Triggers

CS 341 Database Systems

Triggers

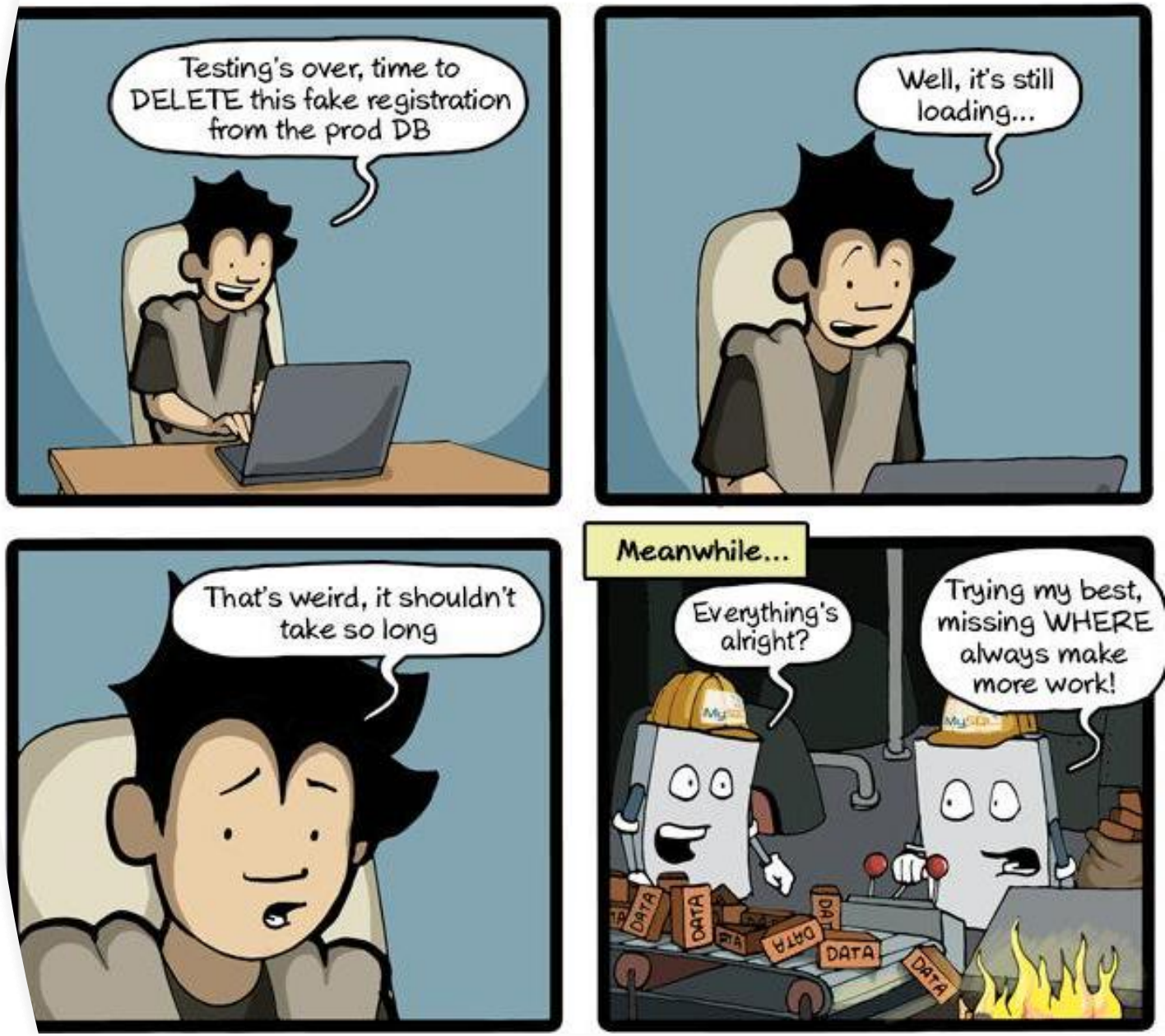
- A stored procedure without parameters
- Automatically fired with DML commands
- Why use triggers?
  - Another way to implement and re-enforce the Business Rules

3 Kind of Triggers

DML	Instead-of triggers	System triggers
INSERT UPDATE and/or DELETE	Used with views only SQL trigger that is processed “instead of” an SQL UPDATE, DELETE or INSERT statement	Fires when a system event (action) occurs

Firing Triggers

- **Firing the trigger**
  - The act of executing a trigger
- They are implicitly executed whenever the trigger event happens (e.g., **INSERT**, **UPDATE** or **DELETE**)



3 Characteristics of Triggers

Timing	Statement (type of action/type of event)	Level
• Refers to whether the trigger fires BEFORE or AFTER the triggering action	• DML → INSERT, UPDATE and/or DELETE • DDL → CREATE, ALTER or DROP • Database operation → LOGON, LOGOFF, STARTUP or SHUTDOWN	• Refers to statement-level (default) or row-level trigger

Characteristics of Triggers

Statement-level triggers

- Fire once per SQL statement
- Cannot read the values of the columns (using correlation values)

Row-level triggers

- Fire many times, once for each row affected
- Can evaluate the values of each column for that row (using correlation values)

Benefits of Triggers

Data integrity: Triggers allow you to *enforce complex business rules and constraints* at the database level, ensuring that data remains consistent and accurate.

Automation: *Automate repetitive or complex tasks* by executing predefined actions whenever a specified event occurs. Reduces the need for manual intervention and improves efficiency.

Audit trails: Triggers can be used to track changes made to data, such as *logging* modifications in a separate audit table. This helps in auditing and maintaining a history of data changes.

Data validation: Triggers can perform *additional validation* checks on data before it is *inserted, updated, or deleted,* ensuring that only valid and conforming data is stored in the database.

Triggers: Syntax

```
CREATE OR REPLACE TRIGGER trigger_Name
   {BEFORE|AFTER} {INSERT|UPDATE| DELETE}
   [OF column_name] ON table_name
   [FOR EACH ROW]
   [WHEN trigger_condition]
DECLARE
    -- variable declaration goes here
BEGIN
    -- statements goes here
END [trigger_Name];
```

Triggers: Predicates

```
CREATE OR REPLACE TRIGGER trigger_Name
    {BEFORE|AFTER} {INSERT|UPDATE| DELETE}
    [OF column_name] ON table_name
    [FOR EACH ROW]
    [WHEN trigger_condition]
DECLARE
     -- variable declaration goes here
BEGIN
    IF INSERTING THEN
            -- statements here
    ELSIF UPDATING THEN
            -- statements here
    ELSIF DELETING THEN
            -- statements here
    END IF;
END[trigger_Name];
```

Triggers: Recommended Naming Convention

- **Table_Name_[A|B][I|U|D][S|R]**
  - [A|B] AFTER or BEFORE
  - [I|U|D] INSERT, UPDATE, DELETE
  - [S|R] statement-level or row-level

Naming Conventions

Item	Naming Convention	Note
primary keys	*_pk	* = tablename
foreign keys	*_fk1	* = tablename; # = a sequential number
	*_fk2	
	*_fk#	
unique keys	*_u1	* = tablename
	*_u2	* = tablename
	*_u#	# represents a sequential number
checks	*_ck1	* = tablename
	*_ck2	* = tablename
	*_ck#	# represents a sequential number
sequences	*_sequence	* = field name
script files	*.sql	* can be any name you choose
spooled files	*.lst	* can be any name you choose (e.g. TEST.LST)
cursors	c_	
exceptions	e_	
records	t_	Explicit records
variables	v_	
parameters	p_	
triggers	*_{B|A}{I|U|D}{S|R}	Choose either B=Before, A=After
		I=Insert, U=Update, D=Delete
		S=Statement-level, R=row-level

Connect to New User

```
SQL> alter session set "_ORACLE_SCRIPT"=true;

Session altered.

SQL> create user lab09user identified by 123;

User created.

SQL> grant connect, resource, dba, unlimited tablespace to lab09user;

Grant succeeded.

SQL> conn lab09user;
Enter password:
Connected.
```

Example

- **Create a table**

```
CREATE TABLE orders (
        order_id number(5) PRIMARY KEY,
        quantity number(4),
        cost_per_item number(6,2),
        total_cost number(8,2),
        updated_date date,
        updated_by varchar2(50) );
```

Insert a Record

```
SQL> INSERT INTO Orders( order_id, quantity, cost_per_item, total_cost) VALUES (1,4, 200,800);

1 row created.

SQL> select * from orders;

  ORDER_ID   QUANTITY COST_PER_ITEM TOTAL_COST UPDATED_D
---------- ---------- ------------- ---------- ---------
UPDATED_BY
----------------------------------------------------
         1          4           200        800
```

Create a Trigger - Example

```
SQL> CREATE OR REPLACE TRIGGER orders_before_insert
BEFORE INSERT ON Orders
FOR EACH ROW
DECLARE
v_username varchar2(50);
BEGIN
-- Find username of person performing UPDATE on the table
SELECT user INTO v_username FROM dual;
-- Update updated_date field to current system date
:new.updated_date := sysdate;
-- Update updated_by field to the username of the person performing the UPDATE
:new.updated_by := v_username;
END;
```

```
CREATE TABLE ORDERS (
        order_id number(5) PRIMARY KEY,
        quantity number(4),
        cost_per_item number(6,2),
        total_cost number(8,2),
        updated_date date,
        updated_by varchar2(50) );
```

Example

- Insert record

SQL> INSERT INTO Orders(order_id, quantity, cost_per_item, **total_cost**) VALUES (1,4, 200,**800**);

- *Total_cost is still manually calculated and entered.*
- Derived attributes can be calculated and stored upon insertion

Create a Trigger

```
SQL> CREATE OR REPLACE TRIGGER orders_before_insert
      BEFORE insert ON Orders
      FOR EACH ROW
      DECLARE
        v_username varchar2(50);
      BEGIN
        SELECT user INTO v_username
        FROM dual;
        :new.total_cost := :new.quantity * :new.cost_per_item;
       :new.updated_date := sysdate;
       :new.updated_by := v_username;
   END;
```

Trigger Created

```
SQL> CREATE OR REPLACE TRIGGER orders_before_insert
  2           BEFORE insert ON Orders
  3           FOR EACH ROW
  4           DECLARE
  5              v_username varchar2(50);
  6           BEGIN
  7              SELECT user INTO v_username
  8              FROM dual;
  9             :new.total_cost := :new.quantity * :new.cost_per_item;
 10            :new.updated_date := sysdate;
 11            :new.updated_by := v_username;
 12       END;
 13  /

Trigger created.
```

Example

- Insert record

SQL> INSERT INTO orders(order_id, quantity, cost_per_item) VALUES (7,4, 200);

```
SQL> INSERT INTO Orders( order_id, quantity, cost_per_item) VALUES (2,4, 300);

1 row created.

SQL> select * from orders;

  ORDER_ID   QUANTITY COST_PER_ITEM TOTAL_COST UPDATED_D
---------- ---------- ------------- ---------- ---------
UPDATED_BY
--------------------------------------------------
         1          4           200        800


         2          4           300       1200 29-OCT-23
LAB09USER
```

Triggers: Correlation Identifiers

- **Correlation identifiers**
  - Row-level triggers can access individual column values
  - **:old** and **:new** (called pseudo records)
  - Syntactically treated as records of type

    triggering_table%ROWTYPE
  - Reference fields within the pseudo records using dot notation (just like implicit records)

    :old.salary **:new.salary**

Triggers: Correlation Identifiers

:old.column_name

- Contains the value prior to the change
- NULL for INSERT statements

:new.column_name

- Contains the value after the change
- NULL for DELETE statements

Triggers: Correlation Identifiers

- Testing :old value against :new value will only work for **UPDATE** statement example **:old.salary <> :new.salary**
- **INSERT** has only **:new** value
- **DELETE** has only **:old** value

Triggers: Correlation Identifiers

Approach 1

```
CREATE OR REPLACE TRIGGER employee_BUR
   BEFORE UPDATE OF SALARY ON EMPLOYEE
   FOR EACH ROW
DECLARE
   v_Sal_Difference  NUMBER;
BEGIN
   IF :old.salary <> :new.salary THEN
        -- do something here
   END IF;
END employee_BUR;
/
```

Approach 2

```
CREATE OR REPLACE TRIGGER employee_BUR
   BEFORE UPDATE OF SALARY ON EMPLOYEE
   FOR EACH ROW
   WHEN (OLD.SALARY <> NEW.SALARY)
DECLARE
   v_Sal_Difference      NUMBER;
BEGIN
   -- do something here
END employee_BUR;
/
```

NOTE: When using Approach 2, do NOT include the colons before the words OLD and NEW in the WHEN clause.

Example of correlation identifiers

```
/* Log any changes to employee salary where the increase is greater
   than 10% */

CREATE Table Employee_Big_Change
   (actionDate date, osal number, nsal number);
```

Example of correlation identifiers

```
/* Log any changes to employee salary where the increase is greater than 10% */

CREATE OR REPLACE TRIGGER employee_BUR
BEFORE UPDATE ON employees
FOR EACH ROW
WHEN (NEW.salary/OLD.salary > 1.1)
BEGIN
        INSERT INTO Employee_Big_Change
         VALUES (sysdate,:OLD.salary,:NEW.salary);
END employee_BUR;
```

Example of correlation identifiers

```
/* Log any changes to employee salary where the new salary is not equal to old
   salary*/

CREATE OR REPLACE TRIGGER employee_BUR
BEFORE UPDATE ON employees
FOR EACH ROW
   WHEN (NEW.salary<>OLD.salary )
BEGIN
        INSERT INTO Employee_Big_Change
        VALUES (sysdate,:OLD.salary,:NEW.salary);
END employee_BUR;
```

Row Level Trigger's Example

Step #1 (create table) - create table new_emp(tdate date, eid number, sal number);

Step#2 (write trigger)

```
/* Log new employees(inserts) or updates to employee salary */
CREATE OR REPLACE TRIGGER employee_ins
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
 if INSERTING then
      insert into new_emp
      values (sysdate,:new.employee_id,:new.salary);
 else
   insert into new_emp
   values (sysdate,:old.employee_id, :new.salary);
end if;
END employee_ins;
```

Row Level Trigger's Example

Step #1 (create table)
create table Dept_stat (DPTNO number, Total_emps number);

Step#2 (write trigger)

```
CREATE OR REPLACE TRIGGER Update_Dept_Stat
BEFORE INSERT OR UPDATE OR DELETE ON EMPLOYEES
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    UPDATE Dept_Stat SET Total_emps = Total_emps + 1
           WHERE   DPTNO=:NEW.Department_id;
    END IF;
  IF DELETING THEN
    UPDATE Dept_Stat SET Total_emps = Total_emps - 1
           WHERE  DPTNO = :OLD.Department_id;
    END IF;
  IF UPDATING THEN
   IF :OLD.Department_id <> :NEW.Department_id THEN
    UPDATE Dept_Stat SET Total_emps = (Total_emps - 1)
           WHERE DPTNO = :OLD.Department_id;
    UPDATE Dept_Stat SET Total_emps = (Total_emps + 1)
           WHERE DPTNO = :NEW.Department_id;
   END IF;
  END IF;
END Update_Dept_Stat;
```

Restrictions on Triggers

- Cannot have any *transaction control* statements (e.g. COMMIT, ROLLBACK or SAVEPOINT)
- Cannot call procedures or functions that issue any transaction control statements
- Triggers cannot declare variables of large datatypes e.g clob, blob, long, long raw

Statement - Level

Statement Level Trigger's Example

```
CREATE OR REPLACE TRIGGER student_AIUDS
    AFTER INSERT OR UPDATE OR DELETE ON Student
DECLARE
    CURSOR c_Statistics IS
        SELECT major, COUNT(*) AS total_students, SUM(current_credits) AS total_credits
        FROM students
        GROUP BY major;
BEGIN
        FOR v_StatsRecord in c_Statistics LOOP
        INSERT INTO major_stats VALUES(v_StatsRecord.major,
        v_StatsRecord.total_credits, v_StatsRecord.total_students);
    END LOOP;
END;
```

Trigger Usage:
Summary data: use a statement-level trigger

```
CREATE OR REPLACE TRIGGER patient_AIDUS
    AFTER INSERT OR DELETE OR UPDATE ON patient
DECLARE
    CURSOR c_stat IS
        SELECT doctor_ID, COUNT(*) total_patients
        FROM patient
        GROUP BY doctor_ID;
BEGIN
    FOR v_StatRec IN c_stat LOOP
        UPDATE doctor_stats
        SET total_patients = v_StatRec.total_patients
        WHERE Doctor_ID = v_StatRec.Doctor_ID;

        IF SQL%NOTFOUND THEN
            INSERT INTO doctor_stats VALUES(v_StatRec.Doctor_ID, v_StatRec.total_patients);
        END IF;
    END LOOP;
END patient_AIDUS;
/
```

Another example would be in an order processing system:

Define the trigger on the orderDetail table to update the total on the OrderHeader table.

Trigger Usage: *Overriding the values supplied in an INSERT/UPDATE statement by changing the :new correlation value*

- Use BEFORE INSERT or BEFORE UPDATE row-level trigger

```
CREATE SEQUENCE SEQUENCE_PATIENTID INCREMENT BY 1 START WITH 1;

CREATE OR REPLACE TRIGGER patient_BIR
        BEFORE INSERT ON PATIENT
        FOR EACH ROW
BEGIN
        SELECT sequence_patientID.NEXTVAL
        INTO :new.patientID
        FROM DUAL;
END patient_BIR;
/
SQL > INSERT INTO patient (Fname, lname) VALUES ('Bob', 'Smith');
/* Note that Patient ID is automatically generated by the trigger. */
```

```
CREATE OR REPLACE TRIGGER patient_bds
BEFORE DELETE ON patient
DECLARE
    e_weekend_error          EXCEPTION;
    e_not_supervisor         EXCEPTION;
BEGIN
    IF TO_CHAR(SYSDATE, 'DY') = 'SAT' OR
            TO_CHAR(SYSDATE, 'DY') = 'SUN' THEN
            RAISE e_weekend_error;
    END IF;
    IF SUBSTR(user, 1, 3) <> 'SUP' THEN
            RAISE e_not_supervisor;
     END IF;
EXCEPTION
    WHEN e_weekend_error THEN
    RAISE_APPLICATION_ERROR (-20001,
            'Deletions allowed Mon thru Fri only');
    WHEN e_not_supervisor THEN
    RAISE_APPLICATION_ERROR (-20002,
            'You '||user||' are not authorised to perform deletions');
END;
/
```

Using Raise Application Error in Triggers

The raise_application_error is actually a procedure defined by Oracle that allows the developer to raise an exception and associate an error number and message with the procedure.

System Triggers

To create HR_USERS_LOG

Step#1: Create the HR_USERS_LOG table

```
CREATE TABLE hr_users_log (
        user_name VARCHAR2(30),
        activity VARCHAR2(20),
        event_date DATE
);
```

HR_LOGON_TRIGGER

Step#2: Create hr_logon_trigger

```
CREATE OR REPLACE TRIGGER hr_logon_trigger
AFTER LOGON ON HR.SCHEMA
BEGIN
INSERT INTO hr_users_log (user_name, activity, event_date) VALUES
(USER, 'LOGON', to_date(SYSDATE, 'yyyy-MM-dd HH24:mi:ss'));
END;
```

Enabling & Disabling Triggers

- **ALTER TRIGGER trigger_name**
  {ENABLE | DISABLE}
  Ex: ALTER TRIGGER Employee_BIUR DISABLE;

- **ALTER TABLE table_name**
  {ENABLE | DISABLE} ALL TRIGGERS;
  Ex: ALTER TABLE emp ENABLE ALL TRIGGERS;

- **DROP TRIGGER trigger_name;**
  Ex: DROP TRIGGER Employee_BIUR;
  When a table is dropped, all triggers for that table are deleted from the Data Dictionary

Enabling & Disabling Triggers

- In order to disable, substitute DISABLE for ENABLE
- By default, all triggers are enabled when they are created
- When a trigger is disabled, it still exists in the Data Dictionary but is never fired
- The STATUS column of USER_TRIGGERS contain either DISABLED or ENABLED value

Data Dictionary

- **user_triggers view :** trigger_type, table_name, triggering_event, status
  - To list all of the triggers that you have created:

    SELECT trigger_type, table_name, triggering_event, status
    FROM **user_triggers;**

- Other views that list triggers
  - **all_triggers:** List triggers that are accessible to current user (but might be owned by a different user)
  - **dba_triggers:** List all triggers in the database

USER_TRIGGERS

```
SQL> Select trigger_name, trigger_type, status from user_triggers;

TRIGGER_NAME                TRIGGER_TYPE     STATUS
--------------------------- ---------------- --------
LOGEMPCHANGE                 AFTER EACH ROW   ENABLED

SQL> select trigger_name, table_name from user_triggers;

TRIGGER_NAME                TABLE_NAME
--------------------------- ---------------------------
LOGEMPCHANGE                 EMPLOYEE

SQL> select description from user_triggers;

DESCRIPTION
--------------------------------------------------------------------------------
LogEmpChange
      AFTER INSERT OR DELETE OR UPDATE ON Employee
      FOR EACH ROW
```

Show errors trigger [name]

```
  1  CREATE OR REPLACE TRIGGER student_AIUDS
  2     AFTER INSERT OR UPDATE OR DELETE ON student
  3  DECLARE
  4     CURSOR c_Statistics IS
  5              SELECT major, COUNT(*) total_students, SUM(current_credits), tot
al_credits
  6              FROM students
  7              GROUP BY major;
  8  BEGIN
  9     DELETE * FROM major_stats;
 10     FOR v_StatsRecord in c_Statistics LOOP
 11              INSERT INTO major_stats VALUES(v_StatsRecord.major,
                 v_StatsRecord.total_credits, v_StatsRecord.total_students);
 12     END LOOP;
 13* END;
SQL> /

Warning: Trigger created with compilation errors.

SQL> show errors trigger student_AIUDS
Errors for TRIGGER STUDENT_AIUDS:

LINE/COL ERROR
-------- -----------------------------------------------------------------
7/2      PL/SQL: SQL Statement ignored
7/9      PL/SQL: ORA-00903: invalid table name
```

Mutating Tables

- *When a table is **mutating**, it is **changing**. If the change is taking place and you try to make another change in the middle of the first change, Oracle will issue a mutating table error with the error code ORA-04091.*
  - A mutating table error happens when a row-level trigger tries to access the same table that triggered it, either directly or indirectly.
  - For example, suppose you have a table called EMPLOYEES that has a trigger called TRG_EMPLOYEES that fires after each update on the table.
  - The trigger queries the EMPLOYEES table to calculate the average salary of all employees and inserts it into another table called SALARY_STATS. If you update one row in the EMPLOYEES table, the trigger will fire and try to query the same table, but the table is in an inconsistent state because the update is not yet committed.
  - *This will cause a mutating table error.*

Solving Mutating tables

1. *Use statement-level triggers instead of row-level triggers.* Statement-level triggers fire once for each DML statement, not for each affected row, so they do not cause conflicts with the table state. However, statement-level triggers have some limitations.
2. *Or use compound triggers*, which are a special type of triggers that combine the logic of multiple triggers into one. Compound triggers have four sections: before statement, before each row, after each row, and after statement.
3. *Or re-evaluate the need of the triggers.* Maybe it is better to use stored procedures, views and constraints to implement business logic and data validation.


IBA