Introduction

• *CONSTRAINTS*: Constraints are general restrictions on the data in the database.

• Three Important Components

  1. *Event*: Execution of *insert*, *delete* and *update* function on a table.

  2. *Condition*: Restrictions on the data in the database defined by the database system or user-procedure.

  3. *Action*: When the *Condition* is satisfied, certain action is taken by the database system or by user-procedure.

• When an *event* occurs, *condition* is checked and if the *condition* is true the *action* is executed.
Built-in Constraints, Assertions and Triggers

• *BUILT_IN CONSTRAINT*: Both the *condition* and the *action* are *system* defined and is created at the time the table is created using *CREATE TABLE* command.

Example:

```sql
CREATE TABLE Student(
    Sid NUMBER (5) CONSTRAINT
    PK_Student PRIMARY KEY,
    Sname VARCHAR2 (30),
    ----
    ----
);
```

• *ASSERTION*: Here the *condition* is *user* defined (not built-in) and the *action* is *system* defined.

Example: A student must take at least two courses:

```sql
CREATE ASSERTION EnrollmentControl CHECK (Not Exists( Select count(*)
    From StudentCourse
    Group by sno
    Having count(cno)>2))
```
Assertion is invoked as a result of insert, delete or update events. CHECK provides the condition and when the condition is true violation occurs and the action is taken by the system which is to prevent the event from taking place.

- TRIGGERS: User defines the condition and the action.

Example: Enter the inserted student record into the table LansingStudents if the inserted student’s address is East Lansing.

CREATE TRIGGER InsertStudent
AFTER INSERT ON Student
FOR EACH ROW
WHEN (NEW.Saddr="East Lansing")
BEGIN
   INSERT INTO
   LansingStudents(Sid, Sname, Saddr, Sdno)
   Values
   (NEW.Sid, NEW.Sname, NEW.Saddr, NEW.Sdno)
END
Various Proposals

- IBM’s Starburst (research prototype), SQL-3 standard, Oracle Triggers, DB2 triggers, Sql-Server.

- Triggers: Implicitly executed as a result of insert, delete and update transactions on a base table. Both condition and action are user defined.

- Statement level invocation, row level invocation.
Potential Applications

- Allow automatic notification of conditions (e.g., Student’s average grade falling below a threshold)
- Automatic maintenance of derived attributes.
- Maintain currency of materialized views.
- Maintaining mirrored (replicated) databases
- Many others
Oracle Triggers

● Basic Structure
  – Event:
    (Triggering Statement)
  – Condition:
    (Trigger Restriction)
  – Action:
    (Trigger Action)

● Example:

Table:
  Inventory(Part_no, parts_on_hand, reorder_point)

CREATE OR REPLACE TRIGGER REORDER
  AFTER UPDATE ON Inventory
  FOR EACH ROW
  WHEN(new.parts_on_hand < new.reorder_point)
  BEGIN
    ORDER PARTS
  END

*Event: UPDATE ON Inventory, Condition: When clause, Action: ORDER PARTS (this can be an SQL statement, a stored procedure, etc.).
12 Possible Combinations for Triggers

- Three Characteristics:
  1. Action (Event): INSERT, DELETE and UPDATE SQL statement
  2. Level: Statement level, Row level
  3. Timing: BEFORE, AFTER

- $3 \times 2 \times 2 = 12$ possible combinations.

- Same table can have a combination of these triggers defined.
Statement Level Versus Row Level

- Two major differences:

  1. How often Trigger fires (invocations):
     - Statement level: Trigger fires once per SQL statement
     - Row level: Trigger fires once for each row affected by the SQL statement.

  2. Visibility
     - Statement level: no accesses to column values currently being updated.
     - Row level: has access to column values currently being updated.
Examples

- Trigger invocations (statement level):
  If total salary exceeds $100000000.00 print a message.
  
  - A statement level trigger:

```
CREATE OR REPLACE TRIGGER MaxTotalSal
AFTER INSERT OR UPDATE ON Employee
BEGIN
  IF 100000000<(select SUM(EmpSal)
       FROM Employee)
  Print "Total Salary Exceeds $100000000.00"
END
```

- Give 10% raise to all employees
  SQL: UPDATE Employee SET EmpSal=1.1 * EmpSal

- Above SQL will fire the trigger MaxTotalSalary only once.
Examples cont’d

- **Trigger invocations (row level):**
  
  A row If new salary $\textgreater$ $1000000.00$ print a message.

  - A row level trigger:
    
    ```sql
    CREATE OR REPLACE TRIGGER MaxSal
    AFTER INSERT OR UPDATE ON Employee
    FOR EACH ROW
    WHEN (NEW.EmpSal > 1000000)
    BEGIN
      PRINT "Employee Salary exceeds $\textdollar1000000.00"
    END
    ```

  - Give 10% raise to all employees
    
    SQL: UPDATE Employee SET
    EmpSal = 1.1 * EmpSal

  - trigger invoked N times if N tuples in Employee affected.
Examples cont’d (visibility)

- Check 50% increase in salary for individual employees

```sql
CREATE OR REPLACE TRIGGER MaxSal
AFTER INSERT OR UPDATE ON Employee
FOR EACH ROW
WHEN (NEW.EmpSal-OLD.EmpSal>1.5*OLD.EmpSal)
BEGIN
  Print "Employee Salary increased by 50%"
END
```

- Detect individual salary increases through NEW.EmpSalary and OLD.EmpSalary

- Statement level trigger does not have access to variables NEW.EmpSalary and OLD.EmpSalary.
Timing

• BEFORE: ACTION is before the event.

• AFTER: ACTION is after the event.

• Both BEFORE and AFTER Row level trigger have access to OLD and NEW

• Check 50% increase in salary could use BEFORE.

• BEFORE has the opportunity to override the new salary.
Column-Name-Clause

Column name clause places tight restrictions on the firing specification. It fires only when the specified columns are affected by the event statements. example:

CREATE OR REPLACE TRIGGER MaxSal
   AFTER UPDATE OF EmpSal ON Employee
   FOR EACH ROW
   WHEN (NEW.EmpSal-OLD.EmpSal>1.5*OLD.EmpSal)
   BEGIN
      Print "Employee Salary increased by 50%"
   END
More Examples

• Maintain a table GoodStudents which has students with grade in a course 4.0. Also indicate the number of rows inserted in each SQL statement.

Trigger1:

```sql
CREATE OR REPLACE TRIGGER GoodStudents
AFTER INSERT OR UPDATE OF Grade ON StudentCourse
FOR EACH ROW
WHEN (NEW.Grade=4.0)
BEGIN
   Insert into GoodStudentsTable
   (NEW.sid, NEW.cno)
   Count=Count+1
END
```

Trigger2:

```sql
CREATE TRIGGER InitializeCount
BEFORE UPDATE ON StudentCourse
BEGIN
   SET Count=0
END
```
Increase the grades of all students in course c1 by .5

UPDATE StudentCourse
SET Grade=Grade+.5
WHERE cno="c1" AND Grade!=4.0

StudentCourse
sid  cno  grade
s1   c1   3.5
s2   c1   4.0
s3   c1   3.5
s4   c1   3.0
s1   c2   3.5
s5   c2   4.0

- Trigger 1 is a row level trigger and will be invoked 2 times.
- Trigger 2 is a statement level trigger and will be invoked only once.
Need a WHEN Clause?

1. WHEN (boolean expression) clause can include any PL/SQL boolean expression.

2. The logic of the boolean test can be moved to the body (action) of the trigger.
Firing Sequence of Multiple Triggers

1. Execute BEFORE statement Trigger

2. LOOP for each ROW affected by SQL statement
   
   (a) Execute BEFORE ROW Trigger
   
   (b) Lock and change ROW
   
   (c) Perform Constraints Checking
   (The lock is not released until the transaction is committed)
   
   (d) Execute AFTER ROW Trigger.

3. Execute AFTER statement trigger.

Note that Constraints have higher run before the triggers.
Firing Within DELETE CASCADE

1. As each parent is deleted, all children related to the parent are deleted. When child is deleted it’s statement and row level triggers are fired.

2. Example:

Delete from Department Where Dno in ('d1', 'd2')

Department BEFORE statement
Department BEFORE ROW
   delete Department tuple 'd1'
Student BEFORE statement
Student BEFORE ROW
   Delete Student tuple 1
Student AFTER ROW
Student BEFORE ROW
   Delete Student tuple 2
   ---
Student AFTER statement
Department BEFORE ROW
   ---
Department AFTER ROW
Department AFTER statement
Data Dictionary Views for Triggers

1. View names: `USER_TRIGGERS, USER_TRIGGERS_COLS`

2. Attributes:
   `USER_TRIGGERS`: `TRIGGER_NAME, TRIGGER_TYPE`
   (before statement, etc.), `TRIGGERING_EVENT`
   (Insert, delete, update), `WHEN_CLAUSE`, etc.

   `USER/TRIGGERS_COLS`: `TRIGGER_OWNER, TRIGGER_NAME, TABLE_NAME, COLUMN_LIST`
   (specified in update), etc.

3. Example: Get all trigger names, trigger types, trigger events, column names for updates for table Student-Course.

   Select `TRIGGER_TYPE, TRIGGERING_EVENT, COLUMN_LIST`
   From `USER_TRIGGERS A, USER_TRIGGERS_COLS B`
   Where `A.TRIGGER_NAME=B.TRIGGER_NAME &`
            `B.TABLE_NAME="StudentCourse"`
Cascading of Triggers

The execution of the action part of a trigger may cause the activation of other triggers. Cascading can be recursive or nested.

1. Recursive trigger:
   When an application updates table T1, which fires trigger TR1 updating table T1. This is a direct recursion. Recursion can be indirect as well where an application updates table T1, which fires trigger TR1 updating table T2. Trigger TR2 defined on table T2 then updates table T1.

2. Nested triggers:
   If a trigger changes a table on which there is another trigger, the second trigger is then activated and can then call a third trigger, and so on.
   Maximum number of cascading is 32 in Oracle; When it exceeds 32, all database changes as a result of original SQL are rolled back.

Example:

CREATE TRIGGER SalaryControl
   AFTER INSERT, DELETE, UPDATE ON Salary OF EMP
WHEN (Select AVG(Salary) From EMP>100)
THEN Update EMP
    Set Salary=.9*Salary
Example: Table EMP
Employee    Salary
    john     90
    David    100

Insert tuple (Rick 200)
How many times the trigger is invoked?
Starburst

1. Research prototype developed at IBM SanJose Research Lab.

2. Based on ECA
   (a) CONDITION: boolean expressed in SQL
   (b) ACTION: SQL statements, rule manipulation statements, transactional instructions such as roll back

3. All are "Statement Level" Triggers invoked implicitly or explicitly:
   Implicitly: Invoked when the transaction issues a COMMIT- Statement level AFTER trigger
   Explicitly: Using PROCESS RULES

4. Partial ordering of rules
   (System maintains a total order to guarantee repeatability).
Rule Processing Algorithm

1. A rule is marked \textit{triggered} by SQL insert, delete, update statements.

2. A set of triggered rules form the conflict set.

3. Rule Processing Algorithm:

   1. Select one rule \( R \) from the conflict set with the highest priority.
   2. Evaluate the \textsc{condition} of \( R \)
   3. If the \textsc{condition} of \( R \) is \textsc{true} execute \textsc{action} of \( R \).

4. Quiescent state: When rule execution terminates and the conflict set is empty.
Transition tables & Visibility

1. Transition tables INSERTED, DELETED and OLD-UPDATED, NEW-UPDATED contains tuples from insert, delete and update SQL statements, respectively.

2. All tuples in the transition table correspond to a transaction.

3. Each tuple affected appears at most in one of the transition tables.

4. Each tuple has a net effect of operations.
   Example: insert tuple T1 will put tuple T1 in the INSERTED table. Following update on the same tuple T1, within the same transaction, to T2 will replace T1 by T2 in the INSERTED table.
Examples

Example 1:
If average salary exceeds 100 then reduce each employee’s salary by 10%.

CREATE RULE SalaryControl ON Emp
WHEN INSERTED, DELETED, UPDATED (Sal)
IF (SELECT AVG(Sal) FROM Emp)>100
THEN UPDATE Emp
    SET Sal=.0*Sal

• Assume the following initial values in the Emp table:

<table>
<thead>
<tr>
<th>Employee</th>
<th>Sal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stefano</td>
<td>90</td>
</tr>
<tr>
<td>Patrick</td>
<td>90</td>
</tr>
<tr>
<td>Michael</td>
<td>110</td>
</tr>
</tbody>
</table>

• Insert tuples (Rick, 150) and (John, 120) through a single INSERT SQL statement into the Emp table.
• Initial tuples in the two tables: table initially as follows:

<table>
<thead>
<tr>
<th>INSERTED</th>
<th>Emp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee</td>
<td>Sal</td>
</tr>
<tr>
<td>Rick</td>
<td>150</td>
</tr>
<tr>
<td>John</td>
<td>120</td>
</tr>
<tr>
<td>Employee</td>
<td>Sal</td>
</tr>
<tr>
<td>Stefano</td>
<td>90</td>
</tr>
<tr>
<td>Patrick</td>
<td>90</td>
</tr>
<tr>
<td>Michael</td>
<td>110</td>
</tr>
<tr>
<td>Rick</td>
<td>150</td>
</tr>
<tr>
<td>John</td>
<td>120</td>
</tr>
</tbody>
</table>

• The two tables will change as follows as a result of *SalaryControl* being triggered twice:

<table>
<thead>
<tr>
<th>INSERTED</th>
<th>INSERTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee</td>
<td>Sal</td>
</tr>
<tr>
<td>Rick</td>
<td>135</td>
</tr>
<tr>
<td>John</td>
<td>108</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Emp</th>
<th>Emp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee</td>
<td>Sal</td>
</tr>
<tr>
<td>Stefano</td>
<td>81</td>
</tr>
<tr>
<td>Patrick</td>
<td>81</td>
</tr>
<tr>
<td>Michael</td>
<td>99</td>
</tr>
<tr>
<td>Rick</td>
<td>135</td>
</tr>
<tr>
<td>John</td>
<td>108</td>
</tr>
</tbody>
</table>
Example 2:
Add the following Trigger \textit{HighPaid} with lower priority than SalaryControl:
Insert any new tuple into a table \textit{HighPaidEmp} if the sal > 100.

\textbf{CREATE RULE HighPaid ON Emp}
\textbf{WHEN INSERTED}
\textbf{IF EXISTS(_SELECT * FROM INSERTED}
\textbf{WHERE Sal>100)}
\textbf{THEN INSERT INTO HighPaidEmp}
\textbf{(SELECT * FROM INSERTED}
\textbf{WHERE Sal>100)}
\textbf{FOLLOWS SalaryControl}

- Triggers SalaryControl and HighPaid will be in the conflict set as a result of the insert of the two tuples (Rick, 150) and (John, 120).

- Trigger SalaryControl will be selected and executed first. It will execute recursively twice.
• At this point the INSERTED table is as follows:

<table>
<thead>
<tr>
<th>Employee</th>
<th>Sal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rick</td>
<td>121</td>
</tr>
<tr>
<td>John</td>
<td>97</td>
</tr>
</tbody>
</table>

• Trigger *HighPaid* is selected next and executed, and the tuples inserted into the HighPaidEmp table are those in INSERTED table.

• Condition clause can be included in the ACTION clause.

• Base table Emp is used for both the condition and the action clauses of example 1.

• INSERTED table is used for both the condition and the action clauses of example 2.

• Which table is used depends on the function of the trigger. However, temporary tables like INSERTED are smaller and more efficient to use.
Applications (using oracle triggers)

1. Derived data Maintenance:
   Computed Attributes: Attribute value is computed on the fly such as the attribute AverageGrade.
   Materialized Views: Physical up-to-date copy of the view is maintained by the system. Challenge is to keep the physical copy updated as soon as the base tables are updated.

2. Database Replication: Mirrored databases (multiple copies of the same database) are maintained at various locations so that copies are physically close to where it is used to improve response time. Challenge is to keep all these copies in sync.

3. Workflow Management: Organize the working activities within an enterprise. Monitor the assignment of tasks and their coordination.
Materialized Views

Maintaining materialized Views:
Done through

- Refresh approach: Recomputing from scratch after each update to base table.

- Incremental approach: Requires computing the positive and the negative deltas, consisting of tuples that should be, respectively, inserted into or deleted from the views.

Example:

Define view GoodStudents(Sname)
    SELECT DISTINCT Sname
    FROM Student, StudentCourse
    WHERE Sno=sno
    GROUP BY sno
    HAVING AVG(Grade)>3.0

Events causing recomputation of the view:
Insert, delete, update(Grade) in StudentCourse.
Refresh:

CREATE TRIGGER RefreshGS1
AFTER INSERT, DELETE, UPDATE(Grade) on StudentCourse
  DELETE * FROM GoodStudent
  INSERT INTO GoodStudent(Sname)
    (SELECT DISTINCT Sname
      FROM Student, StudentCourse
      WHERE Sno=sno
      GROUP BY sno
      HAVING AVG(Grade)>3.0
Incremental (more complex):

CREATE TRIGGER IncrementalGS1
AFTER INSERT, DELETE, UPDATE StudentCourse
FOR EACH ROW
    INSERT INTO GoodStudents (Sname)
    SELECT DISTINCT Sname
    FROM Student
    WHERE new.sno=Sno and 3.0<(Select AVG(Grade)
                        from StudentCourse X
                        where new.sno=X.sno)

CREATE TRIGGER IncrementalGS2
AFTER INSERT, DELETE, UPDATE StudentCourse
FOR EACH ROW
    DELETE FROM GoodStudents X
    WHERE X.Sname=
        (Select Sname
            FROM Student
            WHERE old.sno=Sno and 3.0>(Select AVG(Grade)
                From StudentCourse Y
                where Old.sno=Y.sno)