Active Database Systems

Elisa Bertino
CS Department and CERIAS
Purdue University

Table of contents

- Preliminary aspects
  - Architectural approaches
  - Languages for rule specification
    - Events
    - Conditions
    - Actions
    - Additional features
  - Execution model
    - Rule execution
    - Conflicts resolution
    - Coupling modes
    - Termination
  - Active rules in SQL-99
  - Active rules in Oracle

Preliminary aspects

Passive DBMS vs active DBMS

- Conventional DBMS are passive: they execute operations only upon explicit request
- Often, however, there is the need of reactive capabilities: the DBMS autonomously reacts to some events and executes specified operations
- We refer to active DBMS (ADBMS) to DBMS for which we can specify active rules, also called triggers

Example

Automated management of a store in which if the available quantity of a product goes below 4, 100 items of such product must be ordered

Conventional DBMS:
Preliminary aspects

example (cont.)

- Active DBMS:
  - Active rule A: if quantity <= 4 then orders 100 items

Order 100 items of product x

Sale of 2 items of product x

Preliminary aspects

- The previous example is just one possible application of active rules
- Other examples:
  - integrity constraints
  - alerting
  - auditing
  - security
  - statistics
  - views
- There are 3 different approaches to satisfy applications requiring the automatic execution of actions

Architectural approaches

Passive DBMS: approach 1 (two types of application)

Applications (updates) → Passive DBMS ≫ check Applications periodic polling

+ Monitoring code

Problem: to determine the optimal polling frequency
- Too frequent: inefficiency
- Too seldom: situations requiring a reaction may be “lost”

Architectural approaches

Passive DBMS: approach 2 (only one type of application)

Applications: Code for modifying the database + Monitoring code

Problem:
- Compromises code modularity and re-usability
- If the monitored condition changes, the application has to be changed
- The reaction logic is external to the database
Architectural approaches

Active DBMS (integrated approach)

- Specification of the situations to be monitored
- Queries and updates
- External Events

(re) actions

Active DBMS

- Support for monitoring events
- Integration with other components of the DBMS
- Well-defined semantics
- Efficiency

Active databases

- An active database is a database in which some operations are automatically executed once a given situation arises
- The situation may correspond to the fact that:
  - Some specified events arise, or
  - Specific conditions or state transitions are detected
- An active rule (trigger) is a language construct for defining the system reactions

Specification of active rules

the ECA paradigm

- The most well paradigm for trigger definition is the Event-Condition-Action (ECA)
- Event:
  - If it arises, it activates the trigger
- Condition:
  - If it is satisfied, the trigger action is executed
- Action:
  - It is a sequence of operations that can also modify the database, such sequence is executed only if the condition is satisfied
The most common form of triggers is thus:

**ON event IF condition THEN action**
1. If the event arises, the condition is evaluated
2. If the condition is satisfied, the action is executed

The active rule paradigm originates from the notion of production rules of Artificial Intelligence

Production rules do not typically have events; they have the form (CA):

IF condition THEN action

Why is it useful to have events in rules?
- The condition is expensive (in terms of efficiency) to evaluate, whereas detecting an event is less complex
- Such problem is especially difficult for applications with very large databases
- In addition one can specify different actions for different events and the same condition

What is an event?
"An event is something which happens, which is of interest, and which can be mapped onto some time instant"

**Types of event:**
- **Data modifications:** insertions, deletions, modifications
- **Data accesses:** queries on tables
- **DBMS operations:** login of users, transaction management and authorization
- **Temporal events:** January 12th 2006 at 10 (absolute), each 10 minutes (periodic events)...
- **Application defined events (external event):** room temperature too high

Possibility of defining rules that can be activated before or after an event

Possibility of combining events (composite events):
- **Logical operators:** and, or, etc.
- **Sequence:** a trigger is activated only if two or more events arise according to a specified order
- **Temporal composition:** a trigger is activated when event E2 arises 5 seconds after event E1
### Specification of active rules

**Conditions**

- **What is a condition?**
  "A condition is an additional check that is executed when the trigger is evaluated and **before** the action is executed."

- **Predicates**: WHERE clause of SQL; it is useful to have simple predicates because their evaluation is efficient.

- **Queries**: the condition is true if and only if the query returns the empty set (a possible meaning).

- **Application procedures**: call to a procedure

---

### Specification of active rules

**Actions**

- **What is an action?**
  "An action is a sequence of operations that is executed when its trigger is considered and the trigger condition is true."

- **Types of actions:**
  - **Data modifications**: insertion, deletion, update
  - **Data access**: queries on tables
  - **Other commands**: data definition, transaction actions (commit, rollback), grant and revoke of permissions
  - **Application procedures**: call to a procedure

---

### Specification of active rules

**additional features**

- **Commands for rule management**: allows one to create, modify and drop rules, and to enable and disable rules.

- **Rule priority**: often several rules are activated by the same event; therefore a policy must be devised to determine which rule to process from the set of activated rules; the choice is based on priority:
  - Relative priority (between pairs of rules); more flexible
  - Absolute priority (numeric priority); it requires the priorities to be updated as rules are added
**Execution model – main activities**

- Main activities in an ADBMS:
  1. Detect the events and activate the corresponding rules
  2. Select and execute the activated rule (also called reactive process)

These two activities can be executed concurrently

A possible model (two activities):

**Activity 1**

```plaintext
While true do
  detect events
  activate the rules associated with the detected events
endWhile
```

**Activity 2 (reactive process)**

```plaintext
While there are still active rules Do
  (1) select a rule \( R \) for consideration
  (2) evaluate the condition of \( R \)
  (3) If the condition of \( R \) is true Then 
      execute the action of \( R \) 
      endIf
endWhile
```

1) Non deterministic choice among the rules with the highest priority (the other rules remain activated)
2) The rule is eliminated from the set of activated rules
3) Condition verification and sequential execution of the operations in the rule action

**Execution model – steps**

- The event is detected from the DBMS
- Verify event
- Triggers
- Determine the rule body and its instantiation
- Determine the execution order for rules (conflict resolution)

**Activation**

- Source
- Signaling
- Activated Rules
- Evaluation of rules
- Execution of rules

**Condition evaluation**

- Signal
- Evaluation
- Execution

**Execution model – reactive process**

- **Granularity of the reactive process:** frequency of activation for the reactive process (activity 2)
- **Hierarchy of common granularities:**
  - always, as soon an event arises (temporal events)
  - After each single operation on the database (ex. Insertion of a single tuple)
  - after the execution of an entire DM (ex. After the execution of a SQL INSERT, that is, after the insertion of a set of tuples)
  - At transaction boundaries (start or commit), that is, after the execution of a set of commands
  - activation specified by the application
Execution model – rule execution

- **two modes:**
  - **instance oriented:** the activated rule is executed (action) a time for each element of the database that activates the rule and verifies the condition
  - **set oriented:** the activated rule is executed only once for the set of such elements

- These two modes may result in different states of the relations
- Example: we would like to execute an action each time a tuple is inserted in the Employees relation

---

Example

- relation Employees
- rule R on Employees:
  - event = insert tuples in Employees
  - condition = the inserted tuples are not already in the table (thus it is always verified)
  - action = replace the value of the Salary attribute of the inserted tuples with the average value + 5 of the Salary attribute computed on all the tuples of the Employees relation

- **Set oriented execution (R is executed 1 time for ALL tuples):** all inserted employees will have the same value for the Salary attribute
- **Instance oriented execution (R is executed 1 time for EACH tuple):** all inserted employees will have different values for the Salary attribute

---

Execution model – resolution of conflicts

- The difference between the two modes depends also from the granularity of the reactive process
  - ex. If the granularity of reactive process is “always” (after each single operation in each single statement)
  - the rule execution “becomes” instance oriented (that is, there is no difference between the two modes)

---

Step (1) of the reactive process considers a single rule at the time

- Several rules can be activated at the same time, because:
  - a single event activates several rules
  - the granularity of the reactive process is coarse
    - Several events activating rules arise before the reactive process is executed
    - the rules that are activated and are not selected at step (1) of the reactive process remain activated

- A rule must therefore be selected among the several rules activated
Execution model – resolution of conflicts

- How to choose a rule from a set of activated rules?
  - arbitrarily
  - based on priority
    - absolute
    - relative
  - statistical properties (e.g., when the rule has been created)
  - dynamic properties (e.g., the rule most recently activated)
  - Alternative to the execution of a single rule: evaluate the conditions of several rules among the ones that are activated and eventually concurrently executed the actions of these rules.

Execution model – coupling modes

- What are?
  - Criteria establishing the relationships between the transaction that generates the event and the rule processing
- They establish the relationships between:
  - event and condition (EC)
  - condition and action (AC)

Possible coupling modes (apply to both EC and AC):
- Immediate: immediately when the event is generated in the same transaction
- Deferred: when the transaction commits
  - Useful for integrity constraints
  - During the execution, a transaction could violate a constraint but before the commit it could restore a consistent state
- Separated: in a new transaction; such transaction can be dependent/independent from the commit of the original transaction

Execution model – the termination problem

- The reactive process may not terminate (infinite activations of rules)
- Possible solutions:
  - Leave to the DB designer the task of designing the rules in such a way that their execution always terminates
  - Limit the maximum number if rules that can activated
  - Impose syntactic restrictions on rules:
    - The rules cannot activate each other
    - The rules can activate each other as long as there are no cycles
    - Rules may have cycles but it is guaranteed that the condition of some rules inside of the cycles eventually becomes false
Execution model – Transition tables

- The are tables that allows one to refer to the set of tuples that have been actually
  - inserted
  - deleted
  - modified

- In the case of "modified tuples" such tables are two: one records the values before the update, whereas the other records the values after the update

- Such tables can be used in the evaluation of the condition of a rule and/or in the execution of the actions of a rule

- They improve efficiency, by limiting the evaluation of the rule conditions to the tuples in the transition tables

Active Rules in SQL-99

SQL-99 – syntax of commands

- Creation of an active rule:
  ```sql
  CREATE TRIGGER Name
  [BEFORE | AFTER] Event ON Relation
  [REFERENCING { OLD [ROW] [AS] Variable | NEW [ROW] [AS] Variable | OLD TABLE [AS] Variable | NEW TABLE [AS] Variable}]
  [FOR EACH {ROW | STATEMENT}]
  [WHEN Condition]
  SQL Commands
  ```

- Deletion of an active rule:
  ```sql
  DROP TRIGGER Name
  ```

SQL-99 - triggers

- A trigger definition contains the following information:
  - Name of the trigger (unique)
  - Name of the table (trigger table, only one) on which the events arise that activate the trigger
  - Execution mode for the trigger body (BEFORE/AFTER) with respect to the event that activates the trigger
  - Event (only one) that, when arises on the trigger table, activates the trigger
  - Alias defined for referencing the transition tables (REFERENCING clause)
  - Action granularity (how many and on what the condition is evaluated and the action executed, FOR EACH ROW/STATEMENT clause)
  - Creation timestamp of the trigger (for the priority)
**SQL-99 - event**

- **Events:**
  - possible events: INSERT, DELETE, UPDATE, UPDATE OF attribute list
  - If we specify UPDATE OF a1,…,an, the rule is activated only by an event that modifies all and only all attributes a1,…,an
  - Only a single event can activate a rule; therefore only a single operation on the table can activate a trigger
  - It is possible to specify that the trigger action be executed before or after the event that has activated the trigger
    - trigger before: the action is executed immediately before the execution of the operation associated with the event
    - trigger after: the action is executed after the execution of the operation associated with the event

**SQL-99 – condition and action**

- **Condition:**
  - Arbitrary SQL predicate (WHERE clause)
  - It is not verified if the evaluation returns FALSE or UNKNOWN

- **Action:**
  - a single SQL statement
  - a sequence of statements
    - BEGIN ATOMIC
    - SQL statement 1, SQL statement 2,…
    - END
  - The condition and action can be executed
    - FOR EACH ROW (for tuple involved in the event)
    - FOR EACH STATEMENT (only once for the command that has activated the trigger)
    - It is executed also if the command that activates the trigger has not actually modified any tuple

**SQL-99 - action**

- **Possible actions**
  - Trigger before: data definition, data selection, procedure calls; it is however not possible to execute actions that modify the database state
  - Trigger after: everything that can be executed in the before trigger + data modification operations (INSERT, DELETE, UPDATE)

**SQL-99 - types of trigger**

<table>
<thead>
<tr>
<th>Type</th>
<th>STMT/ROW</th>
<th>BEFORE</th>
<th>AFTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trigger</td>
<td></td>
<td>Trigger before statement: The trigger is executed only once before the execution of the operation that activates it</td>
<td>Trigger after statement: The trigger is executed after each tuple has been modified that is involved in the execution of the operation that activates it</td>
</tr>
<tr>
<td>before</td>
<td></td>
<td>Trigger before statement: The trigger is executed only once before the execution of the operation that activates it</td>
<td>Trigger after statement: The trigger is executed after each tuple has been modified that is involved in the execution of the operation that activates it</td>
</tr>
<tr>
<td>Trigger</td>
<td></td>
<td>Trigger before statement: The trigger is executed before modifying each tuple involved in the execution of the operation that activates it</td>
<td>Trigger after statement: The trigger is executed after each tuple has been modified that is involved in the execution of the operation that activates it</td>
</tr>
</tbody>
</table>
SQL-99 - types of trigger

- Row level vs statement level:
  - It is better to use a row level trigger if the action of the trigger depends on the value of the modified tuple.
  - It is better to use a statement level trigger if the action of the trigger is global for all the modified tuples (execute a complex authorization check, generate a single audit record, compute some aggregate functions).

- Before vs after:
  - It is better to use a before trigger if the trigger action determines whether the command will be actually executed (in this way, we avoid executing a command and executing its rollback) or to derive values of columns to use in an INSERT or an UPDATE.

SQL-99 - REFERENCING clause

- The REFERENCING clause "implements" the transition tables.

- It is necessary to specify the alias if the condition and/or action refer the table on which the trigger is defined.

SQL-99 - REFERENCING clause

- **Questions**: which are the tuples that are visible during the evaluation of the condition and the execution of the action?
- **Reply**: it depends from three factors:
  - from the event that has activated the trigger
  - from the type of trigger (**before/after**)
  - from the type of execution (**row/statement**)

SQL-99 - REFERENCING clause

- **Type of event**:
  - **INSERT**: the inserted tuples can be accessed by using the REFERENCING NEW clause (at tuple or table level).
  - **DELETE**: the deleted tuples can be accessed by using the REFERENCING OLD clause (at tuple or table level).
  - **UPDATE**: the previous and current values of tuples can be accessed by using the REFERENCING OLD and NEW clause (at tuple or table level).
**SQL-99 - REFERENCING clause**

### Type of trigger
- **Before** (before the execution of the command):
  - It is not possible to use REFERENCING OLD TABLE and REFERENCING NEW TABLE (only references to ROW)
  - INSERT: any reference to the table (on which the insert is being executed) does not contain the new tuples
  - DELETE: any reference to the table (on which the delete is being executed) contains the deleted tuples
  - UPDATE: the table contains tuples as they were before the update
- **After** (after the execution of the command):
  - It is not possible to use all the clauses
  - INSERT: any reference to the table (on which the insert is being executed) contains the new tuples
  - DELETE: any reference to the table (on which the delete is being executed) does not contain the deleted tuples
  - UPDATE: the table contains the modified tuples

### Type of execution:
- **FOR EACH ROW**
  - In the REFERENCING clause one can reference both a table or a tuple (OLD/NEW-ROW/TABLE)
- **FOR EACH STATEMENT**
  - In the REFERENCING clause one can reference only a table (only OLD/NEW-TABLE)

---

**SQL-99 – Execution modes**

- **Granularity at level of a single SQL statement**
- **Two execution modes:**
  - FOR EACH ROW
  - FOR EACH STATEMENT (default)
- **Coupling mode:**
  - EC immediate
  - CA immediate
- **Selection criteria for a rule from the set of activated rules**
  - It depends from the type of trigger (before/after) and the priority
  - In SQL-99 the priority is assigned based on the creation order: an "old" trigger is executed before a "young" trigger
- **Recursive execution model:** if the execution of a trigger activates another trigger, then
  - Old values: the initial values
  - New values updated during the computation
SQL-99 – Execution modes

- Problem: how do triggers “interfere” with semantic integrity constraint checking?
- Example:

  ```sql
  CREATE TRIGGER Trigger1 AFTER UPDATE ON Table1 ...
  CREATE TRIGGER Trigger2 BEFORE UPDATE ON Table1 ...
  CREATE TRIGGER Trigger3 AFTER UPDATE ON Table1 ...
  ALTER TABLE Table1 ADD CONSTRAINT Constraint1 ...
  (assume that this constraint is an immediate constraint)
  ```

SQL-99 - example

- What happens when we execute an UPDATE on Table1?
- The following steps are executed (according to this order):
  - Trigger2 is activated
  - Operation UPDATE is executed on Table1
  - Constraint1 is executed (the constraint is checked at the end of the command execution)
  - Trigger1 is activated (it has been defined first)
  - Trigger3 is activated (it is “younger” than Trigger1)

SQL-99 - Execution modes

- Steps for determining the execution order:
  1) Selection based on the trigger types: different types of trigger are executed according to the following order:
     - trigger BEFORE STATEMENT
     - for each tuple involved in the command
       - trigger BEFORE ROW
       - execution of the command and checking of integrity constraints
       - trigger AFTER ROW
     - checking of the constraints that require the execution of the command to be completed
     - trigger AFTER STATEMENT
  2) If there are several triggers of the same type, the order is given by the creation order

SQL-99 - termination

- the standard is not clear on this respect
- it assumes that the system keeps track of the various activations, through an activation graph
  - nodes: tables, modifications on tables
  - edges: events, actions
- the graph is built when triggers are created
- If one try to specify a trigger that may generate non terminating executions, the creation is not allowed
**SQL-99 - Summary**

<table>
<thead>
<tr>
<th>Data model</th>
<th>Relational, object-relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive events</td>
<td>Operations on the database</td>
</tr>
<tr>
<td>Composite events</td>
<td>No</td>
</tr>
<tr>
<td>Parameter passing</td>
<td>No</td>
</tr>
<tr>
<td>Transition tables</td>
<td>Yes (tuple, table)</td>
</tr>
<tr>
<td>Net effect</td>
<td>No</td>
</tr>
<tr>
<td>Coupling modes</td>
<td>Immediate</td>
</tr>
<tr>
<td>Termination</td>
<td>Syntactic check</td>
</tr>
<tr>
<td>Rule ordering</td>
<td>Type + creation order</td>
</tr>
</tbody>
</table>

**SQL-99 - Triggers and constraints**

- The triggers are more flexible than integrity constraints; triggers can use several types of action in their bodies in order to react to a violation
- The flexibility is not always an advantage
- Sometimes defining the integrity constraints has several advantages:
  - better optimization
  - reduced number of programming errors
  - the constraints are part of the standard since a long time; the triggers

**SQL-99 - Example 1**

- We want to keep track in the table Deleted_Emps of the employees removed from the Employees table

```sql
CREATE TRIGGER Delete_Emp
AFTER DELETE ON Employees  // AFTER because it deals with a modification to the table
REFERENCING OLD ROW AS Old //alias to be used in order to reference to a deleted tuple
FOR EACH ROW
INSERT INTO Deleted_Emps
VALUES (Old.Emp#); //the value to be inserted is the value of the Emp# attribute of the tuple which has been deleted
```

**SQL-99 - Example 2**

- Suppose that the Employees table has the following schema:
  Employees(Emp#, Salary, Dept#, Home_Ph, Office_Ph)

  - It is not possible to support such requirement by using the DEFAULT clause (of the Create table command) because DEFAULT ColumnName is not a correct specification
SQL-99 - Example 2

CREATE TRIGGER Default_Home_Phone
AFTER INSERT ON Employees
REFERENCING NEW ROW AS New
FOR EACH ROW
SET New.Home_Phone =
  homeORoffFun(New.Home_Phone, New.Office_Phone);

where: homeORoffFun(value1,values2) is a function such that
  CASE WHEN value1 IS NOT NULL THEN value1
  ELSE value2

SQL-99 - Example 3

Assume that the table Departments has a Budget attribute and that the budget of a department cannot be modified after 5 pm.

CREATE TRIGGER Update_Departments
AFTER UPDATE OF Budget ON Departments
REFERENCING NEW AS New
WHEN (CURRENT_TIME>TIME '17:00:00:00')
SELECT MAX(Budget)/0 FROM New;

N.B. the default is FOR EACH STATEMENT

The action of the previous trigger generates an error; therefore because the execution mode is immediate, the action and the event that has activated the rule are rollbacked.

therefore:
- An update to the table Departments activates the rule
- after 17, the condition is true
- The action fails
- The update is rollbacked

(very) specific cases:
- Event: UPDATE Departments SET budget = v1, deptname = v2
  - The rule is not activated (the event is different)
- Event: UPDATE Departments SET budget = NULL;
  - If the condition is true, a division NULL/0 has to be computed (which is legal?)
  - Therefore the action does not fail and the update cannot be aborted
Consider the following tables:

- **Prime Ministers** (Name, Popularity, ...)
- **Tax Payers** (Name_Tax_Payer, Tax_Amount, ...)
- **National Debts** (...Amount, ...)

Suppose that the first time Bob is elected, the taxes are decreased of 1%; in addition each modification to the taxes influences the national debt and reduces the popularity of Bob.

```sql
CREATE TRIGGER Update_Prime_Ministers
AFTER UPDATE OF Name ON Prime_Ministers
REFERENCING OLD ROW AS Old, NEW ROW AS New
FOR EACH ROW
WHEN (New.Name='Bob' AND New.Name<>Old.Name)
UPDATE Tax_Payers
SET Tax_Amount=Tax_Amount * 0.99;
```

```sql
CREATE TRIGGER Update_Tax_Payers
AFTER UPDATE OF Tax_Amount ON Tax_Payers
REFERENCING OLD ROW AS Old, NEW ROW AS New
FOR EACH ROW
BEGIN ATOMIC
UPDATE National_Debts
SET Amount = Amount+(Old.Tax_Amount-New.Tax_Amount);
UPDATE Prime_Ministers
SET Popularity = Popularity – 0.01
END;
```

```sql
CREATE TRIGGER Update_Tax_Payers
AFTER UPDATE Tax_Amount ON Tax_Payers
REFERENCING OLD AS Old, NEW AS New
FOR EACH STATEMENT
BEGIN ATOMIC
UPDATE National_Debts
SET Amount = Amount+(SELECT SUM(Tax_Amount) from Old)  -
(SELECT SUM(Tax_Amount) from New)
UPDATE Prime_Ministers
SET Popularity = Popularity – 0.01* (SELECT COUNT(*) FROM New)
END;
```

Problem: the triggers seem to activate each other resulting in an infinite reactive process.

The cycle is only apparent because the updates on Prime Ministers are on different columns.
Active Rules in Oracle

Triggers in Oracle

CREATE [OR REPLACE] TRIGGER Name
{BEFORE | AFTER | INSTEAD OF}
[ ( delete | insert | update [of [Column [,]* ] | OR* ]
ON Relation
[] REFERENCING [OLD AS] Variable |
NEW [AS] Variable [,]*
FOR EACH ROW
[WHEN (Condition) ] ]
{PL/SQL Block | Procedure class}

- Additional commands: ALTER TRIGGER with options ENABLE and DISABLE, DROP TRIGGER

Oracle - Event

- Events
  - Commands INSERT, DELETE, UPDATE, UPDATE OF attribute list, on table and views
  - Commands CREATE, ALTER, DROP on a schema object
  - Startup or shutdown of the database
  - Specific or generic error
  - Connection/disconnection of a user
  - It is possible to specify more than one event (OR semantics)
  - Triggers activated before or after or instead of an event

- We consider only triggers activated by database commands

Oracle - Action

- Action
  - It can be PL/SQL block or a procedure call (no DDL nor transactional commands, ex. ROLLBACK)
  - If the trigger includes several events, in the action it is possible to execute different activities through the use of conditional predicates
    - IF inserting
    - IF deleting
    - IF updating, IF updating('column_name')
### Oracle - Condition

- **Condition:**
  - SQL predicates (WHERE clause) without subqueries and user-defined functions
  - The condition can only be specified for row trigger (FOR EACH ROW) and can only involve the attributes of the modified tuple
  - For the statement trigger one can perform checks in the PL/SQL block

### Oracle – Trigger types

- 4 types (already present in SQL-99)
  - Only for triggers created on tables
- **trigger INSTEAD OF**
  - Only for triggers created on views
  - The body of the trigger is executed instead of the command that has activated the trigger
  - They are always of type ROW
  - They are useful for executing updates on views that cannot be executed directly by the DML commands (INSERT, UPDATE, DELETE)

### Oracle - Example

- Consider a view that is defined by an aggregate function
- By using the standard procedures of the DBMS it is not possible to execute DELETE on the view
- **Solution**
  - A trigger is defined of type INSTEAD OF with event DELETE on View_Name
  - The trigger action will modify the table on which the view is defined according the chosen semantics
  - When a DELETE command is issued for the view, the trigger is executed INSTEAD OF the DELETE command

### Oracle – REFERENCING clause

- It can only be specified for row triggers
- By default, in the action block the old row is `:old` and the new row is `:new`; in the condition the old row is denoted by old and new row is denoted by new
- The visibility rules are similar to the ones of SQL-99
Oracle - Restriction

- A table is mutating if it is the table on which the statement (INSERT, DELETE, UPDATE) is executed that activates the trigger.
- The triggers of type row cannot access with INSERT nor modify with INSERT, DELETE, UPDATE the mutating tables.
- This is a strong restriction.
- Motivation: to avoid that a trigger manipulates data that could be inconsistent and behaviors that may depend from the order according to which the tuples are modified during the execution of the command.

Oracle – Execution modes

- Granularity at the statement level.
- Two modes:
  - FOR EACH ROW
  - FOR EACH STATEMENT
- Coupling mode:
  - EC immediate
  - CA immediate
- Recursive execution

Oracle – Execution modes

- Choice of the rule:
  - It depends from the trigger type as in SQL-99
    - trigger BEFORE STATEMENT
      - For each tuple involved in the command
        - trigger BEFORE ROW
          - Command execution and integrity constraint checking
        - trigger AFTER ROW
          - Checking of constraints that require the execution of the command to be executed
      - trigger AFTER STATEMENT
        - If there are several triggers of the same type, the choice is not deterministic.
- Note that because the INSTEAD OF triggers are always distinct from the events that activate other types of trigger, they never need to be ordered with respect to triggers of other types.

Oracle - termination

- by timeout
- Default:
  - 32 recursive calls of rules
- The maximum number of calls can be modified
Oracle – summary

<table>
<thead>
<tr>
<th>Data model</th>
<th>Relational, object-relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive events</td>
<td>Operations on the database</td>
</tr>
<tr>
<td>Composite events</td>
<td>Yes</td>
</tr>
<tr>
<td>Parameter passing</td>
<td>No</td>
</tr>
<tr>
<td>Transition Tables</td>
<td>Yes (tuple)</td>
</tr>
<tr>
<td>Net effect</td>
<td>No</td>
</tr>
<tr>
<td>Coupling mode</td>
<td>Immediate</td>
</tr>
<tr>
<td>Termination</td>
<td>Based on timeout</td>
</tr>
<tr>
<td>Rule ordering</td>
<td>Type + non-deterministic rules</td>
</tr>
</tbody>
</table>

Oracle – additional features

- If an error arises in the action of a trigger, the execution of the trigger is aborted, unless the error is handled by an exception in the action of the trigger.
- How to overcome the limitations concerning the mutating tables:
  - Auxiliary tables can be used.
  - If a procedure is used:
    - The error is not detected at compile time but at run-time.

Oracle - example 1

We want to check that the salary on each employee is in the allowed range for the job of the employee.

```sql
CREATE TRIGGER Verify_Salary
BEFORE INSERT OR UPDATE OF Salary, Job ON Employees
FOR EACH ROW
WHEN (new.Job = 'president')
DECLARE
  minsal number;
  maxsal number;
BEGIN
  SELECT minsal, maxsal INTO :minsal,:maxsal
  FROM Salaries
  WHERE Job = :new.Job;
  THEN raise_application_error(-20601,'salary outside the range for employee' || :new.Name);
  END IF;
END;
```

Oracle - example 1

- Raise_application_error is a procedure that returns to the calling program (in such case the transaction that has activated the trigger) a code and an error message.
- The calling program can identify the error (functions SQLCODE and SQLERRM) and activate the proper procedures (for example ROLLBACK).
- It is useful because ROLLBACK cannot appear in the action.
Oracle - example 2

the same trigger as the previous one, with a call to a procedure VerifySalary, the body of which corresponds to the action of the previous trigger

CREATE TRIGGER Verify_Salary
BEFORE INSERT OR UPDATE OF Salary, Job ON Employees
FOR EACH ROW
WHEN (new.Salary <> 'president')
CALL VerifySalary(:new.Job, :new.Salary, :new.Name);

Oracle - example 3

Order some products when their availability decreases below a given threshold
CREATE TRIGGER OrderProducts
AFTER UPDATE OF Availability ON Inventory
FOR EACH ROW
WHEN (new.Availability < new.MinQty)
DECLARE
x number;
BEGIN
SELECT COUNT(*) INTO x FROM PendingOrders
WHERE ProdCode = :new.ProdCode;
IF (x = 0) THEN INSERT INTO PendingOrders
VALUES (:new.ProdCode, :new.OrderQty, SYSDATE);
END IF;

Oracle - example 4

Maintain a derived column (TotalSalary) that keeps track of the sum of the salaries of the employees in each department

CREATE TRIGGER Total_Salary
AFTER DELETE OR INSERT OR UPDATE OF Dept#, Salary ON Employees
FOR EACH ROW
BEGIN /* assume that Dept# and Salary are NOT NULL columns*/
IF DELETING OR (UPDATING AND :old.Dept# != :new.Dept#)
THEN UPDATE Departments SET TotalSalary = TotalSalary - :old.Salary
WHERE Dept# = :old.Dept#;
END IF;
IF INSERTING OR (UPDATING AND :old.Dept# != :new.Dept#)
THEN UPDATE Dept SET TotalSalary = TotalSalary + :new.Salary
WHERE Dept# = :new.Dept#;
END IF;
END;
Let Reservations and TravelAgencies two tables related by the attribute AgencyName which is a foreign key in Reservations.

Suppose to define the following triggers:
- trigger t1 that upon the first reservation creates a tuple related to the travel agency making the reservation; for the subsequent reservations the trigger simply updates the total expenses and number of reservations.
- trigger t1 is then extended to check that travel agency has at most three reservations; in such a case, an exception is raised.

```sql
create or replace trigger t1
before insert on Reservations
for each row

declare
conta number;

begin
select count(*) into conta guardiamo se agenzia esiste
from TravelAgencies
where AgencyName = :newAgencyName;
if (conta = 0) agenzia non esiste
then insert into TravelAgencies
values (:newAgencyName,1,:newExpense);
else begin
select reservationNumber into reserveN
from TravelAgencies
where AgencyName = :newAgencyName;
if (reserveN = 3)
then raise application error(-20601,’too many reservations’);
set numReservations = numReservations + 1,
ExpenseTotal = ExpenseTotal + :newExpense
where AgencyName = :newAgencyName;
end;
end;
```