Procedural Extension to SQL using Triggers - Lecture 2

Dr Akhtar AlI

Content

Limitations of Relational Data Model

- **Database *vs.* Information System (IS)**
  - **DBMS manages data *regardless of* its usage**
  - **IS processes information *with respect to* its usage**
- **Data model *vs.* system architecture**
  - **Data model does not give interpretation in terms of the application domain**
    - **e.g. relational model, hierarchical model**
  - **IS architecture is developed so that the data can be interpreted as information about a particular applied domain**
    - **e.g. HR information, financial information, sales information**

Event-Condition-Action (ECA)

- **E**vent occurs in databases
  - e.g. addition of a new row, deletion of a row
- **C**onditions are checked
  - e.g. Is batch complete? Has student passed?
- **A**ctions are executed if conditions are satisfied
  - e.g. send batch to supplier, congratulate student

Extending Information Processing Capabilities of DBMS using Triggers

- Processing of database content, performed by the DBMS engine itself, not by the application client
  - execution of the trigger (**E**vent)
- Initiated by certain specified condition, depending on the type of the trigger
  - firing of the trigger (**C**ondition)
- All data actions performed by the trigger execute within the same transaction in which the trigger fires, but in a separate session (**A**ction)
  - Triggers are checked for different privileges as necessary for the processed data
  - Cannot contain transaction control statements (COMMIT, SAVEPOINT, ROLLBACK not allowed)

Database Triggers in SQL

- Not specified in SQL-92, but standardized in SQL3 **(SQL1999)**
- Available in most enterprise DBMSs (Oracle, IBM DB2, MS SQL server) and some public domain DBMSs (Postgres)
  - but not present in smaller desktop (Oracle Lite) and public domain DBMS (MySQL)
- Some vendor DBMS permit native extensions to SQL for specifying the triggers
  - e.g. PL/SQL in Oracle, Transact SQL in MS SQL Server
- Some DBMS also allow use of general purpose programming language instead of SQL
  - e.g. C/C++ in Poet, Java in Oracle, C#/VB in SQL Server
- Some DBMS extend the triggers beyond tables
  - for example also to views as in Oracle

Types of SQL Triggers

- How many times should the trigger body execute when the triggering event takes place?
  - **Per statement:** the trigger body executes once for the triggering event. This is the default.
  - **For each row:** the trigger body executes once for each row affected by the triggering event.
- When the trigger can be fired
  - Relative to the execution of an SQL DML statement (before or after or instead of it)
  - Exactly in a situation depending on specific system resources (e.g. signal from the system clock, expiring timer, exhausting memory)

Statement and Row Triggers

Example 1: Monitoring Statement Events

```
SQL> INSERT INTO dept (deptno, dname, loc)
  2  VALUES (50, 'EDUCATION', 'NEW YORK');
```

Execute only once even if multiple rows affected

Example 2: Monitoring Row Events

```
SQL> UPDATE emp
  2  SET sal = sal * 1.1
  3  WHERE deptno = 30;
```

Execute for each row of the table affected by the event

Firing Sequence of Database Triggers on a Single Row

DEPT table

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

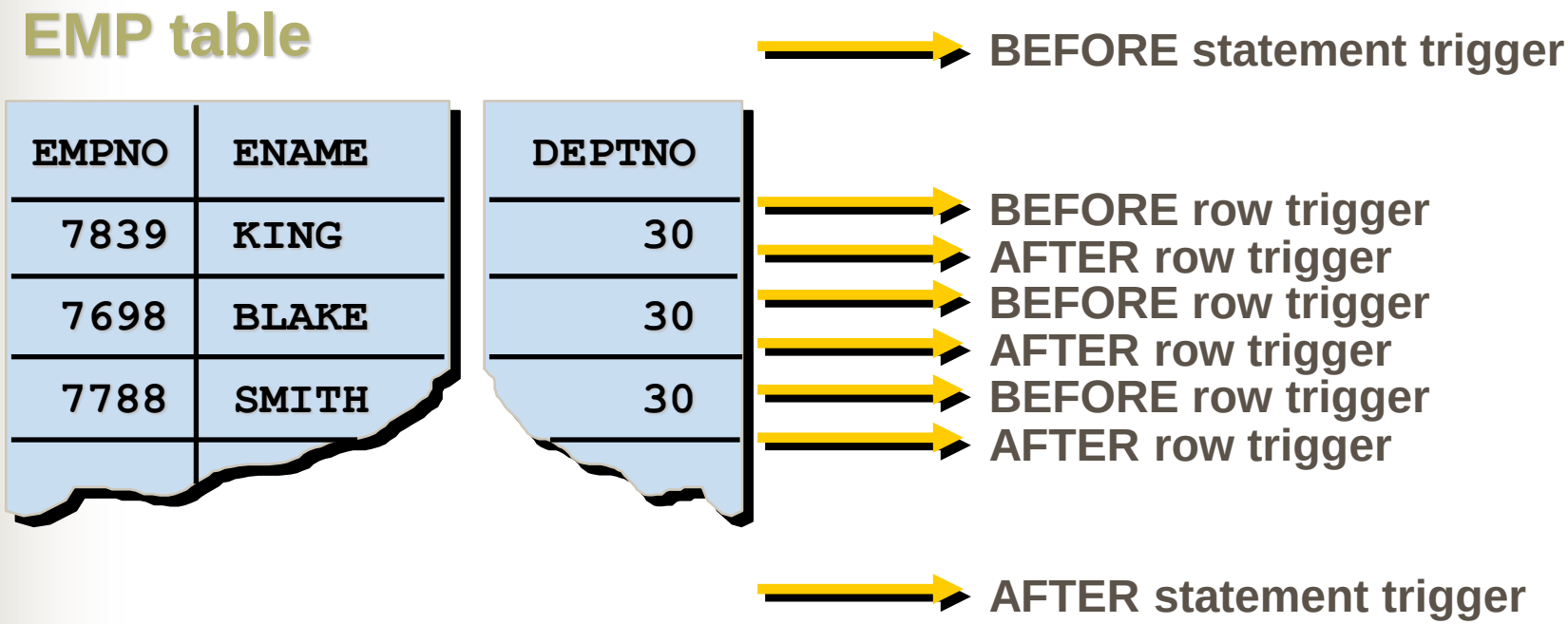
BEFORE statement trigger

BEFORE row trigger

AFTER row trigger

AFTER statement trigger

Firing Sequence of Database Triggers on Multiple Rows



Syntax for creating triggers in SQL

- **Trigger name** - unique within one database schema
- **Timing** - depends on the order of controlled events (before or after or instead of)
- **Triggering event** - event which fires the trigger **(E)**
- **Filtering condition** - checked when the triggering event occurs **(C)**
- **Target** - table (or view) against which the trigger is fired; they should be both created within the same schema
- **Trigger Parameters** - parameters used to denote the record columns; preceded by colon
  - **:new, :old for new and old versions of the values respectively**
- **Trigger action** - SQL statements, executed when the trigger fires; surrounded by **Begin** ... **End (A)**

Syntax for Creating Statement Triggers

```
CREATE [OR REPLACE] TRIGGER trigger_name
    timing event1 [OR event2 OR event3]
    ON table_name
BEGIN
  SQL statements;
END;
```

The trigger body consisting of *SQL statements* will be executed only ***once*** according to the prescribed *timing*, when the *event1 (event2, event3)* occurs against the monitored table in question *table_name.*

Example: Registering Operations

```
SQL> CREATE TRIGGER increase_salary_trg
  2          BEFORE UPDATE OF sal
  3          ON emp
  4  BEGIN
  5    INSERT INTO sal_hist(increased, changedOn)
  6           VALUES ('YES', SYSDATE);
  7  END;
  8  /
```

Trigger name: `increase_salary_trg`
Timing: `BEFORE` executing the statement
Triggering event: UPDATE of `sal` column
Target: `emp` table
Trigger action: INSERT values INTO `sal_hist table`

Syntax for Creating Row Triggers

```
CREATE [OR REPLACE] TRIGGER trigger_name
       timing event1 [OR event2 OR event3]
       ON table_name
       [REFERENCING OLD AS old | NEW AS new]
       FOR EACH ROW
       [WHEN condition]
BEGIN
  SQL statements;
END
```

The trigger body consisting of *SQL statements* will be executed once ***for each row*** affected by *event1* (*event2*, *event3*) in the table named *table_name* subject to the additional *condition*.

Example: Calculating Derived Columns

```
SQL>CREATE OR REPLACE TRIGGER derive_commission_trg
  2 BEFORE UPDATE OF sal ON emp
  3 FOR EACH ROW
  4 WHEN (new.job = 'SALESMAN')
  5 BEGIN
  6   :new.comm := :old.comm * (:new.sal/:old.sal);
  7 END;
  8 /
```

Trigger name:	derive_commission_trg
Timing:	BEFORE executing the statement
Triggering event:	UPDATE of sal column
Filtering condition:	job = 'SALESMAN'
Target:	emp table
Trigger parameters:	old, new
Trigger action:	calculate the new commission to be updated

Note: no (colon :) before new in WHEN

Trigger Execution order

1. Execute all BEFORE STATEMENT triggers
2. Disable temporarily all integrity constraints recorded against the table
3. Loop for each row in the table
   - Execute all BEFORE ROW triggers
   - Execute the SQL statement against the row and perform integrity constraint checking of the data
   - Execute all AFTER ROW triggers
4. Complete deferred integrity constraint checking against the table
5. Execute all AFTER STATEMENT triggers

Controlling Triggers using SQL

- **Disable or Re-enable a database trigger**

```
ALTER TRIGGER trigger_name  DISABLE | ENABLE
```

- **Disable or Re-enable all triggers for a table**

```
ALTER TABLE table_name   DISABLE | ENABLE  ALL TRIGGERS
```

- **Removing a trigger from the database**

```
DROP TRIGGER trigger_name
```

Using Database Triggers for Information Processing

- **Auditing Table Operations**
  - each time a table is accessed auditing information is recorded against it
- **Tracking Record Value Changes**
  - each time a record value is changed the previous value is recorded
- **Protecting Database Referential Integrity:** if foreign key points to changing records
  - referential integrity must be maintained
- **Maintenance of Semantic Integrity**
  - **e.g.** when the factory is closed, all employees should become unemployed
- **Storing Derived Data**
  - **e.g.** the number of items in the trolley should correspond to the current session selection
- **Security Access Control**
  - **e.g.** checking user privileges when accessing sensitive information

Auditing Table Operations

USER_NAME	TABLE_NAME	COLUMN_NAME	INS	UPD	DEL
SCOTT	EMP		1	1	1
SCOTT	EMP	SAL		1	
JONES	EMP		0	0	1

… continuation

MAX_INS	MAX_UPD	MAX_DEL
5	5	5
	5	
5	0	1

Example: Counting Statement Execution

```
SQL>CREATE OR REPLACE TRIGGER audit_emp
  2 AFTER DELETE ON emp
  3 FOR EACH ROW
  4 BEGIN
  5       UPDATE audit_table SET del = del + 1
  6       WHERE user_name = USER
  7       AND table_name = 'EMP';
  7 END;
  8 /
```

Whenever an employee record is deleted from the database, the counter in an audit table registering the number of deleted rows for the current user in system variable USER is incremented.

Example: Tracing Record Value Changes

USER_NAME	TIMESTAMP	ID	OLD_LAST_NAME	NEW_LAST_NAME
EGRAVINA	12-SEP-04	7950	NULL	HUTTON
NGREENBE	10-AUG-04	7844	MAGEE	TURNER

… continuation

OLD_TITLE	NEW_TITLE	OLD_SALARY	NEW_SALARY
NULL	ANALYST	NULL	3500
CLERK	SALESMAN	1100	1100

Example: Recording Changes

```
SQL>CREATE OR REPLACE TRIGGER audit_emp_values
  2 AFTER DELETE OR UPDATE ON emp
  3 FOR EACH ROW
  4 BEGIN
  5   INSERT INTO audit_emp_values (user_name,
  6    timestamp, id, old_last_name, new_last_name,
  7    old_title, new_title, old_salary, new_salary)
  8   VALUES (USER, SYSDATE, :old.empno, :old.ename,
  9    :new.ename, :old.job, :new.job,
 10    :old.sal, :new.sal);
 11 END;
 12 /
```

Whenever some details for an employee are deleted or updated, both the previous and new details are recorded in an audit table to allow tracing the history of changes. An insert operation cannot be recorded with this trigger as old.empno has no value.

Example: Protecting Referential Integrity

```
SQL>CREATE OR REPLACE TRIGGER
   cascade_updates
  2 AFTER UPDATE OF deptno ON dept
  3 FOR EACH ROW
  4 BEGIN
  5   UPDATE emp
  6   SET    emp.deptno = :new.deptno
  7   WHERE  emp.deptno = :old.deptno;
  8 END
  9 /
```

Whenever the department number changes, all employee records for this department will automatically be changed as well, so that the employees will continue to work for the same department.

Restrictions for Database Triggers

- **Problem:** impossible to determine certain values during execution of a sequence of operations belonging to one and the same transaction
- **Mutating tables:** contain rows which change their values after certain operation and which are used again before the current transaction commits
- **Preventing table mutation:**
  - Should not contain rows which are constrained by rows from other changing tables
  - Should not contain rows which are updated and read in one and the same operation
  - Should not contain rows which are updated and read via other operations during the same transaction

Example: Mutating Table

```
SQL> CREATE OR REPLACE TRIGGER emp_count
  2  AFTER DELETE ON emp
  3  FOR EACH ROW
  4  DECLARE
  5       num INTEGER;
  6  BEGIN
  7       SELECT COUNT(*) INTO num FROM emp;
  8       DBMS_OUTPUT.PUT_LINE(' There are now ' ||
          num || ' employees.');
  9  END;
 10  /
```

```
SQL> DELETE FROM emp
  2  WHERE  deptno = 30;

ERROR at line 1:
ORA-04091: table CGMA2.EMP is mutating, trigger/
function may not see it
```

Under the bar is code entered in SQL-PLUS which triggers cascade_updates in this case. Triggers are not executed directly.

Example: Mutating Table (fixed)

```
SQL> CREATE OR REPLACE TRIGGER emp_count
  2  AFTER DELETE ON emp
  3  -- FOR EACH ROW
  4  DECLARE
  5       num INTEGER;
  6  BEGIN
  7       SELECT COUNT(*) INTO num FROM emp;
  8       DBMS_OUTPUT.PUT_LINE(' There are now ' ||
          num || ' employees.');
  9  END;
 10  /
```

Now the trigger becomes a statement trigger and the EMP table is no longer mutating.

```
SQL> DELETE FROM emp  WHERE  deptno = 30;

There are now 8 employees.

6 rows deleted.
```

Rules for Good Practice

- **Rule 1:** Do not *change data* in the primary key, foreign key, or unique key columns of any table
- **Rule 2:** Do not *update records* in the same table you read during the same transaction
- **Rule 3:** Do not *aggregate* over the same table you are updating
- **Rule 4:** Do not *read data* from a table which is updated during the same transaction
- **Rule 5:** Do not use SQL DCL (Data Control Language) statements in triggers

Additional Literature

- **P. Atzeni, S. Ceri, S.Paraboschi and R. Torlone.** *Database Systems*, Chapter 12 “Active Databases”. McGraw-Hill (1999)
- **Oracle Database Server Documentation.** *Oracle9i Database Concepts*, Chapter 17 “Triggers”.
- **Oracle Database Server Documentation.** *Oracle9i Application Developer's Guide – Fundaments*, Chapter 15 “Using Triggers”.