Database Systems

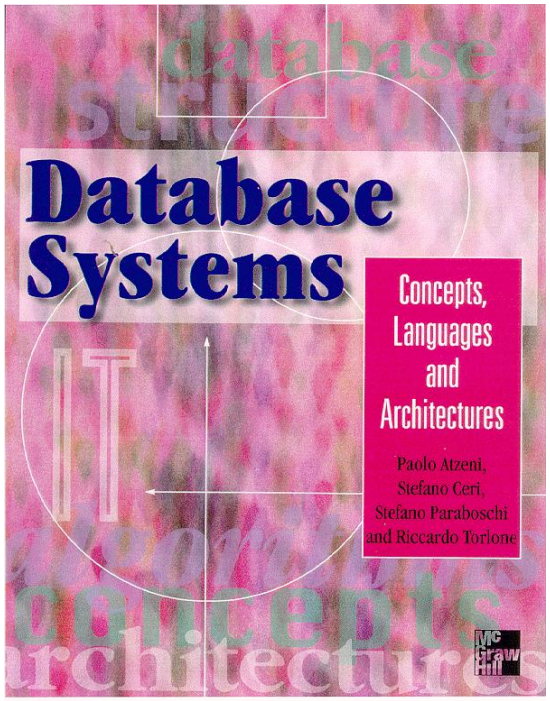
Lecturer: Antonio Pellicani
Room: Box 1, KDDE Lab, 4° Floor, DIB
antonio.pellicani@uniba.it

Slides available at:
https://elearning.uniba.it/course/view.php?id=12972

Course password: ***csdb2526***
Slide password: ***bdii1213***

Database Systems

Book:
Database Systems - Concepts, Languages and Architectures
Paolo Atzeni, Stefano Ceri, Stefano Paraboschi and Riccardo Torlone



https://dbbook.inf.uniroma3.it/

Active Databases

A database is said ***active*** if some operations are performed automatically when a particular *situation* occurs.

The *situation* may correspond to the fact that:

- a specific event has occurred, or
- specific conditions have been detected or state transitions occurred.

The definition of the situations, in which operations must be performed automatically, is done by means of appropriate language constructs, said active rules or triggers.

ECA Paradigm

The rules follow the *Event-Condition-Action* (ECA) paradigm:

- event: each rule reacts to some events (normal situations, changes in the database);
- condition: to be verified to determine whether the action should be executed;
- action: sequence of operations that can modify the database.

A trigger is

- ***activated*** by one of its events,
- ***considered*** during the evaluation of its condition
- ***executed*** if the evaluation is positive.

Active rules vs. production rules

The paradigm is similar to that of production rules studied in Artificial Intelligence.

The difference is that production rules typically do not have events: they have the condition-action form:

IF condition THEN action

The reasons for the presence of events in active rules are:

- The condition is computationally expensive to evaluate, whereas it is easier to detect an event, particularly in large databases.
- It is possible to specify different actions for different events and the same condition.

Reactive Behavior

In active databases, the execution of active rules takes place under the control of an integrated subsystem which allows to manage the active rules, to keep track of events and actually execute the rules.

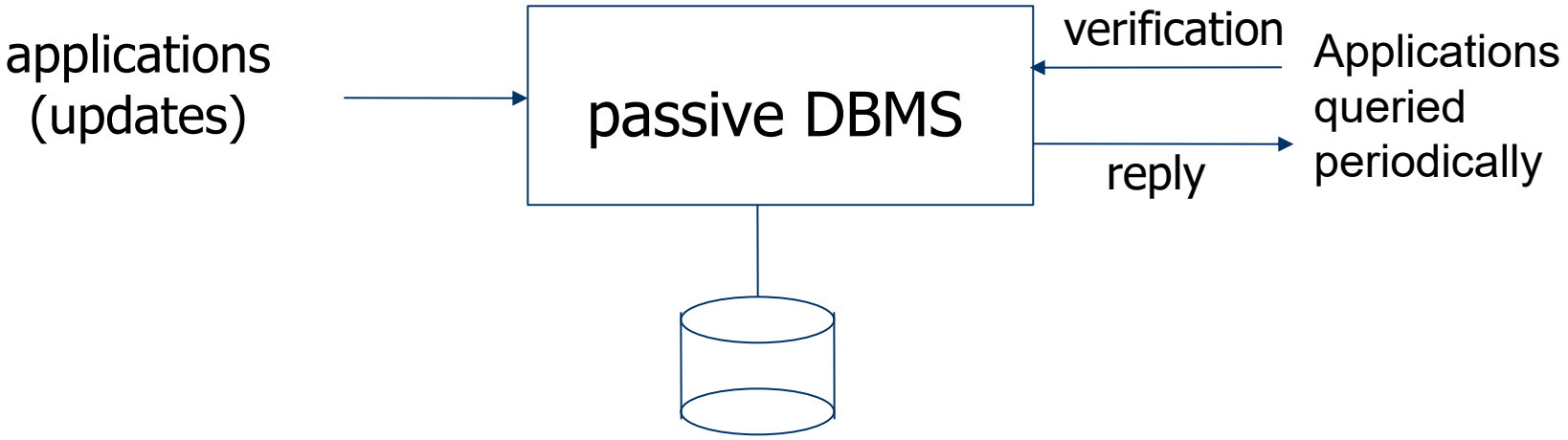
An alternate of executions takes place:

- the transaction (run by users)
- the rules (run by the system)

This alternation manifests a reactive behavior of the database, which is opposed to the typical passive behavior.

Passive Behavior

With two types of applications:

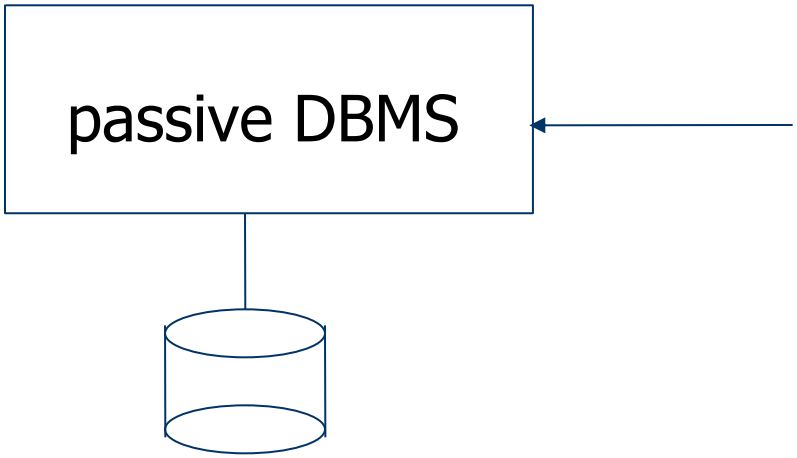


Problem: Determine the optimal query frequency (polling)

- **Too frequent ➔ inefficient**
- **Too rare ➔ reactions could be lost**

Passive Behavior

Only one application type:

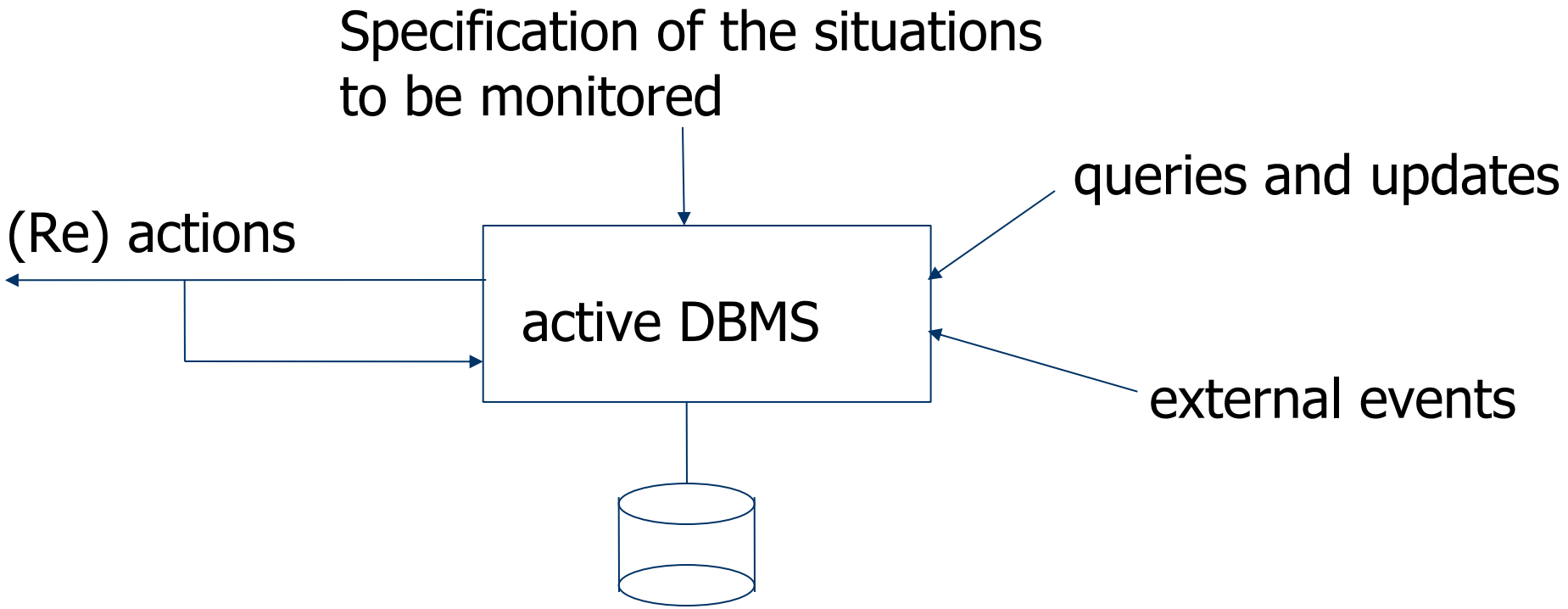


Applications:
Code to modify the database
+
Monitoring Code

Problem: It compromises the invariance to the code changes

- **Monitoring Conditions Change ➔ the application must be changed**

Active behavior



The knowledge concerning the reactive behavior is subtracted to applications and coded in the form of active rules.

Benefit: The knowledge related to the reactive behavior is part of the schema and it is shared by all applications.

Trigger Definition

The triggers are defined using DDL (Data Definition Language).

For each trigger are specified:

- The ***events***, which activate the trigger (e.g., the execution of the insert, delete, update primitives).
- The ***condition*** (Optional), which is a predicate expressed in SQL.
- The ***action***, which is a sequence of generic SQL primitives, sometimes enriched by an integrated programming language available for a specific product (for example, PL / SQL in Oracle).

Active databases differ significantly in the definition of the activation, consideration and execution of the rules.

Events

- ***What is an event?***

"An event is something that happens, that is of interest, and can be put in correspondence with a time instant "

- ***Types of events:***
  - **Data Changes**: Insertions, deletions, updates
  - **Access to data**: Queries on tables
  - **Operations on the DBMS**: User logins, transactions and authorizations management
  - **Temporal Events**: March 16, 1978 9:10 am (absolute), every 10 minutes (periodic events), in 1 hour (relative)
  - **Events defined by the applications (external events)**: too high temperature of the room

Conditions

- ***What is a condition?***

"A condition is an additional control which is carried out when evaluating a trigger and, anyway, before the trigger action is executed "

- ***Types of conditions:***
  - **Predicates**: Like in the WHERE clause of the 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 a non-empty set
  - **Application procedures**: Call to a procedure

  If the condition is not present, it is assumed to be always true.

Actions

- ***What is an action?***

"An action is a sequence of operations which is performed when the trigger is considered and the condition is true "

- ***Types of actions:***
  - **Data Changes**: Insert, delete, update
  - **Data access**: Queries on tables
  - **Other commands**: Data definition, transaction actions (commit and rollback), assignment (grant) and dismissal (revoke) permissions
  - **application procedures**: Procedure calls

Trigger Activation: granularity

It can take place at the level of

- ***tuple*** (*row-level*): i.e. for each tuple involved in the operation. The trigger behavior is oriented to the instance.
- ***primitive*** or ***command*** *(Statement-level)*: the activation occurs only once for each SQL primitive and refers to all the tuples involved in the execution of the primitive. The trigger behavior is oriented to the set of instances.

Example

Employees

ID	Name	Salary
1	Luca	1000
2	Gianni	2000
3	Vincenzo	1500

Query

```
UPDATE Employees SET
   salary = salary + 100
```

Trigger

Event:
UPDATE on Employees
Condition …
Action …

How many times the trigger is activated?

Example

Employees

ID	Name	Salary
1	Luca	1000
2	Gianni	2000
3	Vincenzo	1500

New query

```
UPDATE Employees SET
   salary = salary + 100
WHERE salary > 1800
```

Trigger

Event:
UPDATE on Employees
Condition …
Action …

How many times the trigger is activated?

Cascade activation

Triggers can activate one another (in *cascade*); this happens when the action of a trigger is the event of another trigger.

The activation can also be carried out indefinitely, generating situations of *no termination*.

We will see later how to control non-termination situations.

Consideration of the triggers: execution modes

As mode of execution we intend the link between activation (of an event) and consideration/execution (condition and action).

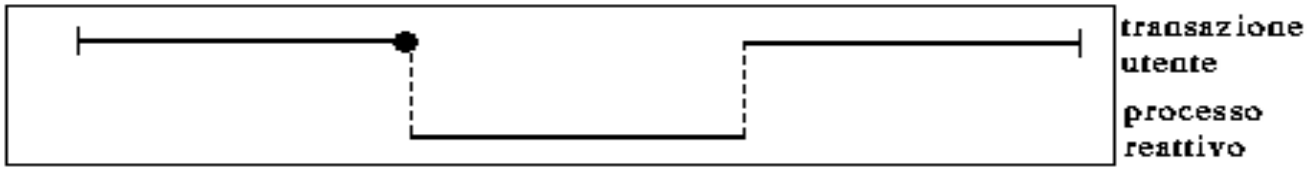
The condition and the action are always evaluated together.

The mode of execution can take different forms:

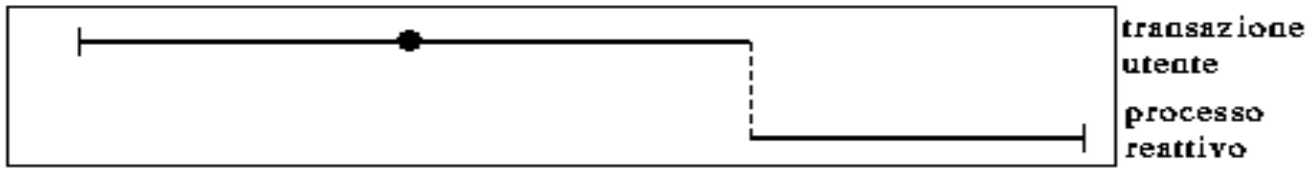
- ***Immediate*** Consideration and execution with the event, immediately after the event that triggered them (option *after*) or, more rarely, before (option *before*).
- ***Deferred***: The trigger is considered at the end of the transaction, as a result of a command Commit-work.
  - *Example*: Triggers that handle integrity constraints that may be violated during a transaction
- ***Decoupled*** (or ***separate***): The trigger is evaluated in another transaction
  - *Example*: Management of a variation of the value of shares

Consideration of the triggers: implementing rules

(a) modalita' immediata



(b) modalita' differita



(c) modalita' separata

Triggers in SQL3

The triggers are present in all the most sophisticated DBMSs, and standards have been provided for SQL: 1999 (known as SQL3) and SQL: 2003.

There syntax for creating a trigger in SQL is as follows:

CREATE TRIGGER *Name*
{ **BEFORE** | **AFTER** } *Event* **ON** *Table*
[**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

Triggers in SQL3

A trigger definition contains the following information:

- **Name** of the trigger (unique)
- **Name of the table (Trigger table,** only one**)** to which events of the trigger refer to
- **Execution Mode** of the trigger body (BEFORE / AFTER) with respect to the event that activates the trigger
- **Event (only one)** that, when concerns the trigger table, activates the trigger
- **Alias** defined for referencing transition tables containing tuples that have been modified, or inserted or deleted (REFERENCING clause)
- **Granularity of the action** (Clause FOR EACH ROW / STATEMENT)
- **Timestamp of the creation of the trigger** (In order to define a priority w.r.t. other triggers created afterwards).

Triggers in SQL3

- **Events**:
  - Possible events: INSERT, DELETE, UPDATE, UPDATE OF list of attributes
  - If we specify UPDATE OF $A_1,\ldots,A_n$ the rule is only activated when the change event involves all the attributes $A_1,\ldots, A_n$
  - **Only one event** can activate a trigger; Therefore, only a single operation on table can activate a trigger.
  - We can specify that **the action of the trigger** is performed immediately before or after the event that triggered the trigger. Therefore, we have two types of triggers:
    - **before trigger**
    - **after triggers**

Triggers in SQL3

- **Condition**:
  - An arbitrary SQL predicate (WHERE SQL clause)
  - It is not "true" if the evaluation of the predicate returns FALSE, or UNKNOWN.
- **Action**:
  - a single SQL command
  - a sequence of commands

```
BEGIN ATOMIC
    first SQL 1 command, second SQL command, ...
END
```

where BEGIN ATOMIC ensures the execution of commands in a single transaction (all trigger commands must be all successful or all undone, restoring the previous state to the SQL command that caused activation).

Triggers in SQL3

- **Action** (Cont.):

  The possible actions are:

  – **Trigger before**: Data definition, data selection, procedure calls; it is not possible to perform actions that change the status of the database, at most, it is possible to modify "new" values in row-level mode (set new.t = expr) before the command that fired the trigger is executed.

  – **Trigger after**: Any command expected from before triggers + data modification operations (INSERT, DELETE, UPDATE)

  Commands may not be related either to the transaction management, or connections, and sessions.

Triggers in SQL3

- The condition / action can be checked / executed
  - FOR EACH ROW (for each tuple involved in the event)
  - FOR EACH STATEMENT (only once for the command that fired the trigger)
    - In this case it should be noted that the execution takes place even if the command that activates the trigger has not actually changed any tuple (because the conditions expressed in the WHERE clause are not met).
- By default triggers are defined at command level.

Triggers in SQL3

- new, old, new_table and old_table variables are implicitly defined and they implement transition tables.
- A transition table refers to all the tuples that have actually been
  - Inserted
  - Deleted
  - Updated
- Two transition tables are used in case of updates: one records the values before the update, while the other records the values after the update.
- These tables can be used in the evaluation of the condition of a rule and / or in the execution of actions of a rule.

Triggers in SQL3

- Transition tables improve the efficiency, limiting the assessment of the conditions of the rules to the actual tuples in the transition tables.
- The REFERENCING clause allows the introduction of variables named in a different way than new, old, new_table and old_table.

Triggers in SQL3

- Tuples that are visible (by means of the variables new, old, new_table, old_table or their re-denominations) during the evaluation of the condition and the execution depend on three factors:
  - From the **event** that activated the trigger
  - From the trigger type (**before / after**)
  - From the execution type (**row / statement**)

Triggers in SQL3

- Event Type:
  - INSERT: tuples inserted are accessible using **new** or **new_table** variables or their re-denominations (REFERENCING NEW clause, at tuple or table level).
  - DELETE: deleted tuples are accessible using **old** or **old_table** variables or their re-denominations (REFERENCING OLD clause, at tuple or table level).
  - UPDATE: previous and current values of the tuples are accessible using **new**, **old**, **new_table** and **old_table** variables, or their re-denominations (clauses REFERENCING OLD and NEW, in terms of tuple or table).

Triggers in SQL3

- Trigger Type
  - *before*:
    - INSERT: any reference to the table on which the insertion is done **does not contain** the new tuples.
    - DELETE: any reference to the table on which the cancellation is done **contains** the deleted tuples
    - UPDATE: the table contains the tuples **as they were before the update**

Triggers in SQL3

- Trigger Type
  - *after*:
    - INSERT: any reference to the table on which the insertion is done **contains** the new tuples.
    - DELETE: any reference to the table on which the cancellation is done **does not contain** the tuples deleted
    - UPDATE: the table contains the **modified tuple**

Triggers in SQL3

- Execution Mode
  - FOR EACH ROW:
    - The REFERENCING clause can reference either a table or a tuple (OLD / NEW -ROW / TABLE)
  - FOR EACH STATEMENT
    - The REFERENCING clause can reference only one table (only OLD / NEW TABLE)

Triggers in SQL3

	OLD ROW	NEW ROW	OLD TABLE	NEW TABLE
before statement	-	-	-	-
before row	delete, update	insert, update	-	-
after statement	-	-	delete, update	insert, update
after row	delete, update	insert, update	delete, update	insert, update

- **OLD** and **OLD TABLE** are not present with the event **insert**
- **NEW** and **NEW TABLE** are not present with the event **delete**

Example of trigger in SQL3

Consider a table **Employee** with attributes **EmpNum** and **Salary**.

```
CREATE TRIGGER LimitEarningsGrowth1
BEFORE UPDATE OF salary ON employee
FOR EACH ROW
WHEN (New.Salary> Old.Salary * 1.2)
SET New.Salary = Old.Salary * 1.2
```

It should be noted that the value of the tuple subjected to modification is reset before the change to the database takes place (and thus, for example, leads to violations of integrity constraints).

Example of trigger in SQL3

```
CREATE TRIGGER LimitEarningsGrowth2
AFTER UPDATE OF salary ON employee
FOR EACH ROW
WHEN (New.Salary> Old.Salary * 1.2)
UPDATE Employee
SET Salary = Old.Salary * 1.2
WHERE EmpNum = New.EmpNum
```

Note that in this case the trigger triggers after the modification of the salary, which is executed in any case, and therefore may violate integrity constraints.

Triggers in SQL3

These two simple examples illustrate the main difference between the triggers of type *before* and *after* in the context of an update:

- *before triggers*: they are used to "condition" the values used by an editing operation. They are also called "conditioners".
- *after triggers*: they are used to "react" to a modification by other operations, which typically cancel unwanted effects. Also said "re-installers".

Example of trigger in SQL3

Consider a table **Account** with attribute **Total** and a table **SingleDeposits** for the registration of individual transactions.

```
CREATE TRIGGER MonitorAccounts
AFTER UPDATE ON Account
REFERENCING OLD AS old NEW AS new
FOR EACH ROW
WHEN (old.NameAccount = new.NameAccount
AND new.Total> old.Total)
INSERT INTO SingleDeposits VALUES
  (new.NameAccount, new.Total-old.Total)
```

Example of trigger in SQL3

Consider a table **`Invoice`** containing invoices and a table **`RemovedInvoices`** for the archive of the removed invoices.

```
CREATE TRIGGER ArchiveRemovedInvoices
AFTER DELETE ON Invoice
REFERENCING OLD TABLE AS SetOldInvoices
INSERT INTO RemovedInvoices
(SELECT * FROM SetOldInvoices)
```

Triggers in SQL3

Details on the Execution Model:

1. Triggers of type **before** with granularity **statement**
2. Triggers of type **before** with granularity **row**
3. The **event** which activated the trigger
4. Triggers of type **after** with granularity **row**
5. Triggers of type **after** with granularity **statement**

Triggers in SQL3

Details on the Execution Model:

How the triggers execution interferes with the constraints expressed in the schema definition?

Example:

```
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 ...;
```

Triggers in SQL3

Example (cont.):

Running an UPDATE on Table1 will result in the execution of the following steps:

1. Trigger2 is activated;
2. the UPDATE operation is performed on Table1;
3. Constraint1 is executed (the constraint is verified at the end of the command execution);
4. Trigger1 is activated (it is the first to be defined);
5. Trigger3 is activated (it is more recent than Trigger1).

Triggers in SQL3

Steps to determine the 'execution order':

1. Selection based on triggers types; different types of triggers are executed in the following order:
   i. BEFORE STATEMENT triggers
   ii. For each tuple involved in the command
      a) BEFORE ROW trigger
      b) execution of the command, and verification of integrity constraints on such tuple, with immediate assessment (NOT DEFERRABLE or DEFERRABLE but with IMMEDIATE execution mode)
      c) AFTER ROW triggers
   – Verification of integrity constraints on the table(*), with immediate evaluation (NOT DEFERRABLE or DEFERRABLE but with IMMEDIATE execution mode)
   – AFTER STATEMENT triggers
2. If there are several triggers of the same type, the order is given by the time of creation

(*) Require that the execution of the command is completed

Triggers in SQL3

Details on the Execution Model.

What happens if the violation of the integrity constraints at point b) causes the activation of other triggers?

Extreme attention must be paid regarding the possibility of recursive activation of the triggers and the joint evaluation of triggers and integrity constraints.

Let's consider a change action **S** carried out on a table **R** in the context of a generic transaction.

If **S** activates any trigger, the evaluation and execution of *before-triggers* is first performed, which can cause changes in **new** values.

Triggers in SQL3

Successively, actions linked to the restore of referential integrity related to **S** are carried out. These actions can result in chained actions (when the **cascade** option is present in the constraint) which can in turn cause the activation of many triggers, either *before* (which are evaluated without causing further changes in the database) and *after* (in addition to *after* triggers activated by **S**).

A set of *after* triggers **I(S)** is then obtained, which are activated directly or indirectly by **S**, and executed according to their priority.

Triggers in SQL3

When the execution of a command **S'** existing in the procedural part of one of the triggers of **I(S)** causes the activation of other triggers, the execution status of the algorithm related to **S** is saved and the system reacts, recursively, to the modification **S'**, executing the corresponding *before* and *after* triggers as described above for **S**.

Upon the completion of **S'**, the status **S** is restored and the execution resumes from where it was suspended.

If during the execution an exception is raised or an error is encountered, all changes made from the transactional primitive that triggers the execution of the trigger, including the same primitive, are undone; therefore, a *partial rollback* of the primitive and all the actions caused by triggers is performed.

Example of cascading triggers in SQL3

Employees

Id	dept
1	A
2	B
3	A
4	A
…	…

Foreign Key
On update CASCADE

Departments

Name
A
B

Command:
UPDATE Departments
SET Name = ‘C’
WHERE Name = ‘A’

UPDATE Employees
SET Dept = ‘C’
WHERE Dept = ‘A’

The command may lead to the execution of other unexpected triggers on other tables:
TRIGGERS UPDATE ON Employees !!!

Triggers in SQL3

Details on the execution model.

Which data structures are used for the trigger execution?

- SQL: 1999 expects that the triggers are handled in a Trigger Execution Context (TEC)
- The execution of the action of a trigger can produce events which activate other triggers, that should be evaluated in a new internal TEC
- At any given moment, there may be more TEC for a transaction, one inside the other, but only one can be active
- For row-level triggers the TEC considers which tuples have already been considered and which are to be considered
- Thus, *a stack structure* is obtained

$$TEC_0 \rightarrow TEC_1 \rightarrow \ldots \rightarrow TEC_n$$

- When a trigger has considered all events, the TEC closes and it is possible to move on the next trigger
- It is a complicated model, but precise and relatively simple to implement

Triggers in SQL3

Details on the execution model.

How to ensure the termination?

Regarding the termination, the SQL3 standard is unclear.

It assumes that the system keeps track of the various activations, through an activation graph that is built when triggers are created.

If the user tries to specify a trigger that could generate non-terminating executions, the creation is not allowed.

Triggers in SQL3

Details on the execution model.

When is the execution of a trigger activated?

- It is automatically enabled after the definition of a trigger has been compiled.
- The command:

  ALTER TRIGGER trigger_name DISABLE | ENABLE

  is used to deactivate / activate a trigger.
- The command

  ALTER TABLE table_name DISABLE | ENABLE ALL TRIGGERS

  is used to deactivate / activate the triggers associated with a table.
- The syntax for deleting a trigger is:

  DROP TRIGGER <Trigger_name>

Triggers in SQL3: Summary

Data Model	Relational (object)
Primitive Events	Database Transactions
Composite Events	No
Transition Tables	Yes (tuples, tables)
Coupling Modes	Immediate
Termination	Syntactic Check
Rules ordering	Type + creation order

Examples of triggers in SQL3 ...

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

```
CREATE TRIGGER Delete_Emp
AFTER DELETE ON Employees
REFERENCING OLD ROW AS Old
FOR EACH ROW
INSERT INTO Deleted_Emps
    VALUES (Old.Emp#);
```

Similar to the trigger **ArchiveRemovedInvoices** but, it operates tuple by tuple and not after all the cancellations.

Examples of triggers in SQL3 ...

Assume that the table **Employees** has the following scheme:
Employees (Emp#,Salary,Dept#,Home_Ph,Office_Ph)
and suppose that, by default, the home phone number must be the same of the office.

We can not meet this requirement using the DEFAULT clause of the CREATE TABLE command because "DEFAULT ColumnName" is not a correct specification.

We will then write a trigger:

```
CREATE TRIGGER Default_Home_Ph
BEFORE INSERT ON Employees
REFERENCING NEW ROW AS New
FOR EACH ROW
  Set New.Home_Ph =
     homeORoffFun(New.Home_Ph,New.Office_Ph);
```

Examples of triggers in SQL3 ...

where the function **homeORoffFun (value1, value2)** is defined as follows:

```
CASE
WHEN value1 IS NOT NULL THEN value1
                        ELSE value2
```

Examples of triggers in SQL3 ...

Assume that the table **Departments** has an attribute **Budget** and that the budget of a department can not be changed after 17:00:

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

Examples of triggers in SQL3 ...

- The action of the previous trigger generates an error; as the mode of execution is immediate, the action and the event which triggered the rule is undone (rollback).
- Therefore:
  - An update on **`Departments`** activates the rule
  - After 5 p.m. the condition is true
  - The action fails
  - The update is undone

Examples of triggers in SQL3 ...

- Specific cases
  - Event:

```
UPDATE Departments Set budget = v1,
  DEPTNAME = v2
```

    - The rule is not activated because the event is different (the update involves two attributes and not just budget)
  - Event:

```
UPDATE Departments SET budget = NULL;
```

    - The rule is enabled
    - If the condition is true, a NULL / 0 division must be computed (which is legal!)
    - Therefore the action does not fail and the update can not be aborted.

Triggers in DB2

- The SQL-3 standard has been greatly inspired by DB2, due to the significant presence of DB2 designers as part of the standardization group.
- Therefore we can consider DB2 as the main achievement of the SQL-3 standard.
- The only substantial difference is the use, in DB2, of procedural statements available within the system, which do not necessarily refer to the SQL-3 standard.

Triggers in DB2

Example:

Consider a database with the following tables:

Part(PartNum, Supplier, City, Cost)

Distributor(NumSupplier, ...)

Audit (User, Date, NModifiedTuples)

Assume the presence of a referential integrity constraint in the table **Part** which refers to the table **Distributor**.

```
FOREIGN KEY (Supplier)
REFEFENCES Distributor
ON DELETE SET NULL
```

Triggers in DB2

Example:
Consider the following two triggers:

```
CREATE TRIGGER UniqueSupplier
BEFORE UPDATE OF Supplier ON Part
REFERENCING NEW AS N
FOR EACH ROW
WHEN (N.Supplier IS NOT NULL)
SIGNAL SQLSTATE '70005' ('Supplier cannot be
  changed')
```

that prevents you from changing the attribute Supplier if not using the *null* value, and …

Triggers in DB2

Example:

```
CREATE TRIGGER AuditPart
AFTER UPDATE ON Part
REFERENCING OLD_TABLE AS OT
FOR EACH STATEMENT
INSERT INTO Audit
VALUES (user, current_date,
    (SELECT COUNT (*) FROM OT))
```

which records in the table **Audit** the number of tuples modified in the table **Part**

Triggers in DB2

Example:

Deleting all providers of "Como" from the table **`Distributor`** causes the violation of the referential integrity constraint.

At this point the management policy of the violations of the integrity constraint causes the change in the *null* value of all the tuples of the table **`Part`** orphaned as a result of cancellations.

This activates the two triggers **`UniqueSupplier`** and **`AuditPart`**; the first is a before-trigger which is then considered first.

Triggers in DB2

Example:

Its evaluation, tuple by tuple, logically occurs before the change. However, we have access to the value **`N`** that characterizes the variation; therefore, this value is found equal to *null*, and the condition is false.

Finally, it is considered and executed the trigger **`AuditPart`** which inserts in the table **`Audit`** one tuple containing the code of the user, the current date and the number of modified tuples.

Triggers in Oracle: Syntax

The basic form of the command to create triggers in PL / SQL:

```
CREATE [OR REPLACE] TRIGGER ⟨TriggerName⟩
⟨TriggerType⟩ ⟨Event⟩ {, ⟨Event⟩}
ON ⟨TableName⟩
[   [REFERENCING ⟨Reference⟩ ]
    FOR EACH ROW
    [ WHEN ( ⟨SQLPredicate⟩ )]]
⟨Block PL / SQL⟩
```

where:

```
⟨TriggerType⟩ :: = BEFORE | AFTER | INSTEAD OF
⟨Event⟩ :: = DELETE | INSERT | UPDATE [OF ⟨Attribute⟩]
⟨Reference⟩ :: = OLD [AS] ⟨VariableOld⟩ | NEW [AS] ⟨VariableNew⟩
```

Triggers in Oracle

In the definition we specify the following information:

- When the trigger must be activated.
- The type of operation that activates the trigger, and possibly about which attribute.
- The trigger granularity, if at the level of tuple (FOR EACH ROW) or at the level of primitive.
- A possible additional condition that must be true in order that the trigger code is executed.
- The code to be executed.

Triggers in Oracle: events

The events that can activate a trigger in Oracle are:

- commands INSERT, DELETE, UPDATE, UPDATE OF ⟨Attribute⟩ on Tables and Views
- commands CREATE, ALTER, DROP of a schema object
- Startup and shutdown of the database
- Specific or generic error
- Connect / disconnect of a user

We will take into consideration the trigger activated by commands on tables and views.

Triggers in Oracle: Consideration

In Oracle, the consideration of a trigger can be:

- Immediately before (BEFORE) or immediately after (AFTER) the triggering event
- In place of (INSTEAD OF) the command that fired the trigger.

Triggers in Oracle: Action

The specified action can be:

- A PL / SQL block or procedure call (DDL and transactional commands are not allowed, as ROLLBACK);
- If the trigger includes several events, the action we can perform various activities based on the activating event. In this case we will use the type of selections:
- IF inserting.
- IF deleting.
- IF updating, IF updating ('⟨Attribute⟩')

Triggers in Oracle: granularity

The activation of the trigger can be:

- at the level of the tuple (FOR EACH ROW)
- a the level of primitive.

Only if the level is that of tuple, we can specify a condition (clause WHEN).

The condition is expressed as an SQL predicate (those used in WHERE), without subqueries and user-defined functions. The condition may involve only the attributes of the modified tuple.

In the case of level of activation of primitive, we may add controls in the PL / SQL block.

Triggers in Oracle: references to the state before and after

- By clause REFERENCING we can refer to status of the tuples before and after the event that fired the trigger.
- The states to which one can refer to is a single tuple, therefore the clause REFERENCING is specified only if the trigger has granularity row-level.
- In the case of *insertion,* only the after state is defined, while in the case of *cancellation*, only the before state is defined. In the case of *modification* both states are defined.
- By default, in the action, the before state of the tuple is indicated as :old, whereas the next state is indicated as :new.
- However, it is possible to introduce new names in the part referencing.

Syntactic differences with SQL3

- Each Oracle trigger check any combination of the three primitives DML (insertion, deletion and modification) on the target table, while in SQL-3 each trigger is triggered by a single event.
- The granularity of the trigger is determined by the clause for each row, which is added in the case of granularity at tuple level, while omitted in the case of granularity primitive level.
- The condition may be present only in triggers of granularity at tuple level and consists of a simple predicate on the current tuple; in the triggers at the level of primitive it is however possible to introduce control structures in the action using the PL/SQL.
- The action part can not contain DDL commands or transactional instructions.

Behavior of triggers in Oracle

Given that there are two types of triggers *immediate* (BEFORE or AFTER) and that they can have a level of granularity of tuple or primitive, we can have, therefore, four types of triggers:

- *before row*
- *before statement*
- *after row*
- *after statement*

The execution of a primitive INSERT, DELETE or UPDATE in SQL is interspersed by the execution of the triggers that are activated by it, according to the following scheme:

Behavior of triggers in Oracle

Order of Execution:

- Run the triggers *before statement*
- For each tuple of the target table involved in the primitive:
  a) Run triggers *before row*
  b) Apply the primitive to the tuple, and run tests on the integrity, if they are executable tuple by tuple.
  c) Runs the triggers *after row*
- Run tests related to the integrity, for the entire table
- Runs the triggers *after statement*

There is a partial rollback of the primitive and all the actions caused by the trigger in the case of exceptions raised that are not handled in the action of the trigger.

NB: Since INSTEAD OF triggers are always distinct from the events that fire other types of triggers, they do not need to be ordered with respect to other types of triggers.

Triggers INSTEAD OF

- They are executed in place of a command.
- They are only for triggers created on views.
- Always at the level of row
- They are useful for performing updates to views that can not be performed directly from the DML commands (INSERT, UPDATE, DELETE).

For example, consider a view that is defined by an aggregation function. Using standard procedures of the DBMS, we can not run DELETE on the view. The solution is to define a trigger of type INSTEAD OF which is activated on DELETE on the view. The trigger action will change the table on which the view is defined according to the chosen semantics. When we run the DELETE statement on the view, the trigger is executed instead of the delete.

Updatable views

A view can be changed when:

- The SELECT does not have the clause DISTINCT and attribute values of the view are not calculated;
- The FROM clause concerns only one base table or virtual, in turn updatable, that is, virtual tables obtained with the join are excluded;
- The WHERE clause does not contain subselects;
- There are no GROUP BY and HAVING operators;
- The columns defined in the base table with the NOT NULL constraint must be part of the virtual table

Updatable views

Employees

Departments

View 1: CREATE VIEW Big_Depts AS
SELECT * FROM Departments WHERE #EMP>100

Query 1: SELECT * FROM Big_Depts

Query 2: INSERT INTO Big_Depts VALUES
('CS', 157, …)

Query 3: INSERT INTO Big_Depts VALUES
('CS', 57, …)

Updatable views

Employees

Departments

View 1: CREATE VIEW Big_Depts AS
SELECT * FROM Departments WHERE #EMP>100

Query 1: SELECT * FROM Big_Depts 

Query 2: INSERT INTO Big_Depts VALUES
('CS', 157, …)

Query 3: INSERT INTO Big_Depts VALUES
('CS', 57, …)

Updatable views

Employees

Departments

View 1: CREATE VIEW Big_Depts AS
SELECT * FROM Departments WHERE #EMP>100

Query 1: SELECT * FROM Big_Depts

Query 2: INSERT INTO Big_Depts VALUES
('CS', 157, …)

INSERT INTO Departments VALUES
('CS', 157, …)

Updatable views

Employees

Departments

View 1: CREATE VIEW Big_Depts AS
SELECT * FROM Departments WHERE #EMP>100

Query 1: SELECT * FROM Big_Depts 

Query 2: INSERT INTO Big_Depts VALUES
('CS', 157, …) 

Query 3: INSERT INTO Big_Depts VALUES
('CS', 57, …)

Updatable views

View 2: CREATE VIEW Emp_Depts AS
SELECT * FROM Departments, Employees
WHERE Departments.name = Employees.dept

Query: INSERT INTO Emp_Depts VALUES
(‘CS’, 157, …)

Updatable views

Employees

Departments

View 2: CREATE VIEW Emp_Depts AS
SELECT * FROM Departments, Employees
WHERE Departments.name = Employees.dept

Query: INSERT INTO Emp_Depts VALUES
('CS', 157, …)

Triggers in Oracle: restrictions

- A table is mutating if it is the table on which a command is being performed, which activates the trigger.
- Row type triggers cannot modify mutating tables through INSERT, DELETE, UPDATE actions provided in the action block.
- In the case of ***after*** triggers, they can not even read. In contrast, triggers ***before*** can read.
- This is a strong restriction.
- The motivation is to avoid that a trigger manipulates data that may be inconsistent or that exhibits behaviors that may depend on the order according to which the tuples are modified during the execution of the command.

Mutating tables: motivating example

```
UPDATE  EMPLOYEES
SET SALARY=SALARY+500
WHERE DEPARTMENT='CS'
```

employees				
ID	NAME	SURNAME	SALARY	DEPARTMENT
1	MARIO	ROSSI	4000	CS
2	ALDA	BIANCHI	4500	CS
3	MICHELE	VERDI	3500	CS
4	IRENE	GIALLI	6000	M
5	UGO	FANTO	2000	M

```
CREATE TRIGGER LIMIT_INCREASE_SALARY
   BEFORE UPDATE OF SALARY ON EMPLOYEES
   FOR EACH ROW
BEGIN
  IF(SELECT SUM(SALARY) FROM EMPLOYEES WHERE
   DEPARTMENT=:NEW.DEPARTMENT)>=13000
   THEN :NEW.SALARY := :OLD.SALARY
  ENDIF
END
```

Behavior of triggers in Oracle

- *How many triggers can be defined for each primitive?*

There is no limitation for Oracle 7.1 and later versions.

- *How to prevent non terminating computations?*

The maximum number of cascaded trigger is 32; reached this threshold, the system assumes an situation of infinite execution and suspends the execution, raising a specific exception.

Examples of Trigger

Example 1:

Suppose in the following database:

Customers(CustomerCode, SurnameAndName, City, Discount)

Agents(AgentCode, SurnameAndName, Zone, Supervisor, Commission)

Orders(OrderNumb, CustomerCode, AgentCode, Product, Date, Amount)

it is required to impose the constraint such that new orders are not accepted from customers with an overdraft of 5000 Euro.

Examples of Trigger

Example (cont):

```
CREATE TRIGGER CreditCheck
AFTER INSERT ON Orders
FOR EACH ROW
DECLARE ToBePaid NUMBER;
BEGIN
        SELECT SUM (Amount) INTO ToBePaid
        FROM Orders
        WHERE CustomerCode = :new.CustomerCode;
        IF ToBePaid> = 5000 THEN
                RAISE_APPLICATION_ERROR (-2061, 'Credit limit
   exceeded');
        END IF;
END;
```

Examples of Trigger

Example 2:

We show the use of a trigger to maintain a stored table, which is automatically updated on the basis of another table, without the use of views. The following program creates a new table of pairs (agent, total number of orders placed), and a trigger that, *from this moment on*, automatically maintains the new table updated.

Examples of Trigger

Example:

```
CREATE TABLE TOTALS (AgentCode CHAR(3), TotalOrders INTEGER);

CREATE TRIGGER exampleTrig
AFTER INSERT ON Orders
FOR EACH ROW / * For each tuple being entered * /
DECLARE exists NUMBER;
BEGIN
/ * It checks to see if the agent is already present in the order of the total
    table * /
    SELECT COUNT (*) INTO exists FROM Totals
    WHERE AgentCode =: new.AgentCode;
    ...
```

Examples of Trigger

Example (cont.):

```
    IF exists = 0  / * The agent is absent; we must insert a new tuple * /
            THEN
             INSERT INTO Totals
             VALUES (:new.AgentCode,:new.Amount);
                                                                    ELSE
             UPDATE Totals
             Set TotalOrders = TotalOrders + :new.Amount
             WHERE AgentCode = :new.AgentCode;
END;
```

Examples of Trigger

Example 3:

It is requested to automatically generate a new order (via insertion of a tuple in the table PendingOrders) each time the available amount AvailQty of a particular part in the table Warehouse falls below a specific reorder threshold (ThreshQty).

Warehouse

Part	AvailQty	ThreshQty	ReordQty
1	200	150	100
2	780	500	200
3	450	400	120

Examples of Trigger

Example (cont): CREATE TRIGGER Reorder

```
AFTER UPDATE of AvailQty ON Warehouse
FOR EACH ROW
WHEN ( :new.AvailQty < :new.ThreshQty)
DECLARE x NUMBER;
BEGIN
        SELECT COUNT (*) INTO x
        FROM PendingOrders
        WHERE Part = :new.Part;

        IF x = 0 THEN
                INSERT INTO PendingOrders
                VALUES (:new.Part,: new.ReordQty, sysdate)
        END IF;
END;
```

Examples of Trigger

Warehouse

Part	AvailQty	ThreshQty	ReordQty
1	200	150	100
2	780	500	200
3	450	400	120

Consider the following transaction:

```
UPDATE WAREHOUSE
SET AvailQty = AvailQty - 70
WHERE Part = 1
```

This transaction causes the activation, the consideration, and the execution of the Reorder trigger, leading to the insertion of the tuple:

(*1, 100, 1-1-2000*)

in the table PendingOrders.

Examples of Trigger

Warehouse

Part	AvailQty	ThreshQty	ReordQty
1	200	150	100
2	780	500	200
3	450	400	120

Consider the following transaction:

```
UPDATE WAREHOUSE
SET AvailQty = AvailQty - 60
WHERE Part <= 3
```

The trigger is executed for all the parts, and the condition is verified for the parts 1 and 3.

The action relative to the part 1 has no effect.

Then, the execution of the trigger involves the insertion in PendingOrders of the single tuple:

(*3, 120, 1-1-2000*)

Examples of Trigger

Check that the salary of each employee is within the limits permitted by the specific job:

```
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;
    IF ( :new.Salary < :minsal OR :new.Salary > :maxsal)
    THEN raise_application_error (-20601, 'salary outside the range for employee' ||
    :new.Name);
    END IF;
    END;
```

Examples of Trigger

Same trigger but with a procedure call to VerifySalary whose body 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);
```

Examples of Trigger

An example of a trigger with INSTEAD OF. Suppose you have a schema with three tables that define employees, departments and projects of a company.

```
CREATE TABLE Projects (
  Prj_level                NUMBER,
  Projno                   NUMBER,
  Resp_dept        NUMBER);

CREATE TABLE Departments (
  DEPTNO           NUMBER (2) NOT
NULL,
  Dname            VARCHAR2 (14),
  Loc          VARCHAR2 (13),
  Mgr_no           NUMBER,
  Dept_type        NUMBER);
```

```
CREATE TABLE Employees (
  Empno          NUMBER NOT NULL,
  Ename          VARCHAR2 (10),
  Job            VARCHAR2 (9),
  Mgr            NUMBER (4),
  HireDate       DATE,
  Sal            NUMBER (7.2),
  Comm           NUMBER (7.2),
  DeptNo         NUMBER (2) );
```

A project is assigned to a department for which we know the manager, who is one of the employees.

Examples of Trigger

Suppose to define the following view with information about the project manager.

```
CREATE OR REPLACE VIEW manager_info
AS      SELECT e.ename, e.empno, d.dept_type, d.deptno, p.prj_level,
p.projno
        FROM Employees e, Departments d, Projects p
        WHERE e.empno = d.mgr_no AND d.deptno = p.resp_dept;
```

It is not possible to directly make any change on the view, because the DBMS will not know how to automatically update the source tables.

A trigger is a valid alternative.

Examples of Trigger

```
CREATE OR REPLACE TRIGGER manager_info_insert
INSTEAD OF INSERT ON manager_info
REFERENCING NEW AS n
FOR EACH ROW
DECLARE
  rowcnt number;
BEGIN
  SELECT COUNT(*) INTO rowcnt FROM Employees WHERE empno = :n.empno;
  IF rowcnt = 0  THEN
        INSERT INTO Employees(empno,ename) VALUES (:n.empno, :n.ename);
  ELSE
        UPDATE Employees SET Employees.ename = :n.ename
        WHERE Employees.empno = :n.empno;
  END IF;
  SELECT COUNT(*) INTO rowcnt FROM Departments WHERE deptno = :n.deptno;
  IF rowcnt = 0 THEN
        INSERT INTO Departments(deptno, dept_type)
VALUES(:n.deptno,:n.dept_type);
  ELSE
        UPDATE Departments SET Departments.dept_type = :n.dept_type
        WHERE Departments.deptno = :n.deptno;
```

Examples of Trigger

```
….
  SELECT COUNT(*) INTO rowcnt FROM Projects
          WHERE Projects.projno = :n.projno;
  IF rowcnt = 0 THEN
          INSERT INTO Projects (projno, prj_level) VALUES(:n.projno, :n.prj_level);
  ELSE
          UPDATE Projects SET Projects.prj_level = :n.prj_level
          WHERE Projects.projno = :n.projno;
  END IF;
END;
```

Triggers in Oracle: Summary

Data Model	Relational (object)
Primitive Events	Operations on Database
Composite Events	Yes
Transition Tables	Yes (tuples)
Coupling Modes	Immediate
Termination	Based on timeout and max limit
Rules ordering	Type + non-deterministic choice

Exercise

Given the following relational schema
EMPLOYEE(Name, Dept)
DEPARTMENT(DeptCode, DeptName, City, Budget)
and the view BARESI defined as:

```
SELECT Name
FROM Employee, Department
WHERE Dept = DeptCode AND City = "Bari"
```

define triggers to incrementally maintain this view.

Exercise

The materialized views are relationships actually stored in the database, as opposed to virtual relations (simply called views) that are not stored in the database but can be used in as they were queries. Active Rules are also appropriate to preserve the coherence of *materialized views* when basic relationships are modified.

This application is also fundamental for new technologies of Data Warehousing.

Exercise

In the exercise, the events that can cause recomputation of the materialized view are:

- insertion of a new employee;
- deletion of an employee;
- update of the name and the department of an employee;
- update of the city of the department.

Exercise

We do not expect any update in case of insertion of a new department, because the existence of a referential integrity constraint between the Dept attribute of Employee and Department is assumed. So when a new department is inserted we are sure that there are no employees in that department. For the same reason, no update is expected for the view Baresi when a department is deleted.

Exercise

```
CREATE TRIGGER NewEmployee
AFTER INSERT ON Employee
FOR EACH ROW
WHEN 'Bari' = (    SELECT City FROM Department
                                    WHERE    DeptCode    =
      :new.Dept)
INSERT INTO Baresi
VALUES (:new.Name)
```

Exercise (Cont.)

```
CREATE TRIGGER RemoveEmployee
AFTER DELETE ON Employee
FOR EACH ROW
DELETE FROM Baresi
WHERE Baresi.Name = :old.Name

CREATE TRIGGER ModifyEmployeeName
AFTER UPDATE OF Name ON Employee
FOR EACH ROW
UPDATE Baresi
SET Baresi.Name = :new.Name WHERE Baresi.Name = :old.Name
```

Exercise (Cont.)

```
CREATE TRIGGER Employee-in-Bari
AFTER UPDATE OF Dept ON Employee
FOR EACH ROW
WHEN new.Dept IN (   SELECT DeptCode FROM Department
                                                WHERE City = Bari)
      AND old.Dept NOT IN (SELECT DeptCode FROM Department
                                                WHERE  City  =
      Bari)
INSERT INTO Baresi
VALUES (:new.Name)
```

Exercise (Cont.)

```
CREATE TRIGGER Employee-leaves-Bari
AFTER UPDATE OF Dept ON Employee
FOR EACH ROW
WHEN :old.Dept IN (   SELECT DeptCode FROM Department
                                          WHERE City = Bari)
     AND      :new.Dept  NOT  IN  (SELECT  DeptCode  FROM
     Department WHERE City = Bari)
DELETE FROM Baresi
WHERE Baresi.Name =: old.Name
```

Exercise (Cont.)

```
CREATE TRIGGER Department-to-Bari
AFTER UPDATE OF City ON Department
FOR EACH ROW
WHEN .old.City <> 'Bari' AND .new.City = 'Bari'
INSERT INTO Baresi
      SELECT Name FROM Employees
                                                  WHERE  Dip  =
      :old.DeptCode
```

Exercise (Cont.)

```
CREATE TRIGGER Department-away-from-Bari
BEFORE UPDATE OF City ON Department
FOR EACH ROW
WHEN .old.City = 'Bari' AND .new.City <> 'Bari'
DELETE FROM Baresi
WHERE Baresi.Name IN (      SELECT Name FROM Employees
                                                    WHERE      Dept      =
      :old.DeptCode)
```

It should be observed that the activation graph of these rules has no edges but only nodes.

Exercise

Given the relational schema:
EMPLOYEE(Name, Salary, NumDept)
DEPARTMENT(NumDept, ManagerName)
Define the following active rules:

Exercise (Cont.)

- A rule which, when a department is deleted, sets a default value (99) NumDept of employees belonging to that department;

```
create trigger T1
before delete on Department
for each row
UPDATE Employee
SET NumDept = 99
WHERE NumDept = :old.NumDept
```

Exercise

- A rule that deletes all the employees belonging to a department when it is deleted;
- A rule that, whenever the salary of an employee exceeds the salary of his managers, sets such salary equal to the salary of the manager;
- A rule that, whenever salaries are modified, verifies that there are no departments where the average salary grows more than two percent, and in that case aborts the modification;

Advanced features of active rules

The examined features can be seen in commercial relational DBMS.

Some advanced prototypes of active databases have several interesting features, which enhance the expressive power of active rules.

- Possibility to define events of different nature. E.g.:
- ***Temporal event***, which specifies an absolute point on the time axis (23:59:59 31/12/2008), a relative one (1 hour after the insertion of a tuple in the customer table) or a series of periodic points (at 17:00:00 every Friday);

Advanced features of active rules

- ***Request of execution of an operation or a method possibly defined by a user***.
- ***External event***, as a message from another process, a signal from a physical device;
- ***Composite event***, obtained by composing basic events with operators like disjunction (if the event *a* or *b* occurs), sequence (if the event *a* and *b* occur in sequence), or repetition (if the event *a* occurs 3 times in a row).

Advanced features of active rules

- Ability to define different coupling models of the execution of the action of a rule with the execution of the transaction that generated the event.
  The three basic models, which do not satisfy all possibilities, are: *immediate* (the action is immediately executed as part of the transaction) *deferred* (the action is executed at the end of the transaction, but before the *commit*) ***decoupled*** (the action is considered as a separate transaction performed concurrently with the one that has generated the event).

Advanced features of active rules

- The conflicts between rules activated simultaneously and present in ***the conflict set*** can be solved by ***explicit priority***, that is, defined by the user when creating the rules. Priorities are expressed either as a *partial* ordering (by precedence relationships between rules) or as *total* ordering (through numerical priority).
- The rules can be organized into groups and each group may be separately *enabled* and *disabled*.

Advanced features of active rules

- Presence of a programming environment to design and implement the rules with tools, analyze them (for example, to get information on the possibility of non-termination due to cycles), test them and to have a track of their execution.

Priorities in Oracle 11g

Consider the following example:

```
CREATE TABLE trigger_follows_test (
  id NUMBER,
  description VARCHAR2 (50));

CREATE OR REPLACE TRIGGER
trigger_follows_test_trg_1
BEFORE INSERT ON trigger_follows_test
FOR EACH ROW
BEGIN dbms_output.put_line (
'TRIGGER_FOLLOWS_TEST_TRG_1 - Executed');
END;
```

Priorities in Oracle 11g

```
CREATE OR REPLACE TRIGGER
trigger_follows_test_trg_2
BEFORE INSERT ON trigger_follows_test
FOR EACH ROW
BEGIN dbms_output.put_line (
'TRIGGER_FOLLOWS_TEST_TRG_2 - Executed');
END;
```

The execution of the trigger will lead to the following result:

```
SQL> set SERVEROUTPUT ON
SQL> INSERT INTO trigger_follows_test VALUES (1,
'ONE');
TRIGGER_FOLLOWS_TEST_TRG_1 - Executed
TRIGGER_FOLLOWS_TEST_TRG_2 - Executed
1 row created.
```

Priorities in Oracle 11g

However, redefining trigger_follows_test_trg_1 using the FOLLOWS clause it is possible to establish a partial order on the execution of the trigger:

```
CREATE OR REPLACE TRIGGER
trigger_follows_test_trg_1
BEFORE INSERT ON trigger_follows_test
FOR EACH ROW
FOLLOWS trigger_follows_test_trg_2
BEGIN dbms_output.put_line (
'TRIGGER_FOLLOWS_TEST_TRG_1 - Executed');
END;
/
```

Priorities in Oracle 11g

Now the output will be as follows:

```
SQL> set SERVEROUTPUT ON
SQL> INSERT INTO trigger_follows_test VALUES (2,
'TWO');
TRIGGER_FOLLOWS_TEST_TRG_2 - Executed
TRIGGER_FOLLOWS_TEST_TRG_1 - Executed

1 row created.

SQL>
```

Oracle 11g: DDL trigger

In Oracle DDL triggers can be defined on the following events:
BEFORE / AFTER ALTER
BEFORE / AFTER ANALYZE
BEFORE / AFTER ASSOCIATED STATISTICS
BEFORE / AFTER AUDIT
BEFORE / AFTER COMMENTS
BEFORE / AFTER CREATE
BEFORE / AFTER DDL
BEFORE / AFTER disassociate STATISTICS
BEFORE / AFTER DROP
BEFORE / AFTER GRANT
BEFORE / AFTER NOAUDIT
BEFORE / AFTER RENAME
BEFORE / AFTER REVOKE
BEFORE / AFTER TRUNCATE
AFTER SUSPEND

Triggers in MS SQL Server

- A trigger is an object that is "*attached* " to a table

- Three types of triggers in SQL Server 2005
    - Instead of Triggers
    - After Triggers
    - Trigger for Data Definition Language
- DML triggers use the logical tables **deleted** and **inserted** (OLD TABLE and NEW TABLE, respectively)

In SQL Server triggers are primarily used to evaluate constraints.

Trigger working on a **context of tuple (row) are not supported** in SQL SERVER 2005

Triggers in MS SQL Server: Syntax

```
CREATE TRIGGER trigger_name
ON { table | view }
[ WITH ENCRYPTION ]
{
   { { FOR | AFTER | INSTEAD OF } { [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ]
     }
      [ WITH APPEND ]
      [ NOT FOR REPLICATION ]
      AS
      [ { IF UPDATE ( column )
         [ { AND | OR } UPDATE ( column ) ]
            [ ...n ]
      | IF ( COLUMNS_UPDATED ( ) { bitwise_operator } updated_bitmask )
            { comparison_operator } column_bitmask [ ...n ]
      } ]
      sql_statement [ ...n ]
   }
}
```

Triggers in MS SQL Server

trigger_name
Name of the trigger. This name must conform to the rules for MS SQL Server identifiers and must be unique into the database. We can also specify the owner of the trigger.

Table | view
It represents the table (or view) on which the trigger can be defined. A view can be used only for triggers of type INSTEAD OF.

WITH ENCRYPTION
It indicates that SQL Server will convert the original text of the CREATE TRIGGER into an encrypted format. This code can (and must), however, be decrypted by SQL Server in the moment the trigger has to be executed. This means (especially in SQL Server 2000, which can be decrypted).

Using WITH ENCRYPTION we prevent the trigger code to be published as part of the replication of a database.

Triggers in MS SQL Server

AFTER
It specifies that the trigger is executed only after all the commands that have fired the trigger are properly made. All the "cascade" actions and constraint checks are performed before the trigger.
AFTER is the default option, if it is not used the clause FOR.
The triggers of type AFTER can not be defined on views.

INSTEAD OF
It specifies that the trigger is performed as an alternative to the statement that fired the trigger.
At the most it can only be defined one trigger of type INSTEAD OF for INSERT, UPDATE, DELETE. However, we can define views on views, where each view has its own INSTEAD OF trigger.
The INSTEAD OF triggers are not applicable to udatable views defined using the WITH CHECK OPTION clause.

Triggers in MS SQL Server

{[DELETE] [,] [INSERT] [,] [UPDATE]}
Event Type. If more events are specified, they are separated by commas. For INSTEAD OF triggers, the DELETE option is not allowed on tables that have the cascade action ON DELETE. Similarly, the UPDATE option is not allowed on tables that have the cascade option to ON UPDATE.

WITH APPEND
Used to maintain compatibility. If a trigger has a low compatibility level, then it can be specified.
WITH APPEND can not be used with INSTEAD OF triggers or AFTER but only with FOR triggers type.

Triggers in MS SQL Server

NOT FOR REPLICATION
It indicates that the trigger is disabled during database replication.

AS
It allows you to express actions.
Transact-SQL (T-SQL)

Triggers in MS SQL Server

SQL Server provides four different ways to determine the effect of DML commands.

- INSERTED (NEW TABLE) and
- DELETED (OLD TABLE), called MAGIC TABLES
- update ()
- columns_updated()

- The Magic Tables do not contain information about columns of type text, ntext, or image. Attempting to access these columns causes an error at run-time.

EVENT	INSERTED	DELETED
Insert	Contains the inserted rows	Empty
Delete	Empty	Contains the rows to be deleted
Update	Contains the rows after update	Contains the rows before update

Triggers in MS SQL Server: After (Trigger priority)

```
CREATE TABLE SOURCE (SOU_ID INT IDENTITY, SOU_DESC VARCHAR(10)
     GO
     CREATE TRIGGER TR_SOURCE_INSERT ON SOURCE AFTER INSERT
      AS
     PRINT GETDATE()
     GO

   INSERT SOURCE (SOU_DESC) VALUES ('TEST 1')

   -- RESULTS -- APR 28 2001 9:56AM
```

A table can have multiple AFTER triggers for each of the three events INSERT, DELETE and UPDATE

In this case it is possible to define an order by the procedure sp_settriggerorder (In T-SQL)

Triggers in MS SQL Server: After (Trigger priority)

sp_settriggerorder [@triggername =] '[triggerschema.] triggername'
, [@order =] 'value'
, [@stmttype =] 'statement_type'
[, [@namespace =] { 'DATABASE' | 'SERVER' | NULL }]

Where @order (formal parameter) can take value

- 'First': the trigger is executed first
- 'Last': the trigger is executed last
- 'None': the trigger is executed in an undefined order

Triggers in MS SQL Server: code encryption

In SQL Server, as in the case of views and stored procedures, triggers can be encrypted and stored in an unreadable format. Neither the owner nor the administrator will be able to see the code.

```
CREATE TRIGGER TRGENCRYPTED

ON DBO.DTL1 WITH ENCRYPTION

FOR INSERT

AS

BEGIN

      PRINT ('AFTER TRIGGER [TRGENCRYPTED] ENCRYPTED  - TRIGGER EXECUTED !!')

END

GO
```

```
SELECT

            SYSOBJECTS.NAME AS [TRIGGER NAME],

            SUBSTRING(SYSCOMMENTS.TEXT, 0, 26) AS [TRIGGER DEFINITION],

            OBJECT_NAME(SYSOBJECTS.PARENT_OBJ) AS [TABLE NAME],

            SYSCOMMENTS.ENCRYPTED AS [ISENCRPTED]


FROM

            SYSOBJECTS INNER JOIN SYSCOMMENTS ON SYSOBJECTS.ID = SYSCOMMENTS.ID

WHERE

            (SYSOBJECTS.XTYPE = 'TR')
```

	TRIGGER NAME	TRIGGER DEFINITION	TABLE NAME	ISENCRPTED
1	trgEncrypted	□□□□□□□□□□□□□□□□□□□□□□□□□□	DTL1	1

DDL Triggers in MS SQL Server

- Unlike most of the DBMSs (PostgreSQL, DB2), SQL Server supports DDL triggers that can be activated on events Create, Alter, Drop, LOGON.

Use of triggers

Depending on their use, we can divide the triggers in

- **Passive,** if they are designed to raise a failure under certain conditions.
- **Active** if, at certain events, they change the state of the database.

We list some possible uses of passive and active triggers:

Use of triggers

Passive triggers:

- To define integrity constraints.

The integrity constraints defined in SQL are classified as:

- ***Predefined:*** They are defined by language clauses, for example, primary key, unique, not null. Referential integrity constraints belong to this category.
- ***Generic***: Predicates that express conditions that must be true (clause check). There are often many restrictions on predicates expressible with these clauses.

Examples: Intra-relational constraint that expresses the permitted values of an attribute

```
Amount      INTEGER CHECK (Amount> 99)
Discount    INTEGER NOT NULL CHECK (Discount> 0 AND Discount
<100)
```

Use of triggers

Example: constraint between the values of different attributes of the same table. We can not involve other tables in the condition.

```
CREATE TABLE Agents
(   AgentCode                   CHAR (3) NOT NULL UNIQUE,
    SurnameAndName    CHAR (30) NOT NULL,
    Area                                CHAR (8) NOT NULL,
    Supervisor                  CHAR (3),
    Commission                  INTEGER,
    PRIMARY KEY (AgentCode)
    CHECK (Supervisor <> AgentCode OR Supervisor IS NULL)
)
```

Use of triggers

- Thanks to the CHECK clause is also possible to define a further component of a database schema: the ***assertions***.
- Introduced in SQL-2, assertions represent constraints that can not be associated to any attribute or table, but directly to the schema.
- Assertions can be checked immediately after each change of the state of the database (***immediate control***) or only at the end of a set of operations (***deferred control***).

The commands:

- SET CONSTRAINTS IMMEDIATE
- SET CONSTRAINTS DEFERRED

allow us to specify the mode or change from one mode to another.
The syntax for defining assertions is:

CREATE ASSERTION ⟨Assertion Name⟩ CHECK ⟨Condition⟩

Use of triggers

Example:

```
CREATE ASSERTION At_least_one_agent
CHECK (1 <= (SELECT COUNT (*) FROM Agents))
```

The problems we may have, using check or assertion clauses are:

- Restrictions on predicates expressible in the *check* clause;
- Need to adopt a reaction policy similar to that of the standard, which could be different from the desired one.

Active Rules are appropriate to implement referential integrity constraints or to specify and implement truly "generic constraints", which can not be expressed in the used data model (e.g., dynamic constraints - define conditions on how the knowledge can evolve over time).

Use of triggers

Example (DB2):

An employee must not have a salary greater that of the director of the department to which he/she pertains.

```
CREATE TRIGGER ExcessiveSalary
AFTER UPDATE ON Salary OF Employee
FOR EACH ROW
WHEN New.Salary>
(SELECT Salary FROM Employee WHERE ID_Emp IN
 (SELECT Director FROM Department WHERE ID_Dep = new.ID_Dep))
SIGNAL SQLSTATE '70006' ( 'employee without department');
```

Use of triggers

Passive triggers:

- For controls on users' eligible operations based on values of the SQL commands parameters.

Example: We can enter certain data only when the department code is that of the user who is running the operation.

Use of triggers

Active triggers:

- To maintain some integrity constraints by modifying the database in a suitable way.

Example (Oracle): keep the department ID of an employee updated when it changes.

```
CREATE OR REPLACE TRIGGER cascade_updates
AFTER UPDATE OF deptno ON dept
FOR EACH ROW
BEGIN
    UPDATE emp
    SET    emp.deptno = :new.deptno
    WHERE  emp.deptno = :old.deptno;
END
```

Example When a tuple is deleted, the referential constraint can be maintained in an active way by deleting all tuples that refer to the deleted tuple.

Use of triggers

Active triggers:

- To store events into the database for control purposes (*auditing and logging*).

Example (Oracle): counts the number of employees deleted by a user

```
CREATE OR REPLACE TRIGGER audit_emp
AFTER DELETE ON emp
FOR EACH ROW
BEGIN
     UPDATE audit_table SET del= del + 1
     WHERE user_name = USER AND table_name = 'EMP';
END;
```

Example: In a dedicated table we can record data about executed transactions on reserved tables (and terminated normally).

Use of triggers

- To propagate on other tables the effects of certain operations on tables (calculated tables).

Example (Oracle): the cancellation of an exam requires to store it in a historical archive.

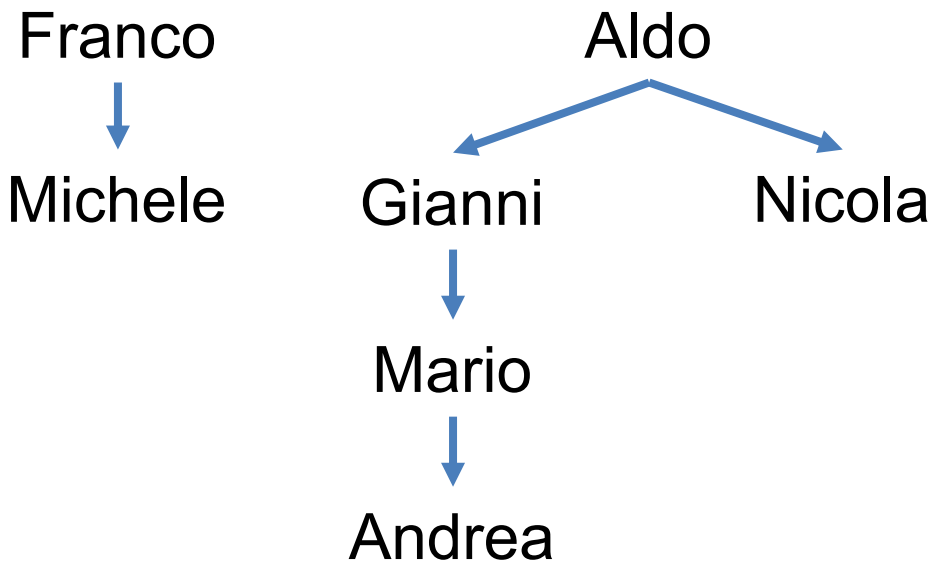
```
CREATE TRIGGER Delete_Test
AFTER DELETE ON Tests
REFERENCING OLD ROW AS Old
FOR EACH ROW
BEGIN
INSERT INTO Previous_Tests
     VALUES (Old.Exam#,Old.Test_Date);
END
```

Use of triggers

- To handle recursion in those DBMS that do not support it at all, or that limit its expression.

Example: Family Tree

Father	Son
Aldo	Gianni
Gianni	Mario
Franco	Michele
Aldo	Nicola
Mario	Andrea

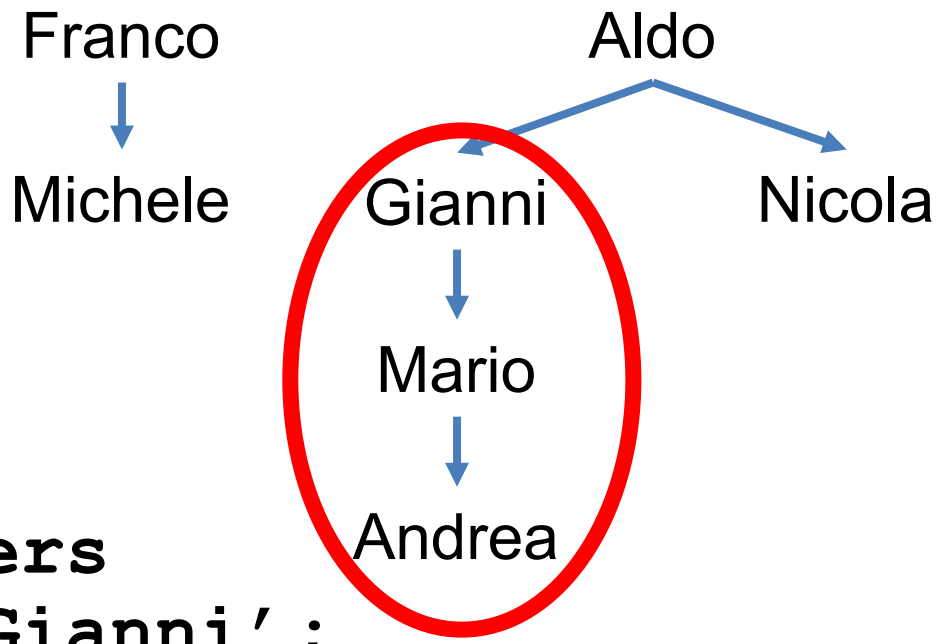


Use of triggers

- To handle recursion in those DBMS that do not support it at all, or that limit its expression.

Example: Family Tree

Father	Son
Aldo	Gianni
Gianni	Mario
Franco	Michele
Aldo	Nicola
Mario	Andrea

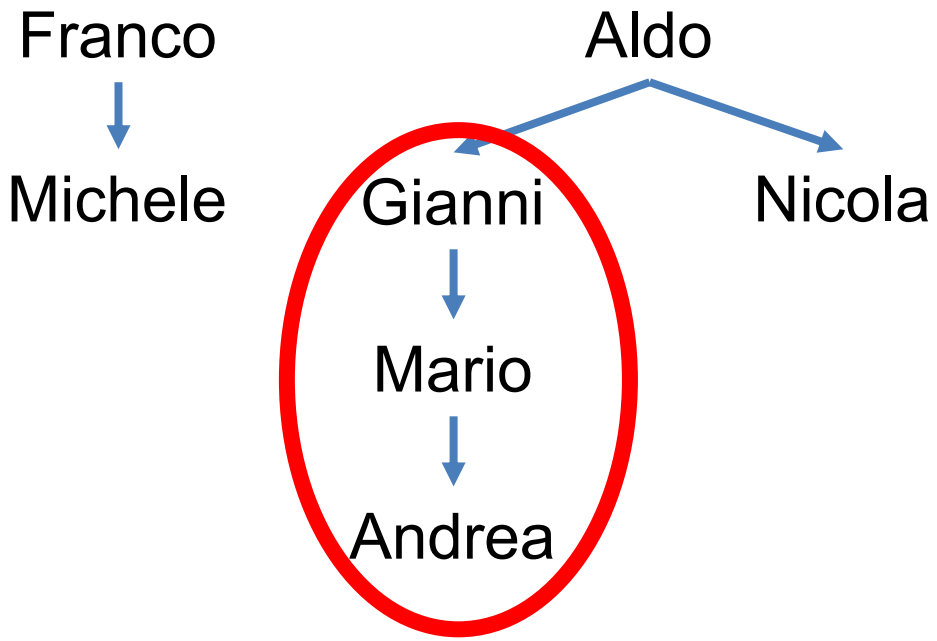



```
DELETE FROM Fathers
WHERE Father = 'Gianni';
```

Recursive delete the subtree associated with a father

Use of triggers

Father	Son
Aldo	Gianni
Gianni	Mario
Franco	Michele
Aldo	Nicola
Mario	Andrea




```
CREATE TRIGGER Del_branch
AFTER DELETE ON Father
FOR EACH ROW
DELETE Father
WHERE Father.Father = OLD.Son
```

Use of triggers

- To keep updated possible redundant data, upon changes.

Example (Oracle): update the commissions of the salesman on the basis of his salary.

```
CREATE OR REPLACE TRIGGER
    derive_commission_trg
BEFORE UPDATE OF sal ON emp
FOR EACH ROW
WHEN (new.job = 'SALESMAN')
BEGIN
    :new.comm:= :old.comm*(:new.sal/ :old.sal);
END;
```

Use of triggers

Active triggers:

- To define the so-called *business rules*, or actions to be performed to ensure proper evolution of the information system.

Example: Actions for automatic inventory maintenance.

The expression of reactive business rules at the schema level (and thus valid for all applications) allows a single, centralized specification. This allows us to enforce the property of **independence of knowledge**.

Independence of knowledge

Reactive Knowledge is subtracted to application programs and codified in the form of active rules.

Recall that databases offer the two dimensions of physical independence (a program does not know the physical organization of data) and logic independence (a program can see the data by means of appropriate external schemas or logical views).

The advantage of "independence of knowledge" is that the application logic on the reactive behavior is defined only once and is shared by all applications.

Use of triggers

We can also distinguish between the use of triggers:

1) ***internal*** use (to the database): the manager of the active rules works as a subsystem of the DBMS to implement some of its functionalities. The administrator of the DBMS has the ability to give a ***declarative specification*** to derive, in whole or in part, the code of active rules.

Features that can be given to active rules:

- Integrity Management.
- Calculation of derived data.
- Management of the replicated data.
- Management of versions.
- Management of privacy and data security.
- Logging of events.
- Management of recursion.

Use of triggers

We can also distinguish between the use of triggers:

2) ***external*** use (to the database): relative to the specific application. Active Rules expressing application knowledge are the *business rules*.

Examples of external uses are:

- Purchase and sale of shares according to market fluctuations.
- Warehouse management based on the change of intervention stock.
- Management of a transport network or energy network.

Some business rules are simple ***alerters*** which only emit messages and alerts.

Classification of the triggers

S. Ceri, R.J. Cochrane and J. Widom (2000) proposed a Trigger classification based on the behavior and functions:

1. **Constraint-preserving:** report the violation of integrity constraints and force the rollback of transactions that cause it.
2. **Constraint-restoring:** detect the violation of integrity constraints and modify the database so as to restore its integrity.
3. **Invalidating:** report the violation of integrity constraints, so as to allow the applications to respond appropriately.
4. **Materializing:** compute derived information.
5. **Metadata:** maintain the consistency of metadata and catalogs of the DBMS.
6. **Replication**: Replicate, migrate, or record the information and/or changes from a table or from a database (the primary copy) in another (the secondary copy). Used in data warehouses.
7. **Extender**: Handle new data types (eg., Validate the input) and maintain specialized external data structures consistent with the database.
8. **Alerter**: Notify the information to users in the form of messages.
9. **Ad hoc**: Implement business rules, scheduling of tasks, workflow management, supply management, or other application-specific logics.

Properties of active rules

It is not easy to understand the behavior of complex sets of rules. The main properties to check are:

- ***Termination.*** A set of rules ensures termination when, for every transaction that triggers the execution of the rules, such execution ends by producing a final state (the local or global abort is also admitted).
- ***Confluence.*** A set of rules guarantees the confluence when, for every transaction that triggers the execution of the rules, such execution ends by producing a single final state, which does not depend on the order of execution of not explicitly prioritized rules.

Properties of active rules

- ***Determinism of the observations***, for each transition that triggers the execution of the rules, the execution is confluent and all visible actions (including messages sent to users) are identical and produced in the same order.

While the termination is an essential property, the other two can remain unsatisfied, especially in the presence of several equivalent solutions of the same application problem.

Analyzing the rules

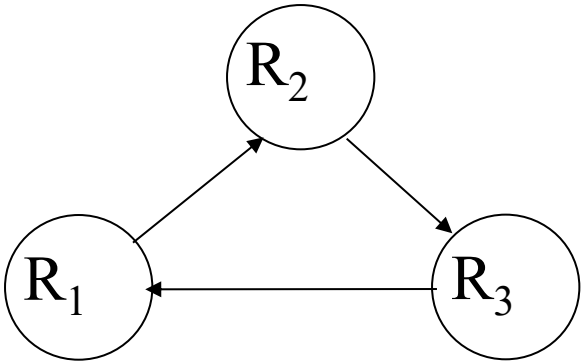
The process of ***analysis of the rules*** allows us to check at the time of creation, that the properties mentioned above are valid for a particular set of rules.

A special tool to check the termination of a set of rules is the ***triggering graph*** where the nodes represent the rules and the edges the dependencies primitive-event between activation rules.

$R_1 \rightarrow R_2$ if the action of R_1 contains a primitive which coincides with one of the events of R_2.

An execution can not terminate only in the presence of cycles in the graph of activation.

Analyzing the rules



The cyclicity of a graph of activation does not always mean problems of non-termination.

Example: The following rule CheckSalaries (Written in DB2), which performs a monitoring of salaries according to a "conservative" policy, reduces the salary of all employees when the average exceeds a certain threshold:

Analyzing the rules

```
CREATE TRIGGER CheckSalaries
AFTER UPDATE OF Salary ON Employee
UPDATE Employee
SET Salary = 0.9 * Salary
WHERE (SELECT avg (salary) FROM Employee)> 100
```

What is the purpose of the trigger?
What is its graph of activation?

Analyzing the rules

```
CREATE TRIGGER CheckSalaries
AFTER UPDATE OF Salary ON Employee
UPDATE Employee
SET Salary = 0.9 * Salary
WHERE (SELECT avg (salary) FROM Employee)> 100
```

The graph of activation of this rule has only one node and a self-loop edge. So it has a cycle.

However, whatever the initial transaction is, the rule execution ends, as there is a progressive reduction of salaries until they fall below the threshold.

Analyzing the rules

The following rule (DB2):

```
CREATE TRIGGER CheckSalaries
AFTER UPDATE OF Salary ON Employee
UPDATE Employee
SET Salary = 1.1 * Salary
WHERE (SELECT avg (salary) FROM Employee)> 100
```

What is happening now?

Analyzing the rules

```
CREATE TRIGGER CheckSalaries
AFTER UPDATE OF Salary ON Employee
UPDATE Employee
SET Salary = 1.1 * Salary
WHERE (SELECT avg (salary) FROM Employee)> 100
```

It has the same graph of activation, but causes non termination problems.

Analyzing the rules

Generally, the analysis of graphs of activation is very conservative; many cycles in the graph of activation does not correspond to non-termination.

Thus the cycles in the graphs only give an indication of any reasons for non-termination.

The designer can focus only on a limited number of cases which appear "dangerous."

Analyzing the rules

There are several techniques to improve the analysis of the rules by the "semantic reasoning", that is based on semantic rules.

For example, an edge $R_i \rightarrow R_j$ can be removed if we guarantee that the new data produced by R_i do not satisfy the condition R_j.

Analyzing the rules

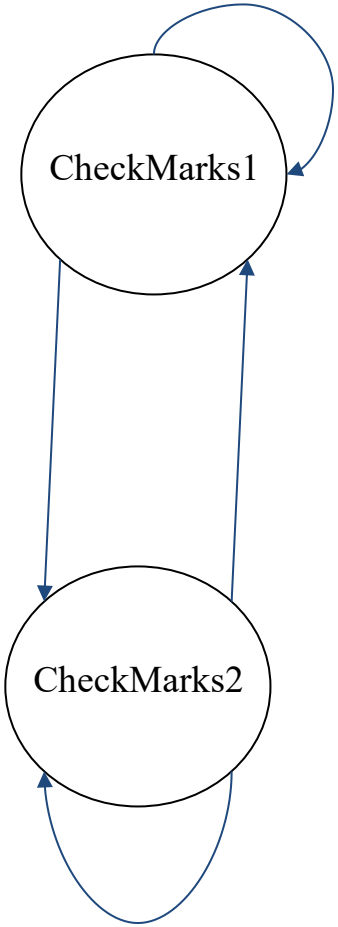
Example: There is a loop between the two following triggers. They control the consistency of a database of tests:

```
CREATE TRIGGER CheckMarks1
AFTER UPDATE OF Mark ON Examination
REFERENCING NEW AS n
FOR EACH ROW
WHEN (n.mark> 30)
    UPDATE Examination
    SET mark= NULL
    WHERE ExamID = n.ExamID
```

```
CREATE TRIGGER CheckMarks2
AFTER UPDATE OF Mark ON Examination
REFERENCING NEW AS n
FOR EACH ROW
WHEN (n.mark < 18)
    UPDATE Examination
    SET mark= NULL
    WHERE ExamID = n.ExamID
```

Do we have non termination?

Analyzing the rules



Analyzing the rules

Example: There is a loop between the two following triggers. They control the consistency of a database of tests:

```
CREATE TRIGGER CheckMarks1
AFTER UPDATE OF Mark ON Examination
REFERENCING NEW AS n
FOR EACH ROW
WHEN (n.mark> 30)
    UPDATE Examination
    SET mark= NULL
    WHERE ExamID = n.ExamID
```

```
CREATE TRIGGER CheckMarks2
AFTER UPDATE OF Mark ON Examination
REFERENCING NEW AS n
FOR EACH ROW
WHEN (n.mark < 18)
    UPDATE Examination
    SET mark= NULL
    WHERE ExamID = n.ExamID
```

The activation of a trigger leads to a state in which the other's condition can not be activated.

We can then remove the edges between the two nodes from the triggering graph.

Confluence

The confluence of statement triggers descends immediately from the termination if the activation rules are totally ordered; In this case, every time several rules are activated, one of them is selected deterministically for consideration and / or execution.

AFTER INSERT OF Exam ... → Trigger1
AFTER INSERT OF Exam ... → Trigger2

Confluence

In most of DBMSs, statement triggers are sorted deterministically by the system (although sometimes the sequence of rules is not easily controlled by the user). It follows that the execution of the statement triggers is normally confluent.

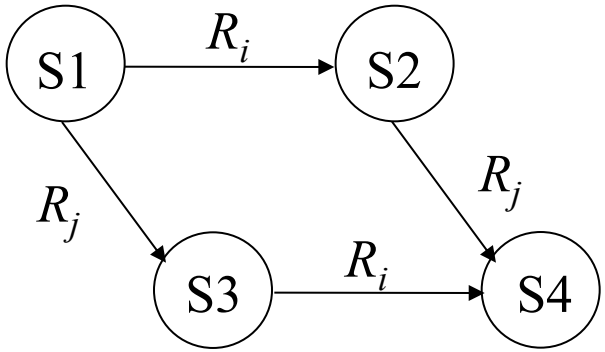
On the contrary, the confluence is a property difficult to check for the tuple triggers. In this case, the confluence requires that each rule produces a final unique state regardless of the sequence in which the tuples are read. Order is typically determined by the system. Difficult to prove!

Sufficient conditions for confluence

for tuple-oriented rules

The basic idea is to determine whether two rules can commute.

It is said that two rules R_i and R_j are ***commutative*** in a set of rules R if, given any state of the database, the consideration of the rule R_i before the rule R_j leads to the same database obtained considering the rule R_j before the rule R_i.



Sufficient conditions for confluence

for tuple-oriented rules

Example:

```
CREATE TRIGGER Ri
AFTER UPDATE OF a1 ON x
FOR EACH ROW
WHEN :new.a1 = 7
    UPDATE x
    SET a2 = a2 + 1
    WHERE x.key = :new.key
```

```
CREATE TRIGGER Rj
AFTER UPDATE OF a1 ON x
FOR EACH ROW
WHEN :new.a1 = 7
    UPDATE x
    SET a2 = a2 + 3
    WHERE x.key = :new.key
```

Sufficient conditions for confluence

for tuple-oriented rules

Given a set of rules for which the termination is guaranteed, a sufficient condition for the confluence requires that all the rules commute.

This condition is too restrictive, since it is sufficient to require the commutativity for a restricted set of rules, whose identification is still complex.

Sufficient conditions for confluence
for tuple-oriented rules

From the application point of view the confluence is **not** always a desirable property: There are many applications where any of the final states produced by a set of terminating rules is also acceptable.

Example: In an application of workflow where each task must be assigned to an employee, it may be acceptable to assign the tasks to the employees aiming only to equally-distribute the overall workload, without actually worry about which employee has received a specific task. In this case, the confluence to an identical final solution is inessential.

Therefore, the use of tuple-oriented rules is better if associated with applications that allow the nondeterminism and discard the confluence.

Determinism of the observations

The determinism of observations is a condition which is stronger than confluence, as it requires that, for any input application, the processing of the rules terminates in the same final state with the same sequence of output.

The examples of output include messages produced by the rules or viewing activities such as reports.

The determinism of the observations is easily obtained for the statement triggers imposing a total ordering on them.

Determinism of the observations

Most applications do not require the determinism of the observations, since the order in which the output is produced is irrelevant.

For example, the following two rules are confluent but do not meet the definition of determinism of the observations because the display reveals the contents of the intermediate states achieved after the execution of one of the two rules.

Determinism of the observations

```
CREATE TRIGGER Ri
AFTER UPDATE OF a1 ON x
FOR EACH ROW
WHEN :new.a1 = 7
    UPDATE x
    SET a2 = a2 + 1
    WHERE x.key = :new.key
     DISPLAY :new.a2
```

```
CREATE TRIGGER Rj
AFTER UPDATE OF a1 ON x
FOR EACH ROW
WHEN :new.a1 = 7
    UPDATE x
    SET a2 = a2 + 3
    WHERE x.key = :new.key
    DISPLAY :new.a2
```

Trigger design: Techniques

To design a trigger we must:

- decide the type of the trigger (row / statement, before / after)
- identify the events
- determine if we need a specific condition and which condition
- determine the action (for integrity violations, in general, is better repair than prevent: to minimize the use of ROLLBACK and raise_application_error)

Trigger design: Techniques

- Before Vs. after:
  - It is better to use a *before* trigger when the trigger action determines whether
    - the event must be actually executed (in this way it avoids to execute the event and then performs a rollback);
    - when we want to verify a change before it occurs and "modify the change" (change of «new»).
  - The mode *after* is the most common, suitable for most applications

Trigger design: Techniques

- Row level vs. statement level:
  - It is better to use a trigger at the level of tuple (finer granularity) if the trigger action depends on the values of the modified tuple.
  - It is better to use statement trigger (coarser granularity) if the trigger action is global for all affected rows (performs a complex verification of authorization, generates a single audit records, calculates aggregate functions).

Trigger design: Techniques for integrity management

More specific considerations can be made relatively to the trigger design for the management of integrity constraints.

This is done in three steps:

1. The constraint should be expressed accurately, as an SQL predicate. Then we can define the part *condition* of one or more triggers associated with the constraint.
2. We consider the *events* that can cause a constraint violation.
3. We decide which *action* to take: transaction rollback, partial rollback of the primitive that caused the constraint violation, correction by a compensating action.

Design: Integrity Management

Consider the referential integrity constraints expressed in this create table command:

```
create table Employee
(
  IDNumber  CHAR (6),
  Name      CHAR (20) not null,
  Surname CHAR (20) not null,
  Dept CHAR (15) REFERENCES Department(DeptName)
  Salary numeric(9) default 0,
 primary key (IDNumber)
 CONSTRAINT DipFK FOREIGN KEY (Dept)
         REFERENCES Department(DeptName)
            ON DELETE SET NULL
         ON UPDATE CASCADE
 UniQue (Surname, Name)
)
```

Design: Integrity Management

The constraint is that for every employee there must exist the department he/she belongs to:

Employee:

```
exists ( select * from Department
         where DeptName = Employee.Dept)
```

so the condition that expresses the constraint violation is as follows:

Employee:

```
not exists ( select * from Department
         where DeptName = Employee.Dept)
```

Design: Integrity Management

The predicate can also be expressed in negated form relatively to tuples of Department:

Department:

```
exists (   select * from Employee
    where Dept not in
   (Select deptName from
  Department))
```

Design: Integrity Management

Operations that may violate this constraint are:

- `insert into Employee;`
- `delete from Department;`
- `update of Employee.Dept;`
- `update of Department.DeptName.`

Design: Integrity Management

Four active rules can then be built:

- Two rules react at each insertion on Employee or to the update of the attribute `Dept`, canceling the effects of the two primitives if they violate the constraint.

```
CREATE TRIGGER DeptRef1
AFTER INSERT ON Employee
FOR EACH ROW
WHEN (not exists
(SELECT * FROM Department
 WHERE Deptname = New.dept))
SIGNAL sqlstate '70006' ( 'employee
    without department');
```

```
CREATE TRIGGER DeptRef2
AFTER UPDATE OF Dept ON
    Employee
FOR EACH ROW
WHEN (not exists
(SELECT * FROM Department
WHERE Deptname = New.dept))
SIGNAL sqlstate '70006' ( 'employee
    without department');
```

Design: Integrity Management

NB: These two triggers are expressed in the language supported by DB2. In PL/SQL we could have written a single trigger.

- A rule responds to the cancellation of rows from Department setting to *null* the attribute value `Dept` for tuples involved:

Design: Integrity Management

```
CREATE TRIGGER DeptRef3
AFTER DELETE ON Department
FOR EACH ROW
WHEN (exists
(SELECT * FROM Employee WHERE Dept = Old.DeptName))
UPDATE Employee
SET dept = null
WHERE dept = Old.DeptName;
```

the condition may even be omitted, since the action is automatically extended to all and only those tuples selected from the condition.

Design: Integrity Management

- A rule reacts to changes of the attribute `DeptName` reproducing the same attribute changes in `Dept.` (taking into account that it constitutes a key to the Employee table)

```
CREATE TRIGGER DeptRef4
AFTER UPDATE OF deptname ON Department
FOR EACH ROW
WHEN (exists
(SELECT * FROM Employee WHERE Dept = Old.DeptName))
UPDATE Employee
SET Dept = New.DeptName
WHERE Dept = Old.DeptName;
```

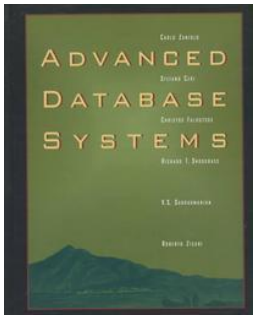
Also in this case, the condition may be omitted.

Problems in the creation of applications for active data bases

- ***Complexity:*** who designs an application must not only understand how the application changes the database directly, but also what is the effect of the triggers activated from that application, in order to know the effects. The situation is even more complex when using active triggers, because they can cause indirect activation of additional triggers. It becomes difficult to understand in what order and in what exact moment the different triggers are executed.
- ***Rigidity***: Typically, it forces the respect of a certain constraint by adopting a certain strategy. The application is forced to look for a way to live with the trigger, because in general, it is not possible to avoid execution.

Example: Automatic cancellation of the departments that have no director.

References



- P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Torlone (2007). *Databases - Architectures and lines of evolution 2 / and*, Chapter 5, McGraw-Hill.

- C. Zaniolo, S. Ceri, C. Faloutsos, RT Snodgrass, VS Subrahmanian, R. Zicari (1997). *Advanced Database Systems*Chapters 2, 3 and 4, Morgan Kaufman.

- S. Ceri, RJ Cochrane, J. Widom (2000). Practical Applications of Triggers and Constraints: Successes and Lingering Issues. *Proceedings of the 26th VLDB Conference*, Cairo, Egypt.

- NW Paton, O. Dìaz (1999). Active Database Systems.*ACM Computing Surveys*, Vol 31, No. 1, pp. 63-103.