've already used functions, such as SUM(), AVG(), etc
- Stored Procedures capture processes that we might wish to implement
  - We can call these from our SQL or we can invoke them through a library call from a procedural language

STORED PROCEDURE - EXAMPLE

CS2208 Information Storage and Management I

```
DELIMITER $$
CREATE PROCEDURE `book_room`(
IN roomid int,
IN booker varchar(45),
IN roomname varchar(45),
IN guestnumber int,
IN eventdate varchar(45),
OUT result int)

BEGIN

  DECLARE roomlimit INT;
  SELECT roombookinglimit.limit into roomlimit FROM roombookinglimit WHERE
roombookinglimit.roomname = roomname;

  IF roomlimit >= guestnumber THEN
    INSERT INTO roombooking values (roomid, booker, roomname, guestnumber,
eventdate);
    SET result = 0;
  ELSE
    SET result = 1;
  END IF;

END$$
```

STORED PROCEDURE – SETUP & RUN

CS2208 Information Storage and Management I

```
CREATE TABLE `roombookinglimit` (
`roomname` VARCHAR(45) NOT NULL,
`limit` INT NULL,
PRIMARY KEY (`roomname`));

CREATE TABLE `roombooking` (
`roomid` INT NULL,
`booker` VARCHAR(45) NULL,
`roomname` VARCHAR(45) NULL,
`guestnumber` INT NULL,
`event_date` VARCHAR(45) NULL);

INSERT INTO `roombookinglimit`(`roomname`, `limit`)
VALUES ('Double', 2),('Triple', 3);

CALL book_room(101, 'HN', 'Double', 3, '2022-11-08', @R1);
SELECT @R1; -- Result: 1

CALL book_room(101, 'HN', 'Triple', 3, '2022-11-08', @R2);
SELECT @R2; -- Result: 0

SELECT * FROM roombooking;
```

TRIGGERS

A trigger extends the idea of a stored procedure and uses many of the same features.

Triggers extend the concept of a stored procedure by automatically invoking (or triggering) in response to some database event such as an insert, update or delete in an associated table.

FUNCTION OF TRIGGERS

- Triggers have many functions

CHECKING INTEGRITY OF DATA

UPDATING OTHER TABLES BASED ON CHANGES TO ONE TABLE

SUPPORTING AUTOMATION OF PARTS OF TRANSACTIONS

CREATING TRIGGERS

- We can create triggers as follows:

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} { INSERT | UPDATE | DELETE}
ON table_name FOR EACH ROW
trigger_body;
```

ACCESSING ROW VALUES

- We can access the original row values using OLD and updated or new values for a row using NEW

EVENT		
INSERT	NO	YES
UPDATE	YES	YES
DELETE	YES	NO

Adding and Showing Triggers

- We can use the same approach to see existing triggers that we might want to edit. We can also create triggers through raw SQL. We can view existing triggers using ‘SHOW Triggers”

An Example

- Let's imagine that we have a table as follows:

  StudentResults(studid, name, mod1, mod2, mod3, total, average, grade)

- We want to minimize the work for staff by having them enter just the id, name and individual scores
  - We will automatically calculate the total, average and grade from there.
  - Grade will be decided by assigning a string value to numeric scores.

An Example - Setup

CS2208 Information Storage and Management I

```
CREATE TABLE `cs2208_hnlab`.`studentresults` (
`studid` INT NOT NULL,
`name` VARCHAR(45),
`mod1` INT,
`mod2` INT,
`mod3` INT,
`total` INT,
`average` INT,
`grade` VARCHAR(45));
```

AN EXAMPLE - TRIGGER

CS2208 Information Storage and Management I

```
DELIMITER $$
CREATE TRIGGER `studentresults_before_insert`
BEFORE INSERT ON `studentresults` FOR EACH ROW
BEGIN
  SET NEW.TOTAL = NEW.MOD1 + NEW.MOD2 + NEW.MOD3;
  SET NEW.AVERAGE = NEW.TOTAL/3;
  IF NEW.AVERAGE > 90 THEN
    SET NEW.GRADE = 'EXCELLENT!';
  ELSEIF NEW.AVERAGE < 90 AND NEW.AVERAGE >= 70 THEN
    SET NEW.GRADE = 'VERY GOOD!';
  ELSEIF NEW.AVERAGE < 70 AND NEW.AVERAGE >= 55 THEN
    SET NEW.GRADE = 'GOOD';
  ELSEIF NEW.AVERAGE < 55 AND NEW.AVERAGE >= 40 THEN
    SET NEW.GRADE = 'OK';
  ELSE
    SET NEW.GRADE = 'OH DEAR!';
  END IF;
END$$
```

AN EXAMPLE - RUN


```
INSERT INTO studentresults(`studid`, `name`, `mod1`, `mod2`, `mod3`)
VALUES (1001, 'A', 90, 100, 90);

INSERT INTO studentresults(`studid`, `name`, `mod1`, `mod2`, `mod3`)
VALUES (1002, 'B', 70, 80, 90);

INSERT INTO studentresults(`studid`, `name`, `mod1`, `mod2`, `mod3`)
VALUES (1003, 'C', 60, 50, 70);

INSERT INTO studentresults(`studid`, `name`, `mod1`, `mod2`, `mod3`)
VALUES (1004, 'D', 50, 50, 40);

INSERT INTO studentresults(`studid`, `name`, `mod1`, `mod2`, `mod3`)
VALUES (1005, 'E', 30, 30, 20);

SELECT * FROM studentresults;
```

studid	name	mod1	mod2	mod3	total	average	grade
1001	A	90	100	90	280	93	EXCELLENT!
1002	B	70	80	90	240	80	VERY GOOD!
1003	C	60	50	70	180	60	GOOD
1004	D	50	50	40	140	47	OK
1005	E	30	30	20	80	27	OH DEAR!

ANOTHER EXAMPLE

CS2208
Information Storage and Management I

```
DELIMITER $$
CREATE TRIGGER `Emp_after_delete`
AFTER DELETE ON `Emp` FOR EACH ROW

BEGIN

DECLARE TOTAL INT;
SELECT COUNT(Emp.empid) INTO TOTAL
FROM Emp
WHERE Emp.deptid = OLD.deptid;

UPDATE Dept SET staffcount = TOTAL WHERE Dept.deptid =
OLD.deptid;

END$$
```

SUMMARY

Calling Stored Procedures

Defining Triggers

Trigger Examples