CMSC 508 Databases

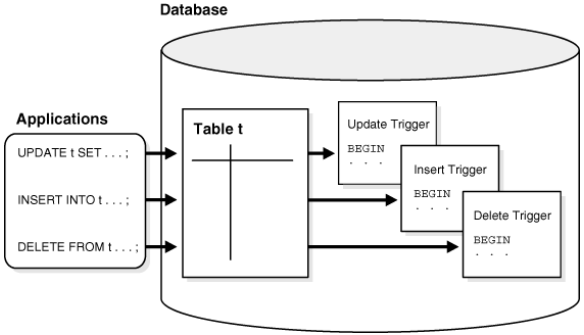
Advanced SQL (II)

VCU College of Engineering

Dr. Alberto Cano
Associate Professor
Department of Computer Science

Chapter 3 from Database System Concepts, 7th Ed. by Silberschatz, Korth, Sudarshan, 2019
Chapter 5 from Database Management Systems, 3rd Ed. by Ramakrishnan, Gehrke, 2003

- Triggers
  - A **trigger** is a statement that is executed **automatically** by the DBMS as a result of a modification to the database
  - To design a trigger we must:
    - Specify the **conditions** under which the trigger is to be executed
    - Specify the **actions** to be taken when the trigger executes
  - **Triggers vs procedures**: a procedure is explicitly run by an user or trigger. Triggers are **implicitly fired** by the DBMS when a triggering condition occurs.



- Triggers
  - Triggering event can be an **INSERT**, **DELETE** or **UPDATE**
  - Triggers may execute **BEFORE** or **AFTER** a manipulation of a table
    - **BEFORE|AFTER INSERT ON** *tablename*
    - **BEFORE|AFTER UPDATE ON** *tablename*
    - **BEFORE|AFTER DELETE ON** *tablename*
  - Triggers **cannot** change a table that is already undergoing change. Otherwise, you'll get a **mutating table error**.

```
CREATE TRIGGER BEFORE|AFTER INSERT|UPDATE|DELETE ON tableName
FOR EACH ROW
BEGIN
  -- DO STUFF
END//
```

MySQL will run the trigger code for every row which was **affected** by the SQL statement and **NOT** for each row in the table itself

- Triggers
  - Values of attributes before and after an operation can be referenced

Statement	**old.attribute**	**new.attribute**
INSERT	NULL	Post-insert value
UPDATE	Pre-update value	Post-update value
DELETE	Pre-delete value	NULL

```
CREATE TRIGGER salary_log_trigger
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
    IF(new.salary <> old.salary) THEN
        INSERT INTO salary_log (employee_id, new_salary, old_salary)
        VALUES (new.employee_id, new.salary, old.salary);
    END IF;
END//
```

- Trigger (**AFTER**)
  - Commonly employed for log information after a modification
  - DB example: maintaining the job history of the employees

```
CREATE TRIGGER update_job_history
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
    IF (new.job_id <> old.job_id OR new.department_id <> old.department_id) THEN
        CALL add_job_history(old.employee_id, old.hire_date, sysdate(),
                             old.job_id, old.department_id);
    END IF;
END//
```

where add_job_history is an existing procedure inserting data into job_history

- Trigger (**BEFORE**)
  - Commonly employed for checking conditions **prior** modification (EXCEPTIONS)
  - Generic SQLSTATE value '45000' means “unhandled user-defined exception”
  - Example: check salary conditions within a valid range prior inserting

```
CREATE TRIGGER salary_check_trigger
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
        DECLARE job_id_avg_sal DECIMAL (9,2);

        SELECT AVG(salary) INTO job_id_avg_sal FROM employees WHERE job_id = new.job_id;

        IF(new.salary > 2*job_id_avg_sal OR new.salary < 0.5*job_id_avg_sal) THEN
                SIGNAL SQLSTATE '45000'
                SET MESSAGE_TEXT = 'Salary deviates from average of the job';
        END IF;
END//
```

- Trigger (**INSTEAD OF**) (IMPLEMENTED ON **ORACLE** BUT **NOT MYSQL**, JUST FYI)

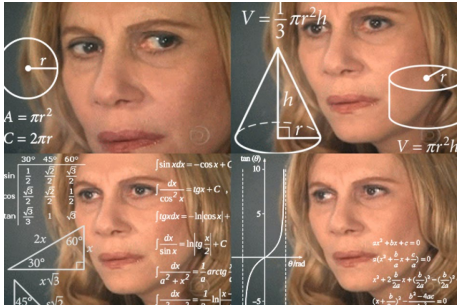
```
CREATE TRIGGER insert_emp_dept  INSTEAD OF INSERT ON emp_dept_join
DECLARE  v_department_id departments.department_id%TYPE;
BEGIN
        BEGIN
                SELECT department_id INTO v_department_id
                FROM   departments
                WHERE  department_name = :new.department_name;
                EXCEPTION
                WHEN NO_DATA_FOUND THEN
                        INSERT INTO departments (department_id, department_name)
                                VALUES (departments_seq.nextval, :new.department_name)
                        RETURNING department_id INTO v_department_id;   -- MySQL use LAST_INSERT_ID();
        END;

        INSERT INTO employees (employee_id, first_name, last_name, department_id)
        VALUES(employees_seq.nextval, :new.first_name, :new.last_name, v_department_id);
END//
```

- Trigger example
  - Create a trigger to maintain a new column in the departments table that stores the total salary of all members in a department
  - Prerequisites:

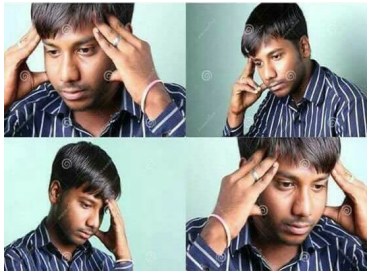
    ALTER TABLE *departments* **ADD** *total_salary* DECIMAL(12,2);

  - Trigger logic:
    - **When** the trigger should be executed by the DBMS?
    - **What** should the trigger do in each firing event case?



- Trigger example
  - Create a trigger to maintain a new column in the departments table that stores the total salary of all members in a department
  - The trigger should be executed when:
    - New employee is inserted
    - Employee is removed
    - Employee's salary is updated
    - Employee's department is updated

Please **ALWAYS** take time to **THINK** what are **ALL** the triggering conditions and **ALL** the possible actions/outcomes for each case

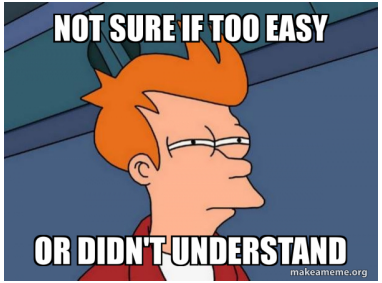


- Trigger example

```
CREATE TRIGGER total_salary_trigger_insert
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
        UPDATE departments
        SET total_salary = total_salary + new.salary
        WHERE department_id = new.department_id;
END//
```


```
CREATE TRIGGER total_salary_trigger_delete
AFTER DELETE ON employees
FOR EACH ROW
BEGIN
        UPDATE departments
        SET total_salary = total_salary - old.salary
        WHERE department_id = old.department_id;
END//
```

- Trigger example




```
CREATE TRIGGER total_salary_trigger_update
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
        IF(old.department_id <> new.department_id) THEN
                UPDATE departments SET total_salary = total_salary - old.salary
                WHERE department_id = old.department_id;

                UPDATE departments SET total_salary = total_salary + new.salary
                WHERE department_id = new.department_id;
        END IF;

        IF(old.department_id = new.department_id AND old.salary <> new.salary) THEN
                UPDATE departments SET total_salary = total_salary - old.salary + new.salary
                WHERE department_id = new.department_id;
        END IF;
END//
```

- Trigger example (ORACLE SQL)

```
CREATE TRIGGER total_salary_trigger
AFTER DELETE OR INSERT OR UPDATE OF department_id, salary ON employees
FOR EACH ROW
BEGIN
        IF DELETING OR (UPDATING AND :old.department_id != :new.department_id)
                THEN UPDATE departments
                SET total_salary = total_salary - :old.salary
                WHERE department_id = :old.department_id;
        END IF;
        IF INSERTING OR (UPDATING AND :old.department_id != :new.department_id)
                THEN UPDATE departments
                SET total_salary = total_salary + :new.salary
                WHERE department_id = :new.department_id;
        END IF;
        IF (UPDATING AND :old.department_id = :new.department_id AND :old.salary != :new.salary)
                THEN UPDATE departments
                SET total_salary = total_salary - :old.salary + :new.salary
                WHERE department_id = :new.department_id;
        END IF;
END//
```

- Trigger examples
  - Create a trigger to **maintain** a new column in the departments table that stores the total salary of all members in a department
  - Issues:
    - How to compute the current total salary?

      1) **UPDATE** *employees* **SET** *salary* = *salary*; ?

      2) **UPDATE** *departments* **SET** *total_salary* = 0; then 1) ?

total_salary is null … total_salary + :new.salary will be null

salary = *salary* … will execute the trigger, but no condition is satisfied

TROLOLOLOL

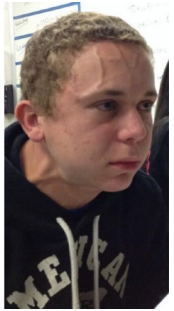
- Trigger examples
  - Create a trigger to **maintain** a new column in the departments table that stores the total salary of all members in a department
  - Issues:
    - How to compute the current total salary?

```
UPDATE departments d
SET d.total_salary =
      (SELECT sum(e.salary) FROM employees e
      WHERE d.department_id = e.department_id);
```

    - What if inserting/updating wrong department ID? Referential integrity
    - What if adding an employee/department for the first time?

Column total_salary is NULL! Triggers in the example are <u>incomplete</u>

- Trigger exercise
  - Create a trigger to increase the salary (+5% of current salary) of the employees belonging to a department every time an employee joins that department
  - Identify conditions to execute the trigger
  - Identify actions using new and old references
  - Merge conditions with common actions



Nope, not this kind of trigger

- Trigger exercise
  - Create a trigger to increase the salary (+5% of current salary) of the employees belonging to a department every time an employee joins that department

```
CREATE TRIGGER update_salary_insert
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
    UPDATE employees
    SET salary = salary*1.05
    WHERE department_id = new.department_id;
END//
```

```
CREATE TRIGGER update_salary_update
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
    IF(new.department_id <> old.department_id) THEN
        UPDATE employees  SET salary = salary*1.05
        WHERE department_id = new.department_id;
    END IF;
END//
```

Triggers compile and everything looks good. Let's run something to execute them

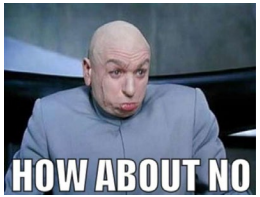
- Mutating table

A **mutating table** is a table that is currently being modified by an update, delete, or insert statement. When a trigger tries to reference a table that is in state of flux (being changed), it is considered "mutating", and raises an error.

```
CREATE TRIGGER update_salary_insert
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
    UPDATE employees
    SET salary = salary*1.05
    WHERE department_id = new.department_id;
END//
```

```
CREATE TRIGGER update_salary_update
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
    IF(new.department_id <> old.department_id) THEN
        UPDATE employees  SET salary = salary*1.05
        WHERE department_id = new.department_id;
    END IF;
END//
```

```
#1442 - Can't update table 'employees' in stored function/trigger because it is already used by statement which invoked this stored function/trigger.
```



- Exercises **These exercises should be included in the second SQL homework.**

- Create a trigger to prevent having employees whose salary is bigger than their manager (or the president's salary if they have no manager). Consider all scenarios.
- Create a table for projects (title, manager, duration (days), cost), and check that the cost must be < 1000 per day nor bigger than the sum of the salaries of the department employees the manager works for. Consider all scenarios.
- Create a new table to keep the count of the number of subordinates of an employee. Create a trigger to keep this table up to date. Remove from this table the data of the employee if fired. Consider all scenarios.
- Create a new log table and a trigger to keep track of any changes to the employees table. The table schema should be (log_event_id, date, description) and the contents should look as e.g. (1234, 04/05/17, "Employee 123 updated salary from 5000 to 10000"). Track salaries, managers, departments, and jobs.

CMSC 508 Databases

Advanced SQL (II)

VCU College of Engineering

Dr. Alberto Cano
Associate Professor
Department of Computer Science

Chapter 3 from Database System Concepts, 7th Ed. by Silberschatz, Korth, Sudarshan, 2019
Chapter 5 from Database Management Systems, 3rd Ed. by Ramakrishnan, Gehrke, 2003