Lesson 7.3: Triggers & Views

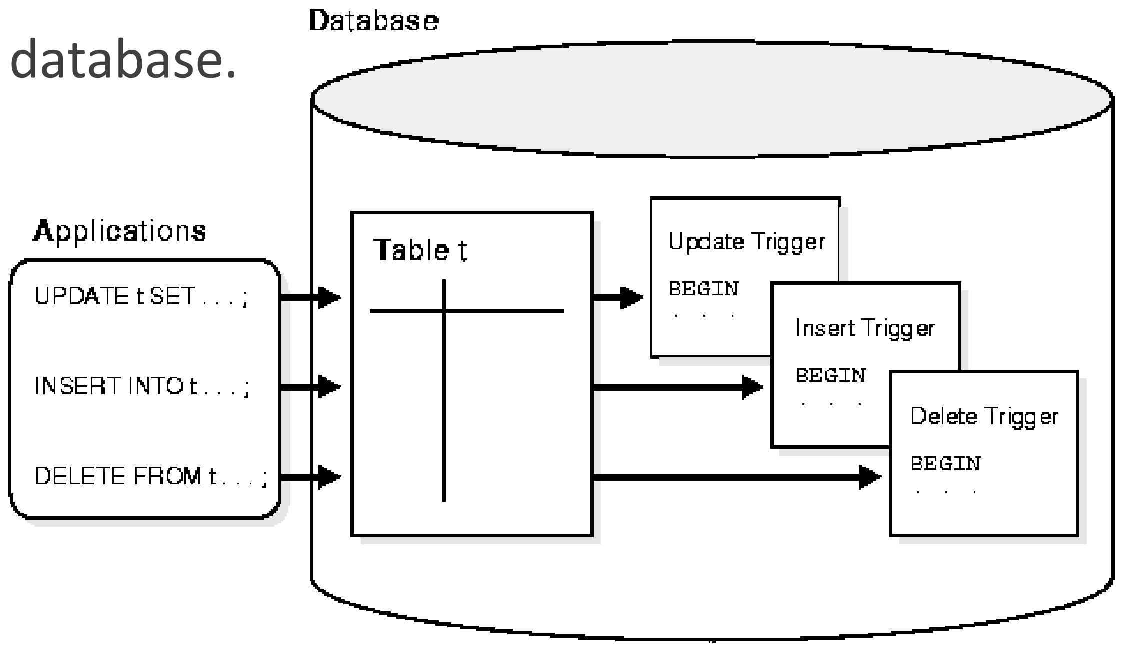
CSC430/530 – DATABASE MANAGEMENT SYSTEMS

OUTLINE

- Triggers.
  - Introduction.
  - Syntax.
- Views.
  - Introduction.
  - Syntax.
  - Implementation & updates.

TRIGGERS: INTRO

- **Triggers** are used to specify **automatic actions** that the database system will perform when certain **events** and **conditions** occur.
- **Why use triggers**?
  - **Actions** are performed **automatically** within the database.
    - No need to write any **extra** application **code**.
  - **Complex** application **logic** is executed "*close*" to the data.
  - Triggers increase the **complexity** and **flexibility** of your database.
- Possible trigger **use cases**:
  - Enforce complex **data integrity rules**.
  - Prevent **invalid DML** transactions from occurring.
  - Enhance complex **database security** rules.



TRIGGERS: FORMAT

- Write a trigger to create a default project for each new department inserted into database.

```
DELIMITER $$              /* change delimiter so semicolon doesn't indicate end */
CREATE TRIGGER NAME       /* give trigger a name */
AFTER/BEFORE              /* trigger either after or before the operation */
INSERT/UPDATE/DELETE      /* what operation will cause this to fire */
ON TABLE                  /* on which table */
FOR EACH ROW              /* for each row that will be or has been changed */
BEGIN                     /* begin the trigger code */
    /* series of sql statements where:
        NEW = row to be inserted / updated row
        OLD = row that was deleted / non-updated row
        SET = command to set a value or attribute
    */
END$$        /* end trigger code */
DELIMITER ; /* change delimiter back to default */
```

TRIGGERS: EXAMPLE

- Example: A trigger to create a default project for each new department inserted into database.

```
DELIMITER $$
CREATE TRIGGER New_Dept
/* fires after a completed insert into the department table */
AFTER INSERT ON Department
FOR EACH ROW
BEGIN
    INSERT INTO Project
    VALUES ( CONCAT(NEW.Dname, ' - Initialization'),
             CONCAT(NEW.DNumber, '1'),
             'Houston',
             NEW.Dnumber );
END$$
DELIMITER ;
/* "NEW" automatically refers to the row that was just inserted. Using dot
notation (like in Object Oriented programming), we can grab the values of
attributes from the row */
```

TRIGGERS: IF THEN

- We can also have **IF THEN** statements to check conditions and take appropriate actions.

  IF *condition*

  THEN *action*

- Condition(s) – anything that is allowed in a **WHERE** clause.
  - A predicate evaluated on the **database state**.
  - Determines whether the **action** is **executed**.
- Action(s) – individual (or sequence of) **SQL statement(s)** or stored **procedures**.
  - May involve **database updates**.

TRIGGERS: EXAMPLE

- Example: A trigger to update the salary of an employee with the average salary of the department where he/she works BEFORE INSERT(ing) the record in the employee table IF the salary is empty or NULL.

```
DELIMITER $$
CREATE TRIGGER Emp_Add_Trigger
BEFORE INSERT ON Employee /* fires before executing the insert */
FOR EACH ROW
BEGIN
    IF (NEW.Salary = "" OR NEW.Salary IS NULL)
    THEN
        /* use SET to change value of salary before it is inserted into table */
        SET NEW.Salary = (SELECT AVG(Salary)
                          FROM Employee E
                          WHERE E.Dno = NEW.Dno);
    END IF;
END$$
DELIMITER ;
```

TRIGGERS

- **AFTER** and **BEFORE**
  - **BEFORE** fires before the operation (insert/update/delete) is executed
  - **AFTER** fires after the operation is completed
- **NEW** v. **OLD**
  - With **INSERT**: Use NEW to refer to the new row being inserted
  - With **DELETE**: Use OLD to refer to the old row being deleted
  - With **UPDATE**: Use NEW to refer to the new row values and OLD to refer to the old (previous) row values that are being replaced
- If wanting multiple triggers (e.g. AFTER INSERT and AFTER UPDATE), you need to make one for each:

```
    CREATE TRIGGER emp_ins AFTER INSERT …
    CREATE TRIGGER emp_up AFTER UPDATE …
```

TRIGGERS: SIGNAL

- Signals can be used (which act similarily to throwing exceptions in Java)
- Syntax:

```
SIGNAL SQLSTATE 'XXXX' SET MESSAGE_TEXT = 'my message';
```

- Where:
  - 'XXXX' is a special code indicating what went wrong
    - Using "45000" means "unhandled user-defined exception."
    - This will block the action from completing
  - MESSAGE_TEXT is a custom message you can display
    - You can NOT use an expression directly to set the MESSAGE_TEXT value
    - But you can put your message into a variable and use that instead
- Find out more information at: https://dev.mysql.com/doc/refman/8.4/en/signal.html

TRIGGERS: EXAMPLE

- Example: A trigger showing the difference between NEW and OLD with an UPDATE.

```
DELIMITER $$
CREATE TRIGGER Trigger_New_And_Old
AFTER UPDATE ON Employee
FOR EACH ROW
BEGIN
    DECLARE msg VARCHAR(255);
    SET msg = CONCAT("changing ", OLD.fname, " to ", NEW.fname);
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = msg;
END$$
DELIMITER ;
/* This will show the old fname and new fname for any update
to employee. This is just used for showing the difference
between NEW and OLD, and also for showing signaling */
```

TRIGGERS: HANDS ON

- Write a trigger to enforce following constraint:
  - Dependent relationship must be either spouse, son, or daughter. If anything else, then display message – “Please, provide valid relationship (Spouse, Son or Daughter).”

TRIGGERS: HANDS ON

- Write a trigger to enforce following constraint:
  - Dependent relationship must be either spouse, son, or daughter, if anything else, then display message – “Please, provide valid relationship (Spouse, Son or Daughter).”

```
DELIMITER $$
CREATE TRIGGER DEPENDENT_RELATIONSHIP
BEFORE INSERT ON DEPENDENT
FOR EACH ROW
BEGIN
    DECLARE MSG VARCHAR(255);
    IF NEW.Relationship NOT IN ('Spouse', 'Daughter', 'Son')
    THEN
        SET MSG = 'Please, provide correct relationship (Spouse, Son or Daughter).';
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = MSG;
    END IF;
END$$
DELIMITER ;
```

VIEWS: INTRO

- **View** is a **single table** that is derived from **other tables** (*base tables* or *other views*).
  - Does not have to **exist** in **physical form** – considered to be a **virtual table**.
    - **Updates** are **limited**, but no **limitations** on **queries**.
- **View** is a way to specify (*joined*) **table** that is **referenced frequently** but does not need to be **defined physically**.
- Company database **example**:
  - Frequently referencing to **employee name** & **project name** he/she works on.
    - Rather than specifying a join of **EMPLOYEE**, **WORKS_ON** and **PROJECT** relation every time we issue the query, it is easier to define a **view** that specifies the **result** of this join.

VIEWS: SYNTAX

- CREATE VIEW is used to **define** new **view**.
  - Must provide a (*virtual*) **table name** (*view name*), a **list** of **attribute** names, and a **query** to specify the **contents**.
    - If some of the view attributes are **derived** by functions or arithmetic operations, then **renaming** those attributes is recommended.
- **Example**:
  - Create a view that provides information about the **employees** and the **projects** that they **work on**.

```
CREATE VIEW WORKS_ON_EXT AS
    SELECT E.Fname, E.Lname, P.Pname, W.Hours
    FROM EMPLOYEE E, PROJECT P, WORKS_ON W
    WHERE E.Ssn = W.Essn AND W.Pno = P.Pnumber;
```

VIEWS: IMPLEMENTATION & UPDATES

- Two main approaches for **views implementation**.
  - **Query modification**.
    - View query is transformed into a query that joins underlying base tables.
  - **View materialization**.
    - Temporary physical view table is created when the view is first created or queried.
- View always contains **updated data**.
  - **Immediate update**.
    - View is updated as soon as base tables are changed.
  - **Lazy update**.
    - View is updated when needed by a view query.
- MySQL natively supports "query modification" and "lazy update".
  - Essentially, views are just stored as queries that are re-run each time a view is used

VIEWS: HANDS ON

- Create a view that displays the number of employees and the total salary paid in each department.

VIEWS: HANDS ON

- Create a view that displays the number of employees and the total salary paid in each department.

```
CREATE VIEW DEPT_INFO AS
    SELECT Dname, COUNT(*), SUM(Salary)
    FROM DEPARTMENT D, EMPLOYEE E
    WHERE D.Dnumber = E.Dno
    GROUP BY Dname;
```

VIEWS: HANDS ON

- Create a view that displays the number of employees and the total salary paid in each department.

```
CREATE VIEW DEPT_INFO(Dept_name, No_of_emps, Total_salary) AS
    SELECT Dname, COUNT(*), SUM(Salary)
    FROM DEPARTMENT D, EMPLOYEE E
    WHERE D.Dnumber = E.Dno
    GROUP BY Dname;
```

VIEWS: Usage

- Can use a view in a select query just like a normal table

Given that this has already executed:

```
CREATE VIEW DEPT_INFO(Dept_name, No_of_emps, Total_salary) AS
    SELECT Dname, COUNT(*), SUM(Salary)
    FROM DEPARTMENT D, EMPLOYEE E
    WHERE D.Dnumber = E.Dno
    GROUP BY Dname;
```

Use it like a normal table:

```
SELECT Dept_name, No_of_emps
FROM DEPT_INFO
```

SUMMARY

- Purpose of triggers.
- Triggers syntax.
- Purpose of views.
- Views syntax.
- Views implementation & updates.