# CSE 215:SQL (Constraints and Triggers)

**Dr. Mohammed Eunus Ali**
**(eunus@cse.buet.ac.bd)**

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology (BUET)
Dhaka-1000, Bangladesh

(with some slides integrated from those of Jenifer Widom, Alon Halevy, Carlo Curino, and Michael Stonebraker.)

# Integrity Constraints (Review)

- **Constraint describes conditions that every *legal instance* of a relation must satisfy.**
  - Inserts/deletes/updates that violate ICs are disallowed.
  - Can be used to :
    - ensure application semantics (e.g., *sid* is a key), or
    - prevent inconsistencies (e.g., *sname* has to be a string, *age* must be < 200)

- ***Types of IC's*:**
  - Fundamental: Domain constraints, primary key constraints, foreign key constraints
  - General constraints : Check Constraints, Table Constraints and Assertions.

# Check or Table Constraints

```
CREATE TABLE  Sailors
          ( sid  INTEGER,
          sname  CHAR(10),
          rating  INTEGER,
          age  REAL,
          PRIMARY KEY  (sid),
          CHECK  ( rating >= 1
                    AND rating <= 10 ))
```

- Can use queries to express constraint.

# Explicit Domain Constraints

```
CREATE DOMAIN  values-of-ratings  INTEGER
          DEFAULT 1
          CHECK  ( VALUE >= 1 AND VALUE <= 10)
```

```
CREATE TABLE   Sailors
          ( sid  INTEGER,
          sname  CHAR(10),
          rating  values-of-ratings,
          age  REAL,
          PRIMARY KEY  (sid))
```

# More Powerful Table Constraints

- Constraint that Interlake boats cannot be reserved:

```
CREATE TABLE  Reserves
        ( sname  CHAR(10),
        bid  INTEGER,
        day  DATE,
        PRIMARY KEY  (bid,day),
        CONSTRAINT  noInterlakeRes
        CHECK  (`Interlake' <>
                        ( SELECT  B.bname
                        FROM  Boats B
                        WHERE  B.bid= bid)))
```

- If condition evaluates to FALSE, update is rejected.

# Table Constraints

- Associated with one table
- Only needs to hold TRUE when table is non-empty.

# Table Constraints with Complex CHECK

*Number of boats plus number of sailors is < 100*

```
CREATE TABLE   Sailors
        ( sid  INTEGER,
        sname  CHAR(10),
        rating  INTEGER,
        age  REAL,
        PRIMARY KEY  (sid),
        CHECK
        ( (SELECT COUNT (S.sid) FROM Sailors S)
        + (SELECT COUNT (B.bid) FROM Boats B)
        < 100 )
```

- Symmetric constraint, yet associated with Sailors.
- If Sailors is empty, the number of Boats tuples can be anything!

# Assertions
# ( Constraints over Multiple Relations)

- ASSERTION not associated with either table.

```
CREATE TABLE  Sailors
           ( sid  INTEGER,
           sname  CHAR(10),
           rating  INTEGER,
           age  REAL,
           PRIMARY KEY  (sid),
           CHECK
           ( (SELECT COUNT (S.sid) FROM Sailors S)
           + (SELECT COUNT (B.bid) FROM Boats B) < 100 )
```

*Number of boats plus number of sailors is < 100*

```
CREATE ASSERTION  smallClub
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT (B.bid) FROM Boats B) < 100 )
```

# Triggers (Active database)

- **Trigger: A procedure that starts automatically if specified changes occur to the DBMS**

- **Analog to a "daemon" that monitors a database for certain events to occur**

- **Three parts:**
  - Event (activates the trigger)
  - Condition (tests whether the triggers should run) [Optional]
  - Action (what happens if the trigger runs)

- **Semantics:**
  - When event occurs, and condition is satisfied, the action is performed.

# Triggers – Event,Condition,Action

- **Events could be :**

```
BEFORE|AFTER INSERT|UPDATE|DELETE ON <tableName>
```

e.g.: `BEFORE INSERT ON Professor`

- **Condition is SQL expression or even an SQL query (query with non-empty result means TRUE)**

- **Action can be many different choices :**
  - SQL statements , body of PSM (persistent store modules), and even DDL and transaction-oriented statements like “commit”.

# Example Trigger

**Assume our DB has a relation schema :**

**Professor (pNum, pName, salary)**

**We want to write a trigger that :**

**Ensures that any new professor inserted has salary >= 60000**

# Example Trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

     for what context  ?

BEGIN

     check for violation here ?


END;
```

# Example Trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

     FOR EACH ROW

BEGIN

       Violation of Minimum Professor Salary?

END;
```

# Example Trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

     FOR EACH ROW

BEGIN

   IF (:new.salary < 60000)
       THEN RAISE_APPLICATION_ERROR (-20004,
       ‘Violation of Minimum Professor Salary’);
   END IF;

END;
```

# Example trigger

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor
   FOR EACH ROW

DECLARE temp int;     -- dummy variable not needed

BEGIN
   IF (:new.salary < 60000)
        THEN RAISE_APPLICATION_ERROR (-20004,
        ‘Violation of Minimum Professor Salary’);
   END IF;

temp := 10;           -- to illustrate declared variables

END;
.
run;
```

# Details of Trigger Example

- **BEFORE INSERT ON Professor**
  - This trigger is checked before the tuple is inserted
- **FOR EACH ROW**
  - specifies that trigger is performed for each row inserted
- **:new**
  - refers to the new tuple inserted
- **If (:new.salary < 60000)**
  - then an application error is raised and hence the row is not inserted; otherwise the row is inserted.
- **Use error code: -20004;**
  - this is in the valid range

# Example Trigger Using Condition

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor
FOR EACH ROW
WHEN (new.salary < 60000)
BEGIN
   RAISE_APPLICATION_ERROR (-20004,          ‘Violation
   of Minimum Professor Salary’);
END;
.
run;
```

- Conditions can refer to  old/new values of tuples modified by the statement activating the trigger.

# Triggers: REFERENCING

```
CREATE TRIGGER minSalary BEFORE INSERT ON Professor

REFERENCING NEW as newTuple

FOR EACH ROW

WHEN (newTuple.salary < 60000)

BEGIN
   RAISE_APPLICATION_ERROR (-20004,
   ‘Violation of Minimum Professor Salary’);
END;
.
run;
```

# Example Trigger

```
CREATE TRIGGER minSalary
     BEFORE UPDATE ON Professor
REFERENCING OLD AS oldTuple NEW as newTuple
FOR EACH ROW
WHEN (newTuple.salary < oldTuple.salary)
BEGIN
  RAISE_APPLICATION_ERROR (-20004, ‘Salary
  Decreasing !!’);
END;
.
run;
```

- **Ensure that salary does not decrease**

# Another Trigger Example (SQL:99)

```
CREATE TRIGGER  youngSailorUpdate
    AFTER  INSERT ON SAILORS
REFERENCING NEW TABLE AS NewSailors
FOR EACH STATEMENT
    INSERT
          INTO YoungSailors(sid, name, age, rating)
          SELECT sid, name, age, rating
          FROM NewSailors N
          WHERE N.age <= 18
```

# Row vs Statement Level Trigger

- Row level:  activated once per modified tuple
- Statement level: activate once per SQL statement

- Row level triggers can access new data, statement level triggers cannot always do that (depends on DBMS).

- Statement level triggers will be more efficient if we do not need to make row-specific decisions

# Row vs Statement Level Trigger

- **Example: Consider a relation schema**

  **Account (num, amount)**

  **where we will allow creation of new accounts only during normal business hours.**

# Example: Statement level trigger

```
CREATE TRIGGER MYTRIG1
BEFORE INSERT ON Account
FOR EACH STATEMENT                --- is default
BEGIN
   IF (TO_CHAR(SYSDATE,’dy’) IN (‘sat’,’sun’))
   OR
   (TO_CHAR(SYSDATE,’hh24:mi’) NOT BETWEEN ’08:00’ AND
  ’17:00’)
    THEN
      RAISE_APPLICATION_ERROR(-20500,’Cannot   create
  new account now !!’);
   END IF;
END;
```

# When to use BEFORE/AFTER

- **Based on efficiency considerations or semantics.**
- **Suppose we perform statement-level after insert, then all the rows are inserted first, then if the condition fails, and all the inserted rows must be "rolled back"**
- **Not very efficient !!**

To stop an action, either raise application error before, or rollback after.

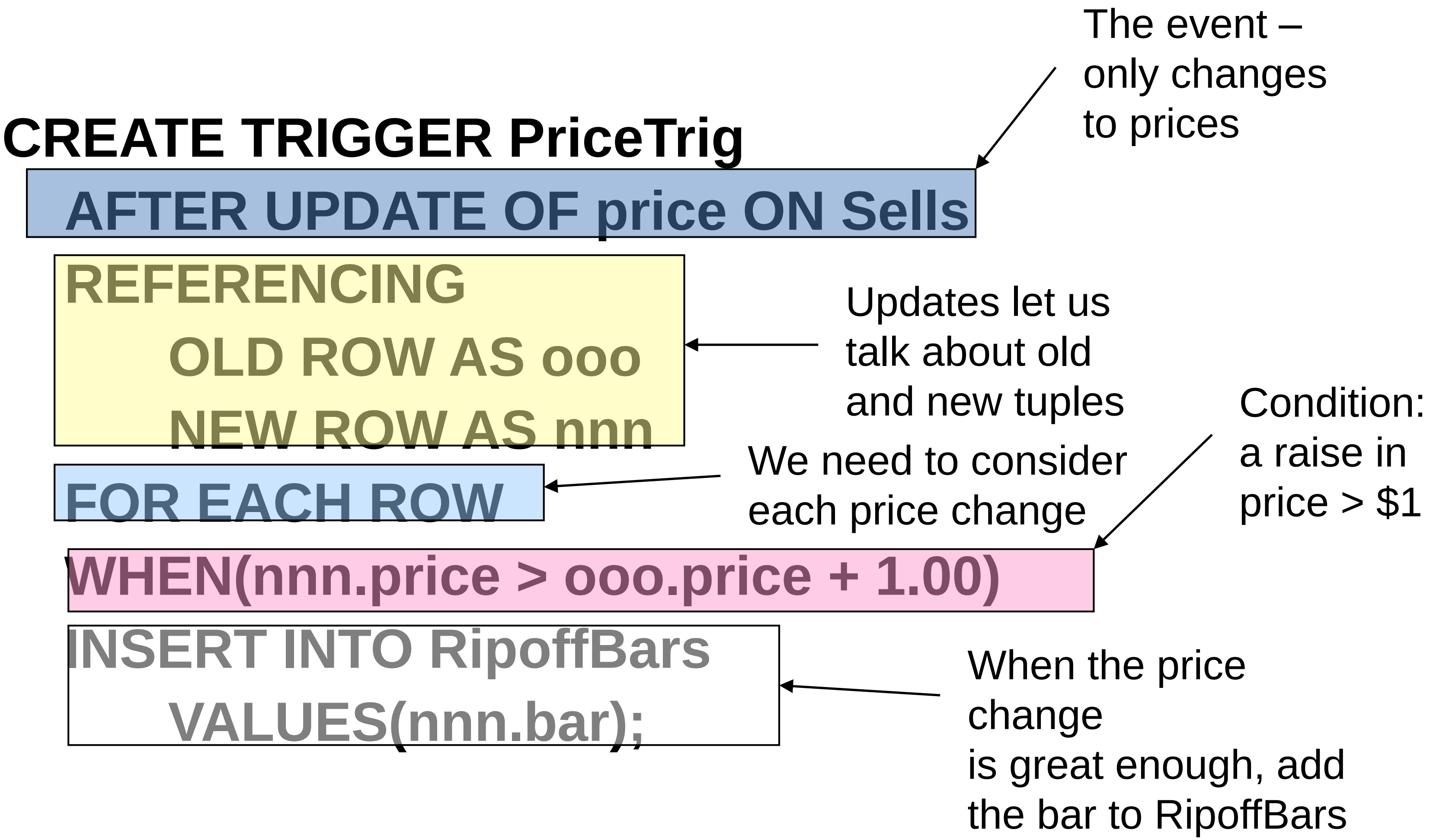
# Combining multiple events into one trigger

```
CREATE TRIGGER salaryRestrictions
AFTER INSERT OR UPDATE ON Professor
FOR EACH ROW
BEGIN
IF (INSERTING AND :new.salary < 60000) THEN
  RAISE_APPLICATION_ERROR (-20004, 'below min
  salary'); END IF;
IF (UPDATING AND :new.salary < :old.salary)
  THEN RAISE_APPLICATION_ERROR (-20004, ‘Salary
  Decreasing !!'); END IF;
END;
```

# Summary :  Trigger Syntax

```
CREATE TRIGGER <triggerName>
BEFORE|AFTER   INSERT|DELETE|UPDATE
   [OF <columnList>] ON <tableName>|<viewName>
   [REFERENCING [OLD AS <oldName>] [NEW AS <newName>]]
[FOR EACH ROW] (default is “FOR EACH STATEMENT”)
[WHEN (<condition>)]
<PSM body>;
```

# The Trigger



```
CREATE TRIGGER PriceTrig
  AFTER UPDATE OF price ON Sells
  REFERENCING
      OLD ROW AS ooo
      NEW ROW AS nnn
  FOR EACH ROW
  WHEN(nnn.price > ooo.price + 1.00)
  INSERT INTO RipoffBars
      VALUES(nnn.bar);
```

The event – only changes to prices

Updates let us talk about old and new tuples

We need to consider each price change

Condition: a raise in price > $1

When the price change is great enough, add the bar to RipoffBars

# Some Points about Triggers

- **Check the system tables :**
  - `user_triggers`
  - `user_trigger_cols`
  - `user_errors`

- **`ORA-04091`: mutating relation problem**
  - In a **row level trigger**, you **cannot** have the body refer to the table specified in the event

- **Also `INSTEAD OF` triggers can be specified on views**

# To Show Compilation Errors

```
SELECT line, position, text
FROM user_errors
WHERE name = 'MY_TRIGGER'
    AND TYPE = 'TRIGGER‘
```

- In SQL*Plus, you can also use the following shortcut:

```
SQL> SHOW ERRORS TRIGGER MY_TRIGGER
```

# Constraints versus Triggers

- **Constraints are useful for database consistency**
  - Use IC  when sufficient
  - More opportunity for optimization
  - Not restricted into insert/delete/update

- **Triggers  are flexible and powerful**
  - Alerters
  - Event logging for auditing
  - Security enforcement
  - Analysis of table accesses (statistics)
  - Workflow and business intelligence …

- **But can be hard to understand ……**
  - Several triggers      (Arbitrary order → unpredictable !?)
  - Chain triggers         (When to stop ?)
  - Recursive triggers  (Termination?)

# Summary

- SQL allows specification of rich integrity constraints  and their efficient maintenance

- Triggers respond to changes in the database: powerful for enforcing application semantics